Introduction

This tutorial hopes to give the reader a simple introduction to the world of 2D games using Java. We're going to cover the following areas at a fairly simplistic level:

- Accelerated Mode Graphics
- Simple game loops
- Keyboard Input
- Brute force collision detection

The final game can be see **here**. The complete source for the tutorial can be found **here**. Its intended that you read through this tutorial with the source code at your side. The tutorial isn't going to cover every line of code but should give you enough to fully understand how it works.
Context highlighted source is also available here:
Game.java
SpriteStore.java
Sprite.java
Entity.java
ShipEntity.java
ShotEntity.java
AlienEntity.java

Disclaimer: This tutorial is provided as is. I don't guarantee that the provided source is perfect or that that it provides best practices.

Design

Before starting any game its always a good idea to work out what's going to be in the game and how you're going to build your classes around that. You don't need to read this section unless you're interested in why the source is written the way it is.
For our space invaders game we're going to have a main window. The window needs to use acceleated graphics. It also needs to respond to the player's key presses to move our player's ship around. For now we can call this class **Game** since it represents our main game.

In that window we want to see some things moving around. The player's ship, the aliens and the shots that the players fire. Since all of these things have common properites (i.e. they display a graphic and move around the screen) we can infer a common class to represent each one with potentially subclasses to define the specific behaviours of these different types. Since "thing" is such a terrible name for a class, for our design I'm going to call them Entities. From this we get 4 classes. **Entity** with 3 subclasses, **ShipEntity**, **AlienEntity** and **ShotEntity**

Finally for each Entity we have we'd like to have an image displayed, using an old term, a Sprite. However, we might use the same Sprite for multiple entities, for instance the aliens. It seems logically therefore to keep the sprite as a seperate object from the entity. In addition we don't want to waste graphics memory so we'd like to only load each sprite once. To manage this it'd be nice to add a class to manage loading of the Sprites and storing them for future use. So we add a pair of classes to our design, **Sprite** and **SpriteStore**.

The Basic Window

Our basic window is going to be created and maintained by a central class, Game. The following sections cover the initial sections of code in the main class.

Game Entry Point

In java our entry point is "public static void main(String arg[])". This is where the application starts when its run. From here we're going to create an instance of our main class which will start everything else running. Game will be a subclass of Canvas, since it will be the main element displaying the graphics. Note, that it needs to be a subclass of Canvas since its one of the only components that supports using accelerated graphics.

```
public static void main(String argv[]) {
        Game g = new Game();
        g.gameLoop();
}
```

Creating the window

First we need to create our window and configure its contents. We're going to fix our resolution to 800x600. However, since the window may have decoration the content must be set to 800x600 and we must rely on pack() (shown a little later) to actually size the window appropriately.

```
// create a frame to contain our game

JFrame container = new JFrame("Space Invaders 101");

// get hold the content of the frame and set up the
// resolution of the game
JPanel panel = (JPanel) container.getContentPane();
panel.setPreferredSize(new Dimension(800,600));
panel.setLayout(null);
```

Projects
- Completed Games
- Libraries and Components
- Tools and Utilities
- Tutorials
  - Space Invaders - Rendering in Java
    - Space Invaders 101 - An

```
        // setup our canvas size and put it into the content of the frame
        setBounds(0,0,800,600);
        panel.add(this);
```

Since the canvas we're working with is going to be actively redrawn (i.e. accelerated graphics) we need to prevent Java AWT attempting to redraw our
surface. Finally, we get the window to resolve its size, prevent the
user resizing it and make it visible.

```
        // Tell AWT not to bother repainting our canvas since we're
        // going to do that our self in accelerated mode
        setIgnoreRepaint(true);

        // finally make the window visible
        container.pack();
        container.setResizable(false);
        container.setVisible(true);
```

Accelerated Graphics

To manage our accelerated graphics canvas we're going to rely on a class provided from the JDK. The BufferStrategy is named that because its a strategy for managing buffers, or rather the swapping of buffers. This supports us using page flipping and accelerated graphics.

Creating a BufferStrategy couldn't be simpler. We simply ask the Canvas to do it for us. The only thing that needs to be specified is how many buffers to use to manage the screen, in this case we're going to use just 2.

```
        // create the buffering strategy which will allow AWT
        // to manage our accelerated graphics
        createBufferStrategy(2);
        strategy = getBufferStrategy();
```

The Game Loop

The game loop is a remarkably important part of any game. Client side games tend to be single threaded. This helps prevent a whole bunch of complexities synchronising game updates and game drawing. Generally this results in more stable and maintainble code. There are good arguments to use threading, however for the purposes of this tutorial we're not going to consider it.

The normal game loop runs something like this:

- Work out how long its been since we last looped
- Process any input from the user/player
- Move everything based on the time since last loop
- Draw everything on screen
- Swap the buffers over to make the new image visible
- Wait for a set period

At this stage we're just interested in getting the screen swap and timing done, so we add a function called gameLoop that does this:

```
        while (gameRunning) {
                // work out how long its been since the last update, this
                // will be used to calculate how far the entities should
                // move this loop
                long delta = System.currentTimeMillis() - lastLoopTime;
                lastLoopTime = System.currentTimeMillis();

                // Get hold of a graphics context for the accelerated
                // surface and blank it out
                Graphics2D g = (Graphics2D) strategy.getDrawGraphics();
                g.setColor(Color.black);
                g.fillRect(0,0,800,600);

                // finally, we've completed drawing so clear up the graphics
                // and flip the buffer over
                g.dispose();
                strategy.show();

                // finally pause for a bit. Note: this should run us at about
                // 100 fps but on windows this might vary each loop due to
                // a bad implementation of timer
                try { Thread.sleep(10); } catch (Exception e) {}
        }
```

Finally we call gameLoop from the bottom of the constructor of Game. This just starts the game loop running around. If you've been following this code you should be able to run what you currently have and a window should be displayed with an accelerated canvas that shows a black screen.

Sprites and Resource Management

The Sprite class

The sprite class will act as a wrapper round the standard java.awt.Image. However, its going to give us a place holder to expand things later on.
So heres the sprite class. Essentially it just takes and holds an image which can be drawn at a specified location onto a graphics context.

```
public class Sprite {
	/** The image to be drawn for this sprite */
	private Image image;

	/**
	 * Create a new sprite based on an image
	 *
	 * @param image The image that is this sprite
	 */
	public Sprite(Image image) {
		this.image = image;
	}

	/**
	 * Get the width of the drawn sprite
	 *
	 * @return The width in pixels of this sprite
	 */
	public int getWidth() {
		return image.getWidth(null);
	}

	/**
	 * Get the height of the drawn sprite
	 *
	 * @return The height in pixels of this sprite
	 */
	public int getHeight() {
		return image.getHeight(null);
	}

	/**
	 * Draw the sprite onto the graphics context provided
	 *
	 * @param g The graphics context on which to draw the sprite
	 * @param x The x location at which to draw the sprite
	 * @param y The y location at which to draw the sprite
	 */
	public void draw(Graphics g,int x,int y) {
		g.drawImage(image,x,y,null);
	}
}
```

Sprite Loading and Management

An important part of any game is managing the resources that are used. In our case we're just looking at sprites. However, we do only want to load any single sprite once. It would also be handy if the collection of sprites existing in one central location. With this in mind we'll implement a SpriteStore object. This will be responsible for loading and caching sprites. To make it easy to get hold of the sprite store we'll implement it as a singleton.

Implementing the singleton

A singleton simply means thats theres a single instance of the object available statically. In many cases this is done simply for convienience, however sometimes there are good design reasons for enforcing that there is only a single instance of the class ever created.
Implementing this in our sprite store looks like this:

```
/** The single instance of this class */
private static SpriteStore single = new SpriteStore();

/**
 * Get the single instance of this class
 *
 * @return The single instance of this class
 */
public static SpriteStore get() {
	return single;
}
```

The "single" is the single instance of the class ever created. The static get method gives us access to the single instance.

Loading the Sprites

The first step is to retrieve the sprite image from disk. Generally in Java it makes sense to use the ClassLoader to find the image for you. This makes it much easier to deploy games with the resources packaged up. However, some people are more comfortable to use direct file access which is also perfectly viable. For this tutorial we'll stick to the ClassLoader since it makes webstart possible.
The first step is locate the sprite:

```
URL url = this.getClass().getClassLoader().getResource(ref);
```

Its important to note that the chain of functions used to get to the class is only there to support specialised class loaders (like the one found in WebStart). The retrieved URL will point to the image specified in the string "ref" relative to the classpath.

The next step is to actually load in the image. In Java this is a simple matter of using the utlity class ImageIO. To load our image we use this code:

```
        sourceImage = ImageIO.read(url);
```

Finally, we need to allocate some accelerated graphics memory to store our image in. This will allow the image to be drawn without the CPU getting involved, we just want the graphics card to do the work for us.

Allocating the graphics memory is achieved like so:

```
        // create an accelerated image of the right size to store our sprite in
        GraphicsConfiguration gc = GraphicsEnvironment.getLocalGraphicsEnvironment().
                                    getDefaultScreenDevice().getDefaultConfiguration();
        Image image = gc.createCompatibleImage(sourceImage.getWidth(),
                                               sourceImage.getHeight(),
                                               Transparency.BITMASK);
```

The final step is to draw our loaded image into our accelerated graphics image. This essentially creates our sprite:

```
        // draw our source image into the accelerated image
        image.getGraphics().drawImage(sourceImage,0,0,null);
```

The only thing we have left to do is create our actual Sprite object.

Caching the Sprites

The other responsibility of our SpriteStore is to cahce the images that have been loaded. To achieve this we add a HashMap. This map will link the string references to images to the sprites that our loaded. Whenever we load a sprite we add it to the map like this:

```
        // create a sprite, add it the cache then return it
        Sprite sprite = new Sprite(image);
        sprites.put(ref,sprite);
```

Likewise, whenever a sprite is requested we check whether its already in the map. If it is, we return our cached copy instead of loading a new copy:

```
        // if we've already got the sprite in the cache
        // then just return the existing version
        if (sprites.get(ref) != null) {
                return (Sprite) sprites.get(ref);
        }
```

Movement

Lets get things moving! Movement in our case is going to be handled by our Entity class. Each loop we'll ask the entities in the game to move themself. To facilitate this we add a list of entities to the main game class. Each loop we'll cycle round this list asking each entity to move and then draw itself. In our game loop, that looks like this:

```
        // cycle round asking each entity to move itself
        for (int i=0;i<entities.size();i++) {
                Entity entity = (Entity) entities.get(i);

                entity.move(delta);
        }

        // cycle round drawing all the entities we have in the game
        for (int i=0;i<entities.size();i++) {
                Entity entity = (Entity) entities.get(i);

                entity.draw(g);
        }
```

Rememebr we calculated how long it'd been since we looped round. This can be used to work out how far an entity should move on the current loop. Right, now we know what Entity needs to be able to do, on to the implementation.

The Entity class

The entity class will contain its currently location, its current movement and its visual representation. When an Entity is constructed these values will be defined. This includes retrieving the sprite from the store that should represent this entity. The location and movement associated with the entity will be definable and retriveable via get and set methods.
Once these properties are defined we can cover the two methods that we require of Entity. The move() method looks like this:

```
        public void move(long delta) {
                // update the location of the entity based on move speeds
                x += (delta * dx) / 1000;
                y += (delta * dy) / 1000;
        }
```

Simple! We take how every much time has passed, multiply it by the movement in each direction and add this on to the location. The division by 1000 is to adjust for the fact that the movement value is specified in pixels per second, but the time is specified in milliseconds. Each loop all the entities will be moved in accordance with their currently movement values (velocities).

Next we need to be able to draw an Entity onto our accelerated graphics context. draw() is implemented like this:

```
	public void draw(Graphics g) {
		sprite.draw(g,(int) x,(int) y);
	}
```

Essentially, this just draws the sprite onto the supplied graphics context at its current location. So each loop, the entity moves and is then redrawn at the right location.

Now we've defined our basic entity we should create a few subclasses as placeholders for some more functionality we'll add later on. We simply need to create the 3 subclasses of Entity; ShipEntity, ShotEntity and AlienEntity. For now we won't bother adding anything extra but it normally pays to be aware and add these things up front.

The final step is to create our entities and add them to the game world. If we add a utility method to central Game class called initEntities(). This will initialise a set of entities at the game start. The current implementation looks like this:

```
	private void initEntities() {
		// create the player ship and place it roughly in the center of the screen
		ship = new ShipEntity(this,"sprites/ship.gif",370,550);
		entities.add(ship);

		// create a block of aliens (5 rows, by 12 aliens, spaced evenly)
		alienCount = 0;
		for (int row=0;row<5;row++) {
			for (int x=0;x<12;x++) {
				Entity alien = new AlienEntity(this,
							"sprites/alien.gif",
							100+(x*50),
							(50)+row*30);
				entities.add(alien);
				alienCount++;
			}
		}
	}
```

As you can see, the initialisation takes two steps. The first is to create the player's ship. We simply create a ShipEntity with the appropriate graphic and center it at the bottom of our canvas.

The second step is to create all the aliens. We loop through creating a block of aliens. Again each alien is just the creation of the AlienEntity positioned at the right location. In addition we count how many aliens we've created so we can track whether the player has won the game.

Assuming the code to support moving and displaying the entities has been added to main game loop, running the game should now show the player's ship and a bunch of aliens.

Now each type of Entity moves in its own way and with its own contraints. Lets look at each one.

Ship Entity

Since the ship will be controlled by the player (see a little lower down) we have very little to do here. However, we do want to prevent the ship moving off the sides of the screen so we add this bit of code to the move() method in ShipEntity:

```
	public void move(long delta) {
		// if we're moving left and have reached the left hand side
		// of the screen, don't move
		if ((dx < 0) && (x < 10)) {
			return;
		}
		// if we're moving right and have reached the right hand side
		// of the screen, don't move
		if ((dx > 0) && (x > 750)) {
			return;
		}

		super.move(delta);
	}
```

What we essentially are saying here is that if we're moving left and we're about to move off the left hand side of the screen then don't allow the movement (i.e. return). In reverse if we're moving to the right and are about to move off the right hand side of the screen then don't allow the movement. Otherwise we just do the normal Entity movement routine.

Shot Entity

The shot entity is pretty simple, it just wants to run up the screen until it either hits an alien (see Collision later on) or runs off the top of the screen, at which point we'd like to remove it from the entity list (for details of the remove entity method check out the source).

To start the shot moving with initialise the vertical movement to a negative number based on the speed we'd like the shot to move. The movement method itself looks like this:

```
	public void move(long delta) {
		// proceed with normal move
		super.move(delta);

		// if we shot off the screen, remove ourselfs
		if (y < -100) {
			game.removeEntity(this);
		}
	}
```

Simply put, if the shot moves off the top of the screen, remove it.

Alien Entity

Aliens are the most tricky part of our space invaders game. As they move around we need to notice when they hit the side of the screen and start them moving in the opposite direction. In conjuction each time they change direction we'd like them all to move down a step. Part of this will be covered in the movement routine and part in the game logic. Game logic is used in this case since we need to first detect that the aliens should change direction then change them all (rather than a localised change like the other entities)

Hopefully, we now know what we want to do, so how? First we initialise the movement of the aliens to start them moving to the left based on the predefined speed. Next we put the detection of an alien hitting the side in the movement routine like this:

```
	public void move(long delta) {
		// if we have reached the left hand side of the screen and
		// are moving left then request a logic update
		if ((dx < 0) && (x < 10)) {
			game.updateLogic();
		}
		// and vice vesa, if we have reached the right hand side of
		// the screen and are moving right, request a logic update
		if ((dx > 0) && (x > 750)) {
			game.updateLogic();

		}

		// proceed with normal move
		super.move(delta);
	}
```

In the same way as in ShipEntity, we check whether the entity has hit the edge of the screen. However, in this case we notify the game that the game logic for all entities need to be run. We'll make this logic adapt the movement of the aliens but more on this later.

Keyboard Input

Our next step is to make the ship controllable. To do this we need to add a simple inner class to our main game. It looks like this:

```
	private class KeyInputHandler extends KeyAdapter {

		public void keyPressed(KeyEvent e) {
			if (e.getKeyCode() == KeyEvent.VK_LEFT) {
				leftPressed = true;
			}
			if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
				rightPressed = true;
			}
			if (e.getKeyCode() == KeyEvent.VK_SPACE) {
				firePressed = true;
			}
		}


		public void keyReleased(KeyEvent e) {
			if (e.getKeyCode() == KeyEvent.VK_LEFT) {
				leftPressed = false;
			}
			if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
				rightPressed = false;
			}
			if (e.getKeyCode() == KeyEvent.VK_SPACE) {
				firePressed = false;
			}
		}

		public void keyTyped(KeyEvent e) {
			// if we hit escape, then quit the game
			if (e.getKeyChar() == 27) {
				System.exit(0);
			}
		}
	}
```

This class simply picks up keys being pressed and released and records their states in a set of boolean variables in the main class. In addition it checks for escape being pressed (which in our case exits the game)

To make this class work we need to add it to our canvas as a "KeyListener". This is done by adding the following line to the constructor of our Game class:

```
	addKeyListener(new KeyInputHandler());
```

With this source code in place the boolean flags will be set to appropriate values as the keys are pressed. The next step is to respond to these boolean flags being set in the game loop. If we add this in the game loop we can add keyboard control to the ship:

```
	// resolve the movement of the ship. First assume the ship
	// isn't moving. If either cursor key is pressed then
	// update the movement appropraitely
	ship.setHorizontalMovement(0);

	if ((leftPressed) && (!rightPressed)) {
```

```
                ship.setHorizontalMovement(-moveSpeed);
        } else if ((rightPressed) && (!leftPressed)) {
                ship.setHorizontalMovement(moveSpeed);
        }
```

As you can see we check if left or right is pressed. If they are we update the movement values associated with the player's ship. Hence when we hit the entity movement code the ship will move left or right.

If the player holds the fire key we'd like the ship to fire a shot up towards the aliens. However, we don't want to let the player just keep firing. It'd be good to limit how often they can fire. We'll put this functionality in a utility method called (somewhat logically) "tryToFire()"

```
        // if we're pressing fire, attempt to fire
        if (firePressed) {
                tryToFire();
        }
```

To prevent the player firing too often we'll record the time whenever they take a shot and prevent them taking a shot if the last shot was less than a set interval ago. Hmm, sounds complicated, its not! really! Here it is:

```
        public void tryToFire() {
                // check that we have waiting long enough to fire
                if (System.currentTimeMillis() - lastFire < firingInterval) {
                        return;
                }

                // if we waited long enough, create the shot entity, and record the time.
                lastFire = System.currentTimeMillis();
                ShotEntity shot = new ShotEntity(this,"sprites/shot.gif",ship.getX()+10,ship.getY()-30);
                entities.add(shot);
        }
```

First we check if the last time the player took a shot was long enough ago. If it wasn't we don't bother firing and just return. If it was we record the current time. Next we create and add an entity to represent the shot fired by the player. Since our shot is setup to move up the screen it shoots off from the player towards the aliens.

Collision Detection

Lets start by saying there are better ways to implement collision detection than the method outlined here. It could be described a "brute force". In our main game loop we're going to cycle round check whether each entity has collided with every other entity.
First, we'll need to implement a check to resolve whether two entities have in fact collided. We'll do this in the Entity class like this:

```
        public boolean collidesWith(Entity other) {
                me.setBounds((int) x,(int) y,sprite.getWidth(),sprite.getHeight());
                him.setBounds((int) other.x,(int) other.y,other.sprite.getWidth(),other.sprite.getHeight());

                return me.intersects(him);
        }
```

This method checks if the entity itself collides with the other entity specified. We're going to rely on rectangular intersection regions and the AWT class Rectangle. In this case the variables "me" and "him" are instances of Rectangle that are held at the class level.

First we configure the two rectangles to represent the two entities. Next we use the inbuilt functionality of java.awt.Rectangle to check if the two entities intersect with each other. This isn't the smartest way to do this by any means, but for our purposes it will be good enough.

The next thing we'll add is a way to notify entities that they have collided with another. To do this we'll add a method like this to the Entity class:

```
        public abstract void collidedWith(Entity other);
```

Its been made abstract since different implementations of the Entity class will want to respond to collisions in their own ways, e.g. Alien<->Shot, Ship<->Alien.

ShipEntity

The ship entity needs to react when it hits an entity, i.e. The player should be killed. To facilitate this we'll should do this:

```
        public void collidedWith(Entity other) {
                // if its an alien, notify the game that the player
                // is dead
                if (other instanceof AlienEntity) {
                        game.notifyDeath();
                }
        }
```

Again, the result of the collision is based on the game logic (covered later). If the entity that the ship collided with is an Alien then notify the game that player should die.

ShotEntity

When the shot entity hits an alien we want the alien and the shot to be destroyed. This is achieved with the following code:

```
        public void collidedWith(Entity other) {
```

```
		// if we've hit an alien, kill it!
		if (other instanceof AlienEntity) {
			// remove the affected entities
			game.removeEntity(this);
			game.removeEntity(other);
			
			// notify the game that the alien has been killed
			game.notifyAlienKilled();
		}
	}
```

If the shot hit and alien then the alien and shot are removed. In addition the game logic is notified that the an alien has been killed (covered later).

Game Loop Additions

The final step in getting the collision detection to work is to add a section to the game loop to cycle through all the entities checking whether they collide with each other. Here's the code:

```
	// brute force collisions, compare every entity against
	// every other entity. If any of them collide notify
	// both entities that the collision has occured
	for (int p=0;p<entities.size();p++) {
		for (int s=p+1;s<entities.size();s++) {
			Entity me = (Entity) entities.get(p);
			Entity him = (Entity) entities.get(s);
			
			if (me.collidesWith(him)) {
				me.collidedWith(him);
				him.collidedWith(me);
			}
		}
	}
```

For each entity we cycle through all the other entities that we haven't compared against and check whether a collision has occured. If a collision has occured we notify both sides.

A smarter way to do this might be to check for the collisions only every so often. It might also be nice to have a flag on an entity whether it should detect collisions.

Game Logic

The final step of our game is to fill the game logic parts. We're implied the existence of a set of methods on the Game class already. Time to work out what they need to do.

notifyDeath()

This method is called to indicate that the player has been killed. This can happen if an alien hits the player or an alien goes off the bottom of the screen.

notifyWin()

This method id called to indicate that the player has won the game. In the current game this is a result of killing all the aliens.

notifyAlienKilled()

Calling this method indicates that an alien has been killed as a result of a collision registered by the ShotEntity. Once all the aliens have been killed the player has won. The code looks like this:

```
	public void notifyAlienKilled() {
		// reduce the alient count, if there are none left, the player has won!
		alienCount--;
		
		if (alienCount == 0) {
			notifyWin();
		}
		
		// if there are still some aliens left then they all need to get faster, so
		// speed up all the existing aliens
		for (int i=0;i<entities.size();i++) {
			Entity entity = (Entity) entities.get(i);
			
			if (entity instanceof AlienEntity) {
				// speed up by 2%
				entity.setHorizontalMovement(entity.getHorizontalMovement() * 1.02);
			}
		}
	}
```

In addition to recording if all the aliens have been killed we also speed the aliens up a bit. Every time an alien is killed the rest speed up by 2%.

The last step we need is to support entity based game logic. We implied the requirement when building the AlienEntity. When a single alien detects the edge of the screen we want all the aliens to change direction. To support this we're going to add two sections of code.

Entity Logic
Each entity should be allowed to support its own logic. To facilitate this we going to add a method to the Entity class.

doLogic() will allow subclasses of Entity to define a bit of logic that will be run whenever game logic is requested (see below). In AlienEntity for instance we request that game logic be run when the edge of the screen is detected. The doLogic() implementation in AlienEntity looks like this:

```
public void doLogic() {
        // swap over horizontal movement and move down the
        // screen a bit
        dx = -dx;
        y += 10;

        // if we've reached the bottom of the screen then the player
        // dies

        if (y > 570) {
                game.notifyDeath();
        }
}
```

So, when an alien detects the edge of the screen it signals that game logic should be run on entities. The alien entities will change direction (dx = -dx) and move down the screen a bit (y += 10). Finally, if the alien has moved off the bottom of the screen then notify the game that the player is dead.

Entity Logic Infrastructure

To complete the game logic at the entity level we need to add a method on the Game class that will indicate that the entity game logic should be run. First, we add this method on Game:

```
public void updateLogic() {
        logicRequiredThisLoop = true;
}
```

This flag indicates that in the game loop we should run the logic associated with every entity currently in the game. To achieve this we add this in the game loop:

```
// if a game event has indicated that game logic should
// be resolved, cycle round every entity requesting that
// their personal logic should be considered.
if (logicRequiredThisLoop) {
        for (int i=0;i<entities.size();i++) {
                Entity entity = (Entity) entities.get(i);
                entity.doLogic();
        }

        logicRequiredThisLoop = false;
}
```

If the flag is set, then cycle round the entities calling the doLogic() method on each. Finally, reset the flag so the logic doesn't automatically get run next loop.

Finishing Off

Hopefully this tutorial has been of some help. If you look through the provided source code you'll find a selection of additions which complete the game more fully. These arn't covered in the tutorial because of their intricacy. However, the comments in the code should make the extra bits and pieces easy to understand.

If you have any comments or corrects feel free to mail me **here**

Exercises for the Reader

The following extensions of the game may help in understanding how game hangs together.

Frame Rate Counter

Add a frame rate counter to the screen while the game is playing. This could be done by counting game loops and recording the number counted each second. This value could then be displayed as part of the drawing section of the game loop.

Animated Aliens

Animating the aliens would require updating the Entity class to support multiple sprites. Then depending on the time passed the sprite being drawn for the entity could be swapped.

Alien Return Fire

With the addition of the a new Entity the aliens could be made to fire back. This could either be done by timing it so each alien fires back every so many seconds. Alternatively, this could be based on a random occurance, say the alien has a 1 in a 100 chance of fireing back at any given moement.

Pause Button

Make it possible to pause the game by pressing a key (P?). This should pause all game updates but still continue to render the screen and the entities within the game. It might be nice to add a message indicating the game is paused.

Further Reading

I'd recommend you take a look at the following material, it'll help support taking the current code to the next stage.

- **A HowTo Guide for WebStart**
- **The SUN fullscreen tutorial trail**
- **The GAGE Library (Most importantly the high resolution timer)**

Possible Future Tutorials

Let me know if you'd like to see any of the following tutorials in the future.

- Space Invaders in OpenGL
- Multiplayer Space Invaders
- Using sprite based Fonts
- Music and Sound Effects
- Space Invaders with 3D Models (Xith3D)

Credits

Tutorial and Source written by **Kevin Glass**
Game sprites provided by **Ari Feldman**
A large number of people over at the **Java Gaming Forums**

previous up next

Space Invaders - 2D Rendering in Java Space Invaders 102 - Timing and Animation in Java

printer-friendly version