Programming Methods
Dr Robert Harle

IA NST CS and CST
Lent 2008/09
Handout 1

Programming Methods

- This is a new course this year
- It is a little experimental
  - In content
  - In approach
- My goals here are:
  - To **consolidate** a lot of this term's programming ideas
  - To bring together elements of Software Design and Java
  - To **demonstrate** some new ideas
  - To make sure you don't think programming is trivial
- This is less about exams and more about understanding
  - There is a "Programming Methods & Java" exam question which will end up being a mix of Java, Programming methods, and Software Design

Programming Methods

- Most of what we do will be worked examples
  - We will be taking particular sample programs and looking at how we get from idea to implementation
  - On our journeys, we will be highlighting various *interesting* points
  - We won't be going through line by line!!
  - Sometimes there will be new material – we won't go in depth
  - You don't need to know anything more than I say in lectures
- These notes will not be a complete record
  - There may be examples, sketches, code reviews
  - You will need to make notes...
- Don't fret over the official syllabus either – the 'syllabus' is whatever I do in lectures
  - If you can give sensible answers to the questions on the examples sheet, you'll be fine.

Different Levels

- The Java ticks have made it very clear that there is a wide variety of programming experience at IA
  - That's why we've moved to Java practicals
  - But I can't really do Lecture*s!
- If you're a Java god
  - You might find some of the coding stuff easy, but *hopefully* the example matter will keep you interested
  - Try to spot where I've cut coding corners for the sake of giving short lectures... and try fixing them!

The Examples

- Don't take them too seriously
  - The code is meant to be short and readable, emphasising specific points. It is *not* perfect.
  - There are deliberate bugs *(for clarity: **all** of my bugs are deliberate) ;-)*
  - You won't always see error checking, exceptions, etc where there should be some. This is just to make the code more readable in lectures.
  - Similarly there won't always be time to follow through the whole design process.
  - It would be a *very* valuable exercise to take the code and improve it in some way.
    - See the exercise sheet

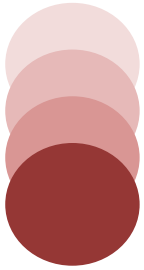
Getting the code

- I encourage you to get the code and play with/improve it
- It's the best way to learn
- After each lecture I will post the relevant code on the course website

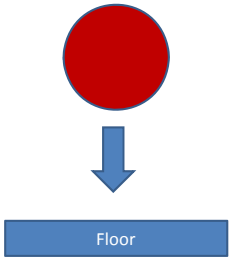
  http://www.cl.cam.ac.uk/teaching/0809/ProgMethod/

Let's start...

- Our first example is especially for the NatScis here
  - Not sure how Java is going to help you in the future?
  - Can't see why CS is part of the NST?
- We're going to make a simulation of a bouncing ball
  - Sounds trivial, but it's actually quite interesting
  - The actual physics we will use is really basic
  - The physics is *not* examinable!
    - But we do need to review it to understand the code



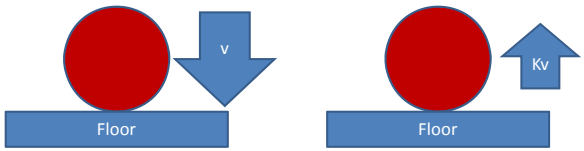
The Setup



- One dimensional motion (yes, I'm lazy)
- Drop the ball from a height
- Simulate what happens

The Physics

- When the ball hits the surface with speed v, it bounces up with speed Kv, where K is the coefficient of restitution
  - K lies between 0 and 1 of course



- The only other thing that acts is gravity

For the Physicists

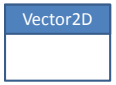
- Calm down – it's just an example
- Yes, I am ignoring:
  - Floor movement
  - Ball compression
  - Ball size (it's a point mass)
  - Wind resistance (!)
  - [insert other minor, almost irrelevant things here]
- Trust me, the world will keep on spinning ;-)

Identify the Classes

The simulation is of a ball that bounces on a floor.

- If we play noun-spotting we end up with three classes: Simulation, Ball, and Floor
- The Simulation will embody everything about our simulated world (time, state, etc).
- The Ball encapsulates the ball properties (K-value), its state (position, speed, acceleration) and how it behaves (motion equations)
- The Floor has a height. And that's it. Giving it its own class is overkill so we choose not to at this stage.

Identify the Classes



- Additionally, we need to represent vector quantities (position, speed, acceleration)
  - A general purpose class for 2D vectors would be very useful not just here but for other projects
  - We abstract out the notion of a vector into Vector2D
  - Vector2D is a *utility* class

- I have also written a Visualisation class which provides us with a graphical depiction of what's going on
  - Don't worry about this code at this stage
  - But do notice that OO programming has allowed me to encapsulate all the ugly graphical stuff into one class that can you can ignore...

Vector2D.java

- Variables to represent the x and y components of a vector quantity [mX, mY]
- We keep the state *private*
  - Private allows us to change the underlying representation if we want
  - E.g. Angle/magnitude or switch to using longs or floats
- But note that we have to take care because the accessors have a 'contract' to return a double
  - If we suddenly want to return longs, we could cause problems
- Is there any reason to change a Vector2D object? Not really.
- So we make it *immutable* (so its state cannot be changed)

Immutability

- We need to make mX and mY private and final
- Then we can have

```
public Vector2D add(Vector2D v) {
   return new Vector2D(mX+v.mX, mY+v.mY);
}
```

✓ Less code in Vector2D.java (no set()s)

✓ v=v1+v2+v3 looks nice: v = v1.add(v2).add(v3);

✗ You have to *know* that Vector2D is immutable to be sure that v.add(w) does not change the state of v or w

✗ v.add(w).multiply(6) appears ambiguous: 6(v+w) or (v+6w)? Note that to Java this is not ambiguous, but to novice programmers it is.

Immutability...

- An alternative is to use use static functions:

```
public static Vector2D add(Vector2D v1, Vector2D v2) {
   return new Vector2D(v1.mX+v2.mX, v1.mY+v2.mY);
}

Public static Vector2D multiply(Vector2D v, double m) {...
```

✓ Multiple operations are not now ambiguous: Vector2D.multiply(Vector2D.add(v,w),6)

✗ ...but ugly!

- If we could somehow enforce that v1 and v2 aren't changed by the above function, it may be a better choice
- But actually we can't in Java!

Ball.java

- Back to our simulation...
- The ball has state
  - Variables for position, velocity, acceleration
  - Variable to hold the value of K
  - Provide accessors and mutators as usual
- The state must change over time
  - The updateState() method implements a simple update to the position and velocity given that some time dt has passed
    - $v = u + a\,dt$
    - $s = u\,dt + \frac{1}{2} a\,dt^2$

There's Never Enough Time...

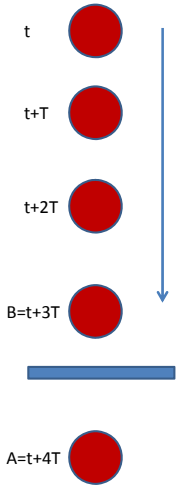
- Real things change over time. Time is continuous (as far as we know). But computers can't deal with continuous things, just discrete things.
- Fundamentally, simulations end up using chunks of time

For constant period T

- So immediately we introduce error
  - Our simulated world can never be the same as our real world.
  - How do we select T? Too big and it's a poor approximation, too small and it will take forever to run our simulation.
  - We choose T=10ms=0.01s

Where does the bounce come in?

- So our simulation runs by computing the ball's state for some time t. Then it computes it for some time (t+T). And then for (t+2T). And so on...
- ***Collision detection***: With each loop or iteration, we look at whether the ball has hit the floor. Either:
  - It will have the same height as the floor, or
  - It will have dropped below the floor since the last step!
- We have to spot this and back track...

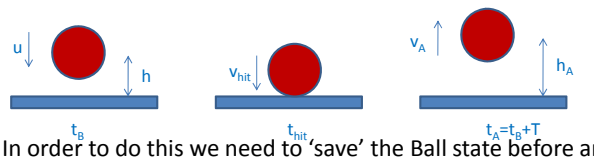


Alternatively...

- We are doing *a-posteriori* event detection
  - 1. Let the simulation run
  - 2. When we see it's gone wrong, undo the last time step and correct it
  - 3. Loop to 1
- Could do it *a-priori*
  - 1. Compute next collision time for the ball, tc
  - 2. When we get to the step that will incorporate tc, execute bounce code
  - 3. Loop to 1
- In principle *a-priori* is more efficient (we don't have to keep checking whether something has gone wrong on each step). However, imagine calculating the next collision time for a box of 1000 balls rattling around: it's pretty complex!
  - *a-posteriori* is more common

Back Tracking for the Bounce

- We record the state at time B (height h, speed u, acceleration a)
- We compute after step B when it would have hit the floor, t_{hit}
  - $h=ut_{hit}+1/2at_{hit}^2$; solve for t_{hit} (and we expect $t_{hit}>0$, $t_{hit}<T$)
- Now the speed it will have hit at
  - $v_{hit} = u+at_{hit}$
- So the ball rebounds with speed Kv_{hit} and has $(T-t_{hit})$ seconds of deceleration before we get to the time for step A
- $h_A=Kv_{hit}(T-t_{hit})+1/2(-g)(T-t_{hit})^2$ and $v_A=Kv_{hit}-g(T-t_{hit})$

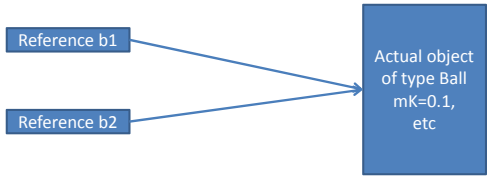


- In order to do this we need to 'save' the Ball state before an update. But how do you copy an object?

Cloning (Aside)

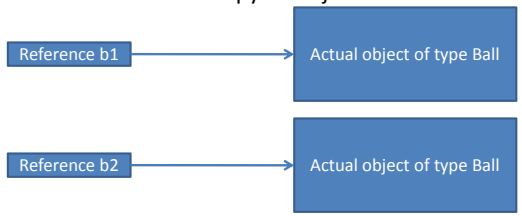
- Java handles objects by reference as you know
- This means we don't accidentally end up copying objects (copying takes time, processing and memory)

```
Ball b1 = new Ball(...);

Ball b2 = b1;
```



Cloning (Aside)

- But what if we do want to copy an object?

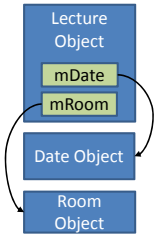


- Everything in Java has a top level parent called **Object**
- And **Object** has a method **clone()**
- Thing is, it's not always clear what it should copy...

Cloning (Aside)

- What does clone() do here for example?

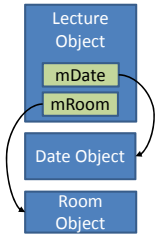
```
public class Lecture {
   private Date mDate;
   private Room mRoom;
   ....
}
```

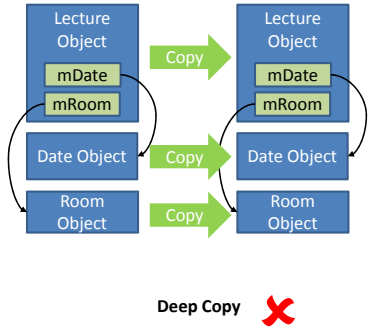


Cloning (Aside)

- What does clone() do here for example?

```
public class Lecture {
   private Date mDate;
   private Room mRoom;
   ....
}
```



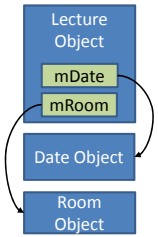


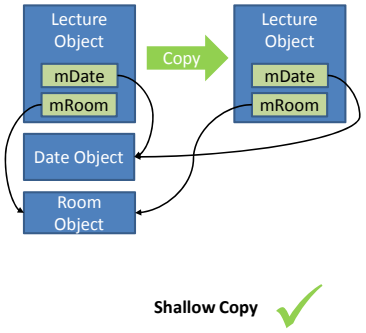
Deep Copy ✗

Cloning (Aside)

- What does clone() do here for example?

```
public class Lecture {
   private Date mDate;
   private Room mRoom;
   ....
}
```





Shallow Copy ✓

Cloning (Aside)

- By default, Java can't know what type of copy is appropriate
  - So there must be some mechanism to tell it what to do
  - How might we have chosen to do this?
    - Use an interface to make the designer write an appropriate **clone()** method

```
public interface Cloneable {
   public Object clone();
}
```

    - The problem with this is that cloning is going to be a relatively expensive operation (lots of memory copies etc)
    - Sun decided that **Object** would implement an optimized ("native") **clone()** method that anything could use to do an efficient **shallow copy**
    - If you want a deep copy, you would normally
      - Use **super.clone()** to get copy of everything above in the tree
      - Add some code to do any deep copy that's needed

Cloning (Aside)

- But Sun worried that programmers might mistakenly just rely on clone() doing the 'right' thing and not think about it
- So they made **clone()** protected and the implementation throws an exception if you try to clone something that does not implement the **Cloneable** interface

Cloning Vector2D

- So to make a Ball cloneable you need to:
  - Implement the **Cloneable** interface

```
public class Ball implements Cloneable {
```

  - Write a *public* clone() method

```
public Object clone() throws CloneNotSupportedException {
   Ball b = (Ball) super.clone();
   b.mPosition = new Vector2D(mPosition.getX(), mPosition.getY());
   b.mVelocity = new Vector2D(mVelocity.getX(), mVelocity.getY());
   b.mAcceleration = new Vector2D(mAcceleration.getX(),
                                  mAcceleration.getY());
   return b;
}
```

  - This will use the efficient **clone()** method in Object which does a **shallow copy** (primitive types get copied whatever you do).

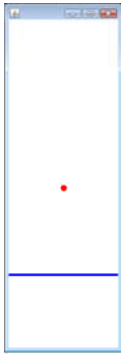
Cloning (Aside)

- The interesting thing is that **Cloneable** doesn't actually have any methods:

```
public interface Cloneable  {
}
```

- It's just used to indicate that you've thought about the issue of cloning
- It's called a **marker interface**
  - All it does is mark a class in some way
  - Java has a number of such marker interfaces in its class library

Try it...



NaN?

- We were doing so well.. and then it did something weird
- Somehow the ball height ends up as NaN?!
- Debugging Tools
  - We can set ***breakpoints***. When we run the program in a debug mode, it will be paused whenever we get to a breakpoint and we can ***inspect*** the program state and variables.
  - Here we will engineer it so that it will break whenever the ball is moved to a height of NaN

```
if (mBall.getPosition().getY()!=mBall.getPosition().getY()) {
    int x=1;
}
```

Break here

  - I use eclipse to do this because it makes it easy, but all it does is use a debugging tool that ships with Java called jdb

What's happening?

- It's our discretized time again...
- No matter what we choose as T, we will end up eventually having multiple bounces per step
  - This breaks our correction step :-(
  - This is a *sampling* problem:

The Fix

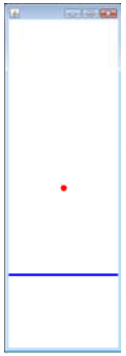
- First we figure out the minimum rebound speed we need the ball to have if it is to stay in the air for at least time T

```
private final double mMinReboundSpeed = 9.81*mDeltaTime/2.0;
```

- Then we make sure that when we correct the ball, it gets at least this speed or we end the simulation

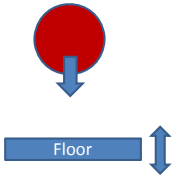
```
// Check that this is a sufficient speed
if (vy<=mMinReboundSpeed) {
        // It's going to bounce again faster that we will be checking!
        // The simulation is not correct any more
        System.exit(0);
}
```

Try it...



Too easy?

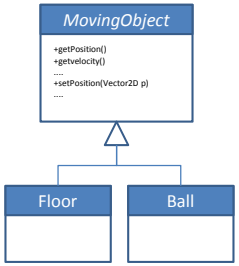
- That was a pretty trivial simulation – many of you could have worked out the trajectory by hand in 5 minutes. So let's make it harder
- Now the floor moves up and down harmonically (think earthquake or loud music)



- Hmmm... Now what happens??!

Alter the simulation

- We have two actors in our problem now
  - The ball and the floor
  - Add a class for the Floor
  - Ball and Floor both move and we will end up duplicating code
    - Use an abstract base class, MovingObject



- Make sure you know why this is abstract and not an interface...
- *MovingObject* also contains code to update the state based on the current state
  - See *applyMotionEquations()*
- Plus there are clone() methods for you to look over

Make the Floor Oscillate

- The floor does not act under gravity, but rather has a constantly changing acceleration
- With each step we must update the state appropriately
  - Harmonic motion
  - $s=A\sin(\omega t)$, $v=A\omega\cos(\omega t)$, $a=-A\omega^2\cos(\omega t)$
  - So do we now have to keep track of the time, t?

```
...
private double mTime=0.0;
...
public void updateState(double delta) {
    mTime+=delta;
    ....
}
```

  - mTime can overflow though. If we can avoid that, we should.

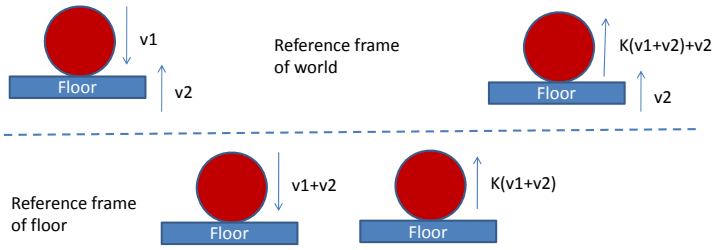
Make the Floor Oscillate

- The phase term (ωt) just increments with each step by (ωT)
- So let's track the phase directly and wrap it at 2π
  - That won't overflow

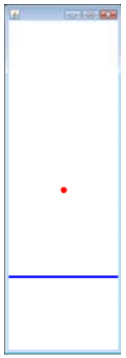
```
private double mPhase = 0.0;
...

@Override
public void updateState(double delta) {
        // phase loops every 1/w s
        // Each dt moves the phase by 2pi*w*dt
        mPhase += mOmega*delta;
        if (mPhase>2.0*Math.PI) mPhase-=2.0*Math.PI;
        this.setAcceleration(new Vector2D(0.0,
              -1.0*mAmplitude*mOmega*mOmega*Math.sin(mPhase)));
        this.applyMotionEquations(delta);
}
```

Adjust the Simulation to Handle the Floor

- Everywhere we had the floor height, we replace with a call to mFloor.getPosition().getY()
- We can now detect collision easily enough, but how does the moving floor affect the rebound speed?
  - We assume that the floor does not accelerate within each timestep
  - i.e. That we have a series of constant velocity frames
  - This is justifiable for small T



Try it...



Parting words…

"Finally, we must admit that a model may confirm our biases and support incorrect intuitions. Therefore, models are most useful when they are used to challenge existing formulations, rather than to validate or verify them. Any scientist who is asked to use a model to verify or validate a predetermined result should be suspicious."

Verification, Validation, and Confirmation of Numerical Models in the Earth Sciences.
Science, 1994, Vol. 263, 5147
Naomi Oreskes, Kristin Shrader-Frechette, Kenneth Belitz