Re-imagining CS1/CS2 with Android Using the Sofia Framework

http://sofia.cs.vt.edu/sigcse2013

Stephen H. Edwards, Virginia Tech

Why?

- **Mobile apps** are popular with everyone (not just CS folks)
- **Android** is popular
- Android uses Java
- … Why not use Android in class?
- Because it's **complicated for beginners**!

The bottom line …

- The Android API is designed for professionals
- Most basic tasks require framework glue:
  - Anonymous inner classes
  - Adapters
  - Extra levels of indirection
  - Type casting, dynamic type checking, instance checking
- All because the API uses well-known techniques that have been around for decades

Android's app lifecycle adds complications, too …

- Android apps can be removed from memory at nearly any time
- Users can switch between applications at any time, possibly never to come back
- Control flow between multiple "screens" of an app requires callbacks and indirection
  - Simple models from window-based desktop applications don't work
  - Neither do naïve student models of understanding

What do we re-imagine?

- What if …
  - We could get rid of all the "**clutter**"?
  - Teach **straight CS1/CS2 concepts** in Java?
  - In the context of Android apps?
  - With a **clean, simple API** students can understand?

We've been working on Sofia

- **S**implified **O**pen **F**ramework for **I**nventive **A**ndroid applications
- A better Android API
- Doesn't just simplify development
- Better abstractions
- Professional quality

At Virginia Tech

- Successful CS2 Integration
  - Past five semesters
  - Students clearly motivated, engaged
  - Testing and automated grading support (Web-CAT)
- We've also pushed into CS1
  - Past two semesters
  - Using a customized version of Greenfoot where all applications also run as Android applications

A sampling of assignments

In CS1:

- Scavenger Hunt
- Maze Runner
- Invasion of the Greeps
- Battleship
- Foxes and Rabbits
- Asteroids
- Build Your Own Game

In CS2:

- Adventure Time!
- Maze Solver
- Mine Sweeper
- Guitar Synthesizer
- "Design Your Own" App

Sofia's design goals

- Not just simpler, but better
  - For beginners and pros alike
- Principle of least astonishment (POLA)
- Convention over configuration
- Don't repeat yourself (DRY)
- Use strong typing smartly
- Extra flexibility through annotations
- It just works (IJW)

How?

- Combine the best ideas from earlier beginner-friendly frameworks
  - Objectdraw
  - JTF
  - Testing
- With new framework design strategies
  - A new event dispatch model
  - Coding by convention
  - Fluent interfaces

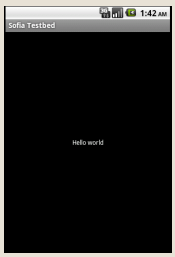
Areas of focused improvements

- Simplified beginner programs
- Approachable graphics/drawing
- Collision detection and physics
- Declarative animation support
- Cleaner multi-activity communication
- Employing multi-touch and gestures
- Common app models (CRUD)
- "Micro world" modeling

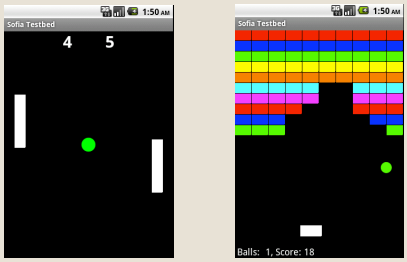
Simplifying intro programs

Can **Hello World** look like this?

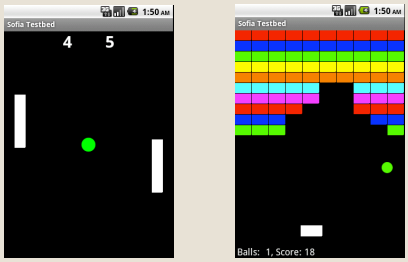
```
public class HelloWorldDemo
    extends ShapeScreen
{
  public void initialize()
  {
    add(new TextShape("Hello world",
        180, 150));
  }
}
```



Shape-based drawing is also important

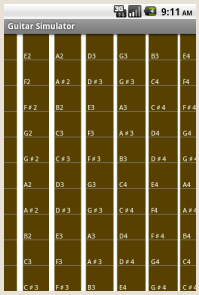


… As is animation support … with collision detection



Multi-activity communication

And even music …



- A guitar fretboard inspired by one of last year's "nifty assignments" at SIGCSE

A New Approach to Event Dispatch in Sofia

Re-imagining CS1/CS2 with Android

What does this have to do with event dispatch?

- Most of the Android API is designed for **pros**, not **beginners**
- Event listeners are ubiquitous, but require several non-beginner language features
- "Events" aren't hard for beginners—it is the **language features around them**

In its simplest form, think of the *Observer* design pattern

- In Java:

```
public interface Observer
{
    void update(Observable o, Object arg);
}
```

In Swing, a MouseListener is a good example

```
public interface MouseListener
{
    void mouseClicked(MouseEvent e);
    void mousePressed(MouseEvent e);
    void mouseReleased(MouseEvent e);
    void mouseEntered(MouseEvent e);
    void mouseExited(MouseEvent e);
}
```

In Java, interfaces are typical

- This provides compile-time advantages
- Presence of handling method(s) on receiver is **checked statically**
- Normal method invocation syntax
- Leverages polymorphism

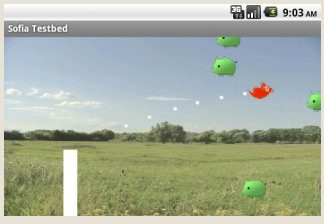
… But there are disadvantages

- Receiver may not need/want **all** of the handling methods in the interface
- The interface must use **more general types** for parameters, to support all possible handlers
- Only **one fixed entry point** for each handling method is supported

An aside about other models

- Objective-C (and Smalltalk before it) uses a **dynamic method lookup** technique: avoids some of the disadvantages, but also some advantage
- **Delegates** in C# are in between, avoiding some of the disadvantages while trying to keep the advantages

Let's look at an example




```
public class Bird
    extends BitmapShape
{
    ...
}
```

```
public class Pig
    extends BitmapShape
{
    ...
}
```

… With interfaces

```
public class IrritatedAvians extends Controller
   implements CollisionListener
{
   public void onCollisionBetween(Shape s1, Shape s2) {
      if (s1 instanceof Bird && s2 instanceof Pig) {
         Bird bird = (Bird) s1;
         Pig pig = (Pig) s2;
         pig.die();
         bird.bounce();
         scoreboard.add(pig.pointValue());
      }
      else if (s1 instanceof Bird && s2 instanceof Brick) {
         ...
      }
   }
   public void onCollisionBetween(Shape s1, ViewEdges e) {
   }
}
```

Even harder for beginners …

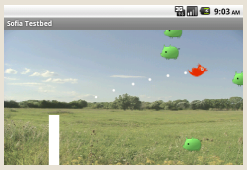
- Event handlers often defined using **anonymous inner classes** that **implement listener interfaces**
- … These serve as "**glue**" to transfer control to behaviors of the surrounding class

The essence of the problem

```
public class IrritatedAvians extends Controller
   implements CollisionListener
{
   public void onCollisionBetween(Shape s1, Shape s2) {
      if (s1 instanceof Bird && s2 instanceof Pig) {
         Bird bird = (Bird) s1;
         Pig pig = (Pig) s2;
         pig.die();
         bird.bounce();
         scoreboard.add(pig.pointValue());
      }
      else if (s1 instanceof Bird && s2 instanceof Brick) {
         ...
      }
   }
   public void onCollisionBetween(Shape s1, ViewEdges e) {
   }
}
```

What if …

```
public class Pig extends BitmapShape {
   ...
   public void onCollisionWith(Bird bird) {
      die();
   }
}

public class Bird
   extends BitmapShape
{
   ...
   public void onCollisionWith(Pig pig) {
      bounce();
   }
   public void onCollisionWith(Board board) {
      ...
   }
}
```

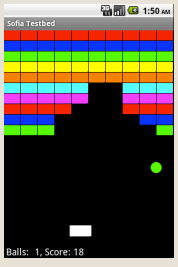


Place *all* the dispatch in the framework

- Use **reflection** to locate handler methods on first use
- Use compiler-like **inheritance search** and **overload resolution** to identify the best match on the receiver
- Cache handlers for better performance
- Leverage **strong typing** to simplify and clean up the design (POLA, coding by convention, IJW)

With any kind of event

```
public class Paddle extends RectangleShape
{
   ...
   public void onTouchMove(MotionEvent e)
   {
      setPosition(CENTER.anchoredAt(
         e.getX(), CENTER.of(this).y));
   }
}

public class Ball extends OvalShape {
   ...
   public void onCollisionWith(Shape shape)
   {
      yVelocity = -yVelocity;
      doAnimation(0);
   }
}
```



This is a "type-centric" approach

- Because **dispatch choices are driven by the types** of the object(s) involved
- We're using Java's type system to **drive** the method search, even though the search is dynamic

… With advantages

- Only provide the handlers you **need**
- Only provide them **where** you need them
- No interface to implement
- No empty method stubs
- No adapters needed
- No extra "glue" to write by hand

… And another advantage

- Use the **specific parameter type** that is most appropriate for your situation
  - Not forced to use the most general type
  - No `instanceof` tests needed
  - No downcasts needed

… And another …

- Can have **multiple handlers** on the same receiver for different types of arguments
  - No `instanceof` tests needed
  - No "internal dispatch code" needed
  - No anonymous inner classes needed

Disadvantages

- Performance
  - Reflective dispatch is more costly than standard method invocation
  - Other search/lookup costs can be minimized
- Gives up static checks against an interface to confirm handler methods are present
  - Still fully type-safe, however

Let's talk!

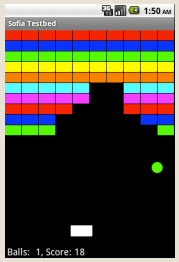
Additional considerations

- Handling multi-object events
- Handling multiple receivers (or multiple handling methods)
- Supporting alternative parameter choices, instead of simply subtyping
- Providing additional name flexibility

Multi-object events

```
public class Ball extends OvalShape {
   ...
   public void onCollisionWith(Brick brick)
   {
      ...
   }

   public void onCollisionWith(
      Set<Brick> bricks)
   {
      yVelocity = -yVelocity;
      doAnimation(0);
   }
}
```

Could use boolean instead

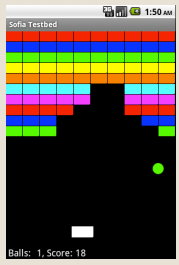


Multiple receivers work the same

- Dispatch to all, allowing each handler to preempt the remaining ones with a boolean return value
- Void methods can be used too (indicating no preemption)

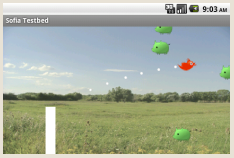
Alternative parameter choices

```
public class Paddle extends RectangleShape
{
   ...
   public void onTouchMove(MotionEvent e)
   {
      ...
   }
}

public class Paddle extends RectangleShape
{
   ...
   public void onTouchMove(int x, int y)
   {
      ...
   }
}
```



Annotations for flexibility

```
public class Pig extends BitmapShape {
   ...
   public void onCollisionWith(Bird bird) {
      die();
   }
}
```



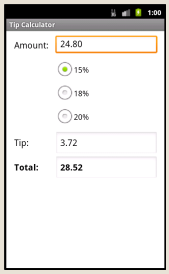

```
public class Pig extends BitmapShape {
   ...
   @Handles("onCollision") Bird.class)
   public void die(Bird bird) {
      ...
   }
}
```

What About Non-Graphical apps?

Re-imagining CS1/CS2 with Android

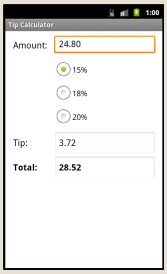
Let's take a look at a "tip calculator"

- We use this as a lab assignment in CS2
- Simple text input
- Radio buttons
- Simple event handling
- Observable



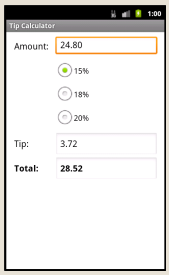
First, an MVC-style "model" class

```
public class TipModel extends sofia.util.Observable
{
    private float billAmount;
    private float tipRate;

    public float getBillAmount()
    {
        return billAmount;
    }

    public float getTipRate()
    {
        return tipRate;
    }

    public float getTipAmount()
    {
        return billAmount * tipRate;
    }

    public float getBillTotal()
    {
        return billAmount + getTipAmount();
    }
}
```

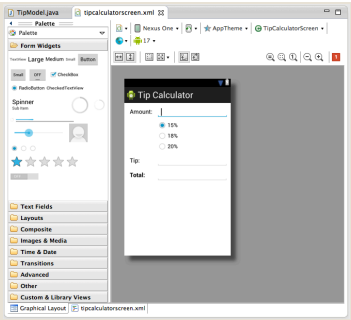


The model includes two mutators

```
public class TipModel extends sofia.util.Observable
{
    ...
    public float setBillAmount(float newBillAmount)
    {
        billAmount = newBillAmount;
        notifyObservers();
    }

    public float setTipRate(float newTipRate)
    {
        tipRate = newTipRate;
        notifyObservers();
    }
}
```

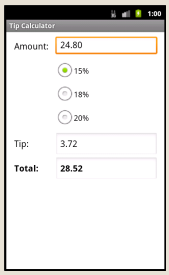


Create the layout graphically



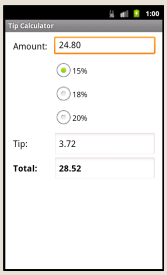
The screen is the MVC "view"

```
public class TipCalculatorScreen extends Screen
{
    private EditText billAmount;
    private EditText tipAmount;
    private EditText billTotal;

    private TipModel tipModel;

    public void initialize()
    {
        tipModel = new TipModel();
        tipModel.addObserver(this);
        tipModel.setTipRate(0.15f);
    }

    ...
}
```



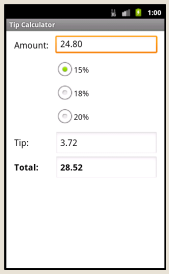
Event handling for the radio buttons

```
public class TipCalculatorScreen extends Screen
{
    ...

    public void tip15Clicked()
    {
        tipModel.setTipRate(0.15f);
    }

    public void tip18Clicked()
    {
        tipModel.setTipRate(0.18f);
    }

    public void tip20Clicked()
    {
        tipModel.setTipRate(0.20f);
    }
}
```



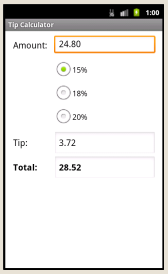
… When the amount is edited

```
public class TipCalculatorScreen extends Screen
{
    ...

    // Called when "done" or "enter" is pressed
    // in the billAmount edit control
    public void billAmountEditingDone()
    {
        float amount = 0.0f;

        try
        {
            Float.parseFloat(
                billAmount.getText().toString());
        }
        catch (NumberFormatException e)
        {
            // Leave amount at 0.0f
        }

        tipModel.setBillAmount(amount);
    }
}
```



… When the model changes

```
public class TipCalculatorScreen extends Screen
{
    ...

    // Called when the model changes
    public void changeWasObserved(TipModel theTipModel)
    {
        String tipAmountString = String.format(
            "%.2f", tipModel.getTipAmount());
        String billTotalString = String.format(
            "%.2f", tipModel.getBillTotal());

        tipAmount.setText(tipAmountString);
        billTotal.setText(billTotalString);
    }
}
```



Contrast with Java's Observable

- In Java:

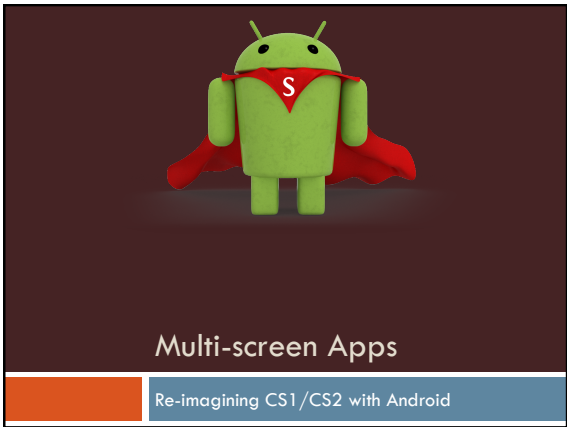
```
public interface Observer
{
    void update(Observable o, Object arg);
}
```

Contrast with Java's `Observable`

- In java.util:

```
public interface Observer
{
    void update(Observable o, Object arg);
}

public class Observable
{
    public void addObserver(Observer o) ...
    public void notifyObservers() ...
    public void notifyObservers(Object arg) ...
}
```

Let's talk!

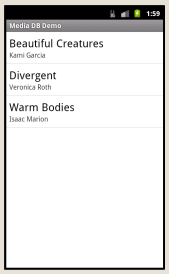


Multi-screen Apps

Re-imagining CS1/CS2 with Android

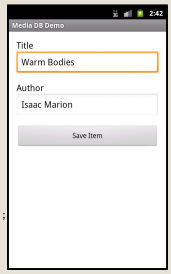
Let's look at a list-driven app

```
@OptionsMenu
public class MediaListScreen extends ListScreen<MediaItem>
{
    ...

    // Called when an item in the list is clicked
    public void ListViewItemClicked(MediaItem item)
    {
        presentScreen(MediaItemScreen.class, item);
    }

    // Called when "add" menu item is clicked
    public void addItemClicked()
    {
        presentScreen(MediaItemScreen.class,
            new MediaItem());
    }

    ...
}
```



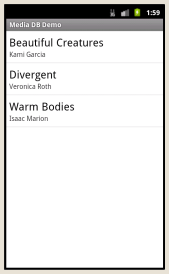
… Which uses an item detail screen

```
public class MediaItemScreen extends Screen
{
    private MediaItem item;
    private EditText itemTitle;
    private EditText itemAuthor;

    public void initialize(MediaItem item)
    {
        item = theItem;

        itemTitle.setText(item.getTitle());
        itemAuthor.setText(item.getAuthor());
    }

    public void saveItemClicked()
    {
        item.setTitle(itemTitle.getText().toString());
        item.setAuthor(itemAuthor.getText().toString());

        finish(item);
    }
}
```



… And back to the list

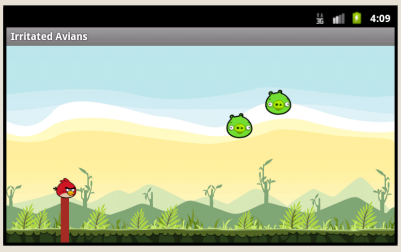
```
@OptionsMenu
public class MediaListScreen extends ListScreen<MediaItem>
{
    ...

    // Called when the item screen finishes
    public void mediaItemScreenFinished(
        MediaItem item)
    {
        if (item.isNew())
        {
            add(item);
            item.clearNew();
        }
    }
}
```



Animation

Re-imagining CS1/CS2 with Android

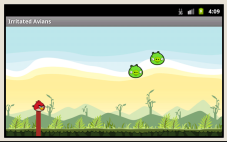
Irritated Avians



- 121 lines in 6 classes (4 are < 15 lines each)

Animation with a "fluent" interface

```
public class Piggy extends OvalShape
{
    ...

    public void onCollisionWith(Bird bird)
    {
        die();
    }

    // "Kills" the piggy by fading it out, making it spin around, and then
    // removing it from the playing field.
    public void die()
    {
        animate(400).alpha(0).rotation(720).removeWhenComplete().play();
    }
}
```



The trail of dots is also animated

```
public class TrailDot extends OvalShape
{
    public TrailDot(Bird bird)
    {
        super(CENTER.of(bird), 0.25f);

        ...

        // Begin an animation with a half-second duration, starting after one
        // second, that fades the dot out and removes it from the field when
        // complete.
        animate(500).delay(1000).alpha(0).removeWhenComplete().play();
    }
}
```

Let's talk!

Bringing Android to CS1 using Greenfoot(4Sofia)

Re-imagining CS1/CS2 with Android

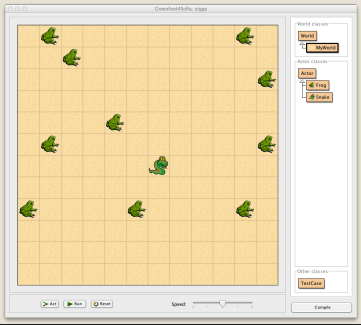
The basic idea …

- Targeted at CS1 students who may have no prior programming experience
- Eclipse and the Android SDK is not the ideal place to start them off
- Need:
  - Simple, graphical programming tasks
  - In a no-experience-necessary environment
  - With direct visual feedback
  - That is still Android-compatible

The details

- **`sofia.micro`**: A Sofia-based package that supports micro-world applications, layered on top of basic 2D shape support
- An open-source fork of Greenfoot
- Completely reimplemented the sofia.micro core under Swing
- Result: **Greenfoot4Sofia** uses Sofia's API for micro-world applications, but apps are retargetable to Android

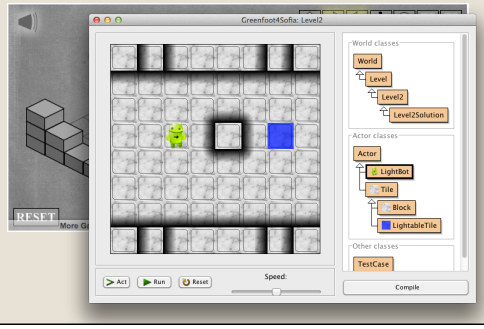
Greenfoot4Sofia



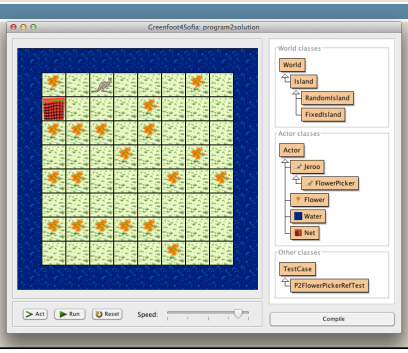
Enhancements to "stock" Greenfoot

- Electronic **project submission** for grading
- **Unit testing** support
- Support for **scenario-specific library classes** that are not provided in source form
- **Event-driven programming** for user interaction
- Support for **sequential logic solutions** when desired, instead of purely cell-automata-like approach

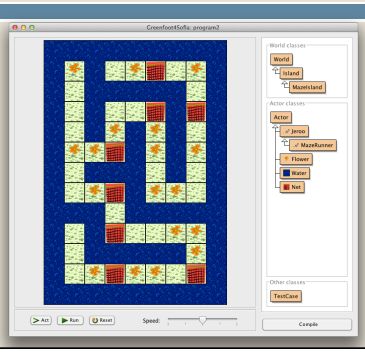
Micro-world: LightBot



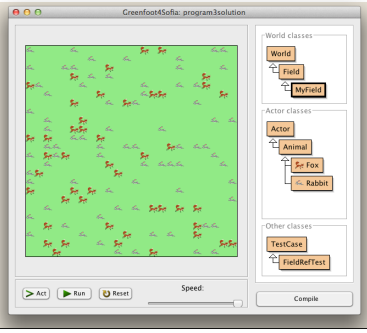
Micro-world: Jeroo



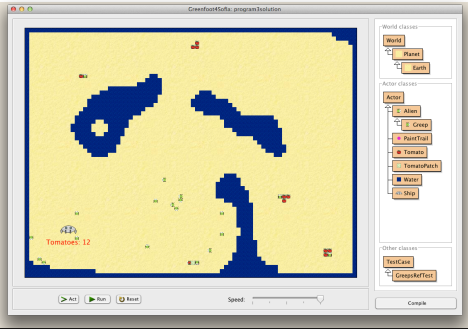
Jeroos on Maze Island



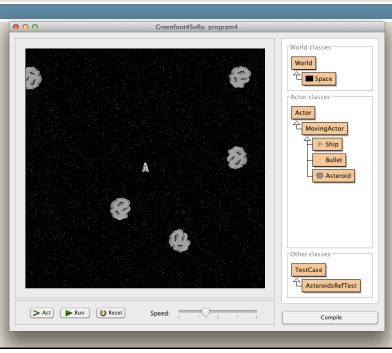
Any Greenfoot-style world



The Greeps contest (from SIGCSE)

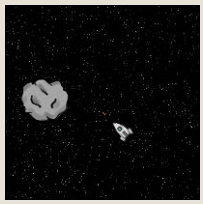


Asteroids



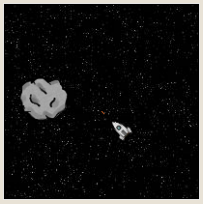
The basics of moving

```
public class Ship extends Actor
{
    private int speed;

    public void act()
    {
        move(speed);
    }

    public void dpadNorthIsDown()
    {
        speed++;
    }

    public void dpadEastIsDown()
    {
        turn(5);
    }

    public void dpadWestIsDown()
    {
        turn(-5);
    }
    ...
}
```



… And collisions

```
public class Asteroid extends MovingActor
{
    ...

    public void act()
    {
        super.move();

        // Did we hit a ship?
        Ship ship =
            getOneIntersectingObject(Ship.class);
        if (ship != null)
        {
            ship.remove();
            this.remove();
        }
    }
}
```



Create your own game

Thank you!

- Talk to me!