JetDog Game Framework

Marcia Lovas
Krish Pillai

Game Engine

The game engine handles the rendering of entities on screen, manages their positional updates, and keeps track of various events that occur during a game. A timer clock is the heartbeat of the game engine. Each time the clock ticks, a frame is generated and displayed on the screen. This cycle decides the frame rate of the game engine. The game engine repositions and renders entities once every frame and all processing for a specific frame must be completed before the next frame can be processed. As a result of positional changes on the screen, entities may collide with each other, drift out of the visible area of the screen, or run into impenetrable barriers. When they occur, such events are collected and communicated to the game implementation during each cycle of the game loop. Additionally, the keyboard must be polled periodically to check for user input as well. All these actions and event processing are carried out for every clock tick of the game.

The Game Loop

In the simplest sense, all games modeled by this framework consist of a cyclic invocation of a sequence of methods until some conditions are met. The game controller instance moves the game through its various states. A game is essentially composed of a sequence of frames that are generated at a stipulated frame rate (60 frames per second), like a flip book. To enable this strict periodicity, the framework maintains a game clock that fires repeatedly at the stipulated time interval. The clock invokes various methods in the proper sequence to move the game forward. Each frame displays various entities on the screen, to which the user responds through mouse and keyboard actions. The user inputs affect the attributes of each entity on the screen. The entities may move unobstructed within the viewable screen bounds or may go out of bounds, get blocked by barriers, or the entities may interact with each other through collisions. The way these events are resolved is implemented by the design of each specific game and is not part of the framework. The narrative of the game may even involve the inclusion or removal of entities from the next frame to be computed. As a result of their behavior and interaction with the user and other entities, entities may change their trajectories, exit the game, wrap-around, or disappear beyond screen bounds. Before every frame can be computed and displayed, the game must go through all entities known to it, relocate them on the screen based on the elapsed time, and check for collisions, blockages, or out-of-bound situations as shown in Figure 1.

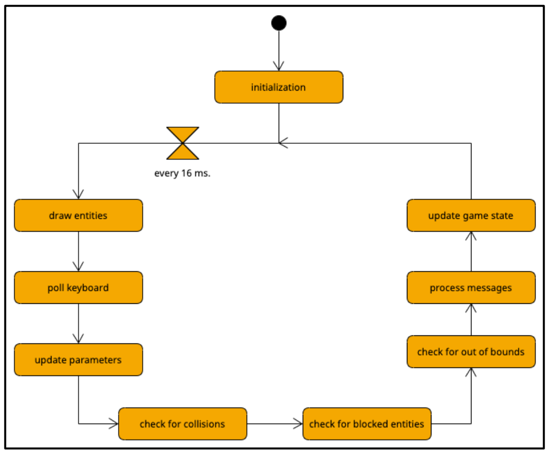


Figure 1. The game engine cycle (or game loop)

The invocation of various methods is the responsibility of the game clock which fires periodically at 16.667 milliseconds. The game clock invokes methods on the game controller and the entity model. Each event is collected as an array and supplied as an argument to the respective concrete method. For example, all entities are checked for out-of-bounds events during each cycle of the game loop. An out-of-bound event object containing the entity that stepped out of bounds and the edge that was crossed is created for each out-of-bound event. An array of the events is built and supplied as an argument to the onOutOfBounds(OutOfBounds[]) method, which the implementer of the game overrides for the specific game being built. The game is designed by overriding all the event-related abstract methods that the game engine invokes during every cycle. What makes one 2D game different from another is the way events are handled. The game engine, therefore, serves to abstract out the commonality across all 2D games.

Model-View-Controller

The implementation of the framework is based on the Model-View-Controller design pattern. Each entity is represented by a base class called the EntityModel which contains the fundamental attributes that are common across various entities. The controller functionality is provided by the game engine represented by the GameController abstract class. Entities can be of different types, and each of the supported types is represented by the following abstract classes shown in Figure 2:

- Scrolling scenery or background represented by the SceneryModel class
- Animated sprite-based entities represented by the SpriteModel class
- Textual data represented by the TextModel abstract class
- Obstacles or impervious barriers represented by the BarrierModel

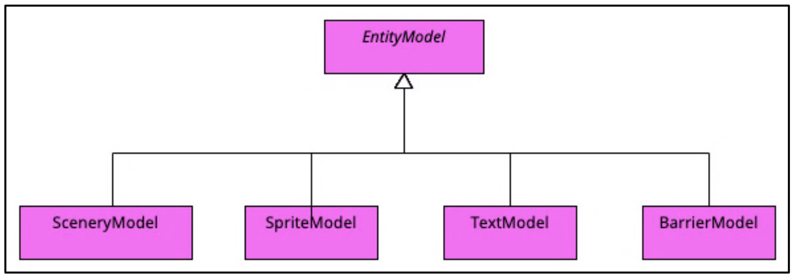


Figure 2. Model hierarchy

Each of these model types contains a reference to its view, which is rendered on screen when the controller invokes the draw() method on the entity. The draw() method is invoked on all entities during each cycle and the call is dynamically bound to the method implemented by the entity instance, which could be a sprite, text, or scenery. The view hierarchy shown in Figure 3 is supported by the base class EntityView, which is further extended into the TextView, SpriteView, and SceneryView classes. Each view type is associated with the corresponding model.

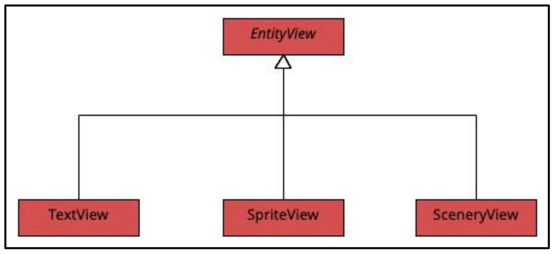


Figure 3. View hierarchy

The game controller, the entity model, and the associated view contribute to building the M-V-C pattern as shown in Figure 4. The controller interacts with the user, updates the view, and serves as a listener for all user interactions. The controller then updates the parameters of each entity based on the elapsed time and user input. This function is carried out for each iteration of the loop.

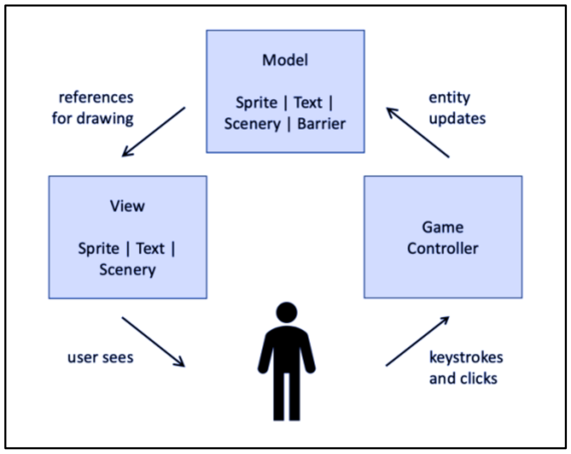


Figure 4. Implementation of the model-view-controller (M-V-C) pattern

A substantial amount of generic functionality is provided by the game controller, freeing up the developer to concentrate on the narrative of the game, which is implemented by providing the concrete implementation for various abstract methods which are invoked every cycle of the game loop.

Event-Handling

Various events can occur during a game cycle. Entities may go off-screen, they may collide with one another, or be obstructed by a barrier placed on the screen. Entities may even signal to the controller about something that is relevant to the narrative of the game. Classes that represent and manage these events are provided in the event handling framework, which is part of the game engine. Each game handles these events in a specific way that is pertinent to the narrative of the game. The game controller checks all entities for any of these events, aggregates them into arrays of events, and passes them as arguments to the corresponding overridden method. The developer can implement each of these callbacks in the concrete game controller by providing the appropriate handler for each event.

All of the event classes extend the base class GameEvent as shown in Figure 5. The events supported by the framework and the handler for each of those events are as follows:

- CollisionEvent
  - representing two colliding entities passed as an argument to the onCollisionEvent() method
- OutOfBoundEvent
  - representing an entity and a screen edge passed as an argument to the onOutOfBoundsEvent() method
- MessageEvent
  - representing an entity (sender) and a string message (description) passed as an argument to the onMessageEvent() method
- BlockedEvent
  - representing an entity and a barrier passed as an argument to the onBlockedEvent() method

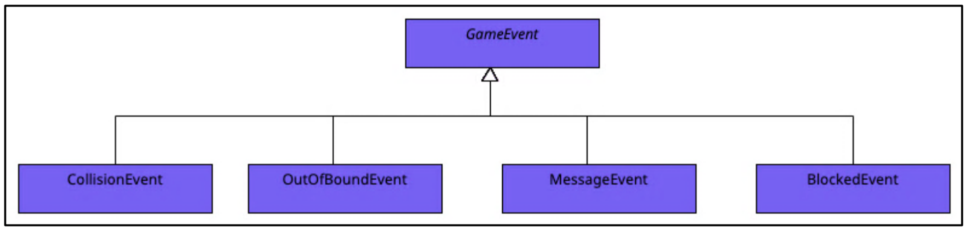


Figure 5. Event classes

Implementing a Game

Writing a game using the framework involves extending the game controller, and the required entity model types, setting the view for those entities, and writing appropriate event handlers for collisions, out-of-bounds, blockages, and messages. Sounds can be generated by invoking the play() method on a predefined set of enumerations provided by the SoundEffects enum. Textual information, such as a scoreboard can be created by providing a concrete class extension for the TextModel. When a concrete game controller is instantiated, the game clock is started up and the game starts to run. The game clock in turn sets off a sequence of message calls for every cycle. These triggers cause the rendering on screen and various event handlers to be invoked on the game controller instance. The relationship between the various classes is shown in the consolidated class diagram in Figure 6.

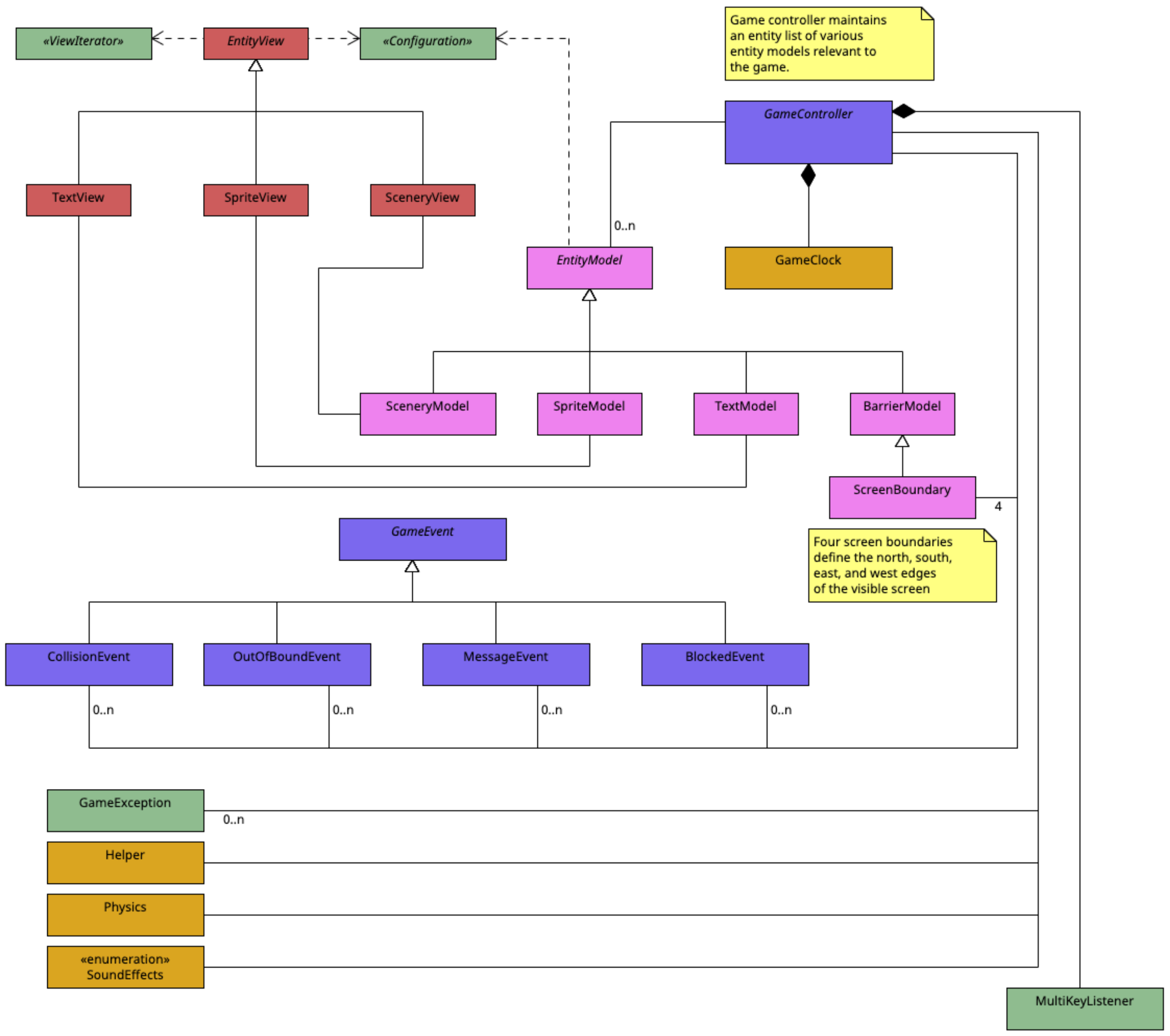


Figure 6. Class relationship diagram

Implementing a game using this framework involves writing derived classes that extend the GameController and the required subclasses of the EntityModel, which could be text, sprite, barrier, or scenery. The GameController provided by the framework is an abstract extension of the Swing JFrame. This gives unlimited access to the developer to all the functionality provided by the JFrame class. Keyboard and Mouse listeners, or any other features supported by the Swing toolkit, are available to the concrete game controller.

Example

Consider a simple 2D simulation of a pair of entities that appear on the screen. Let this class be called TwoBodySimulator as shown in Figure 7. The objective is to simulate an inelastic collision between the two bodies when they run into each other. Let us also assume that these entities shall bounce off the edge of the screen. The first step is to extend the GameController class. The enlistEntities() method is overridden to add the two entities. Since collisions and out-of-bounds are the two events that need to be defined for this simulator, the onCollisionsPolled() and the onOutOfBounds() methods are implemented. All other events are stubbed out, and not shown in the listing below for the sake of brevity. A simple class with three or four methods creates the basic simulation.

```
public class TwoBodySimulator extends GameController{

    @Override
    protected void enlistEntities() {
        // place red dot at (100,100) and blue dot at (200,200)
        SpriteModel a = createSpriteModel(Color.RED, 100, 100);
        SpriteModel b = createSpriteModel(Color.BLUE, 200, 200);
        // send them off in different directions
        a.setXVelocity(7);
        a.setYVelocity(8);
        b.setXVelocity(-5);
        b.setYVelocity(-7);
        a.setActive(true);
        b.setActive(true);
        // present them on the screen
        addEntity(a, b);
    }

    private SpriteModel createSpriteModel(Color color, int x, int y) {
        return new SpriteModel(x, y) {
            @Override
            protected void updateParameters(long elapsedTime) {
            // TODO Auto-generated method stub
            }

            @Override
            protected void setAppearance() {
                setView(color);
            }
        };
    }

    @Override
    protected void onCollisionsPolled(CollisionEvent[] events) {
        for (CollisionEvent event : events) {
            EntityModel a = event.getA();
            EntityModel b = event.getB();
            Physics.rebound(a, b);
        }
    }

    @Override
    protected void onOutOfBounds(OutOfBoundsEvent[] events) {
        for (OutOfBoundsEvent event : events) {
            EntityModel em = event.getEntity();
            ScreenBoundary sb = event.getBoundary();
            Physics.rebound(em, sb);
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(()->new TwoBodySimulator());
    }
}
```

Figure 7. Two-body simulator

More elaborate games are feasible by extending classes from the framework and implementing abstract methods on the controller and models. The periodic game clock invokes a series of abstract methods, the concrete versions of which are to be supplied by the user-implemented classes. The rules of the game can be coordinated by the updateGameState() method, which is invoked by the game clock at the end of each clock tick. The sequence in which the game clock invokes various methods on the game controller each time it is fired is shown in the detailed life-line diagram in Figures 8-10. The first cycle executed by the timer activity is unique. It starts off by invoking enlistEntities(), where entities that are to be seen on the screen can be added. When each entity is added, the corresponding setView() method is invoked by the controller. This initialization does not happen subsequently. Once the game loop starts off, the timer activity repeats the sequence of calls.

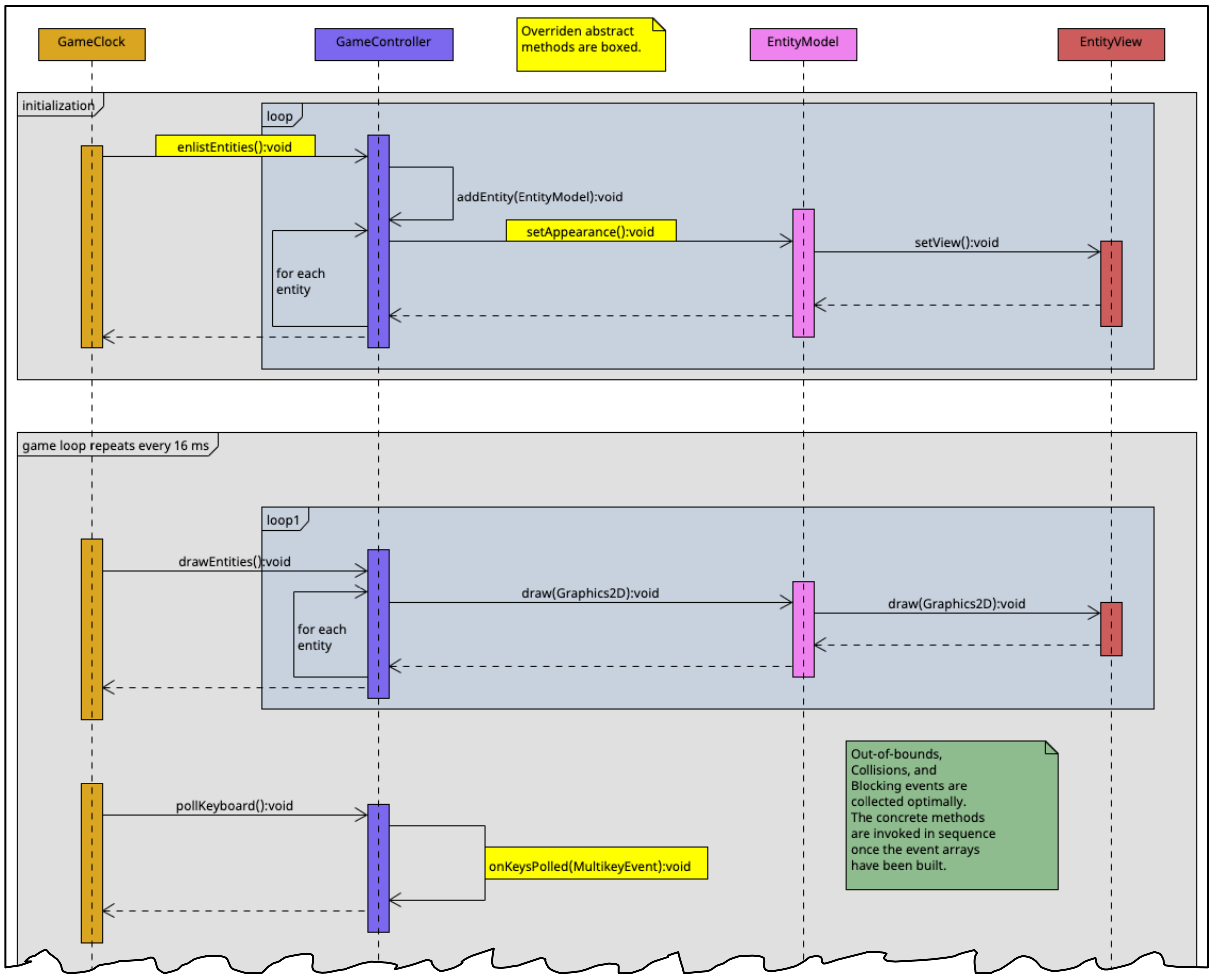

GameClock
GameController
Overriden abstract methods are boxed.
EntityModel
EntityView
initialization
loop
enlistEntities():void
addEntity(EntityModel):void
setAppearance():void
setView():void
for each entity
game loop repeats every 16 ms
loop1
drawEntities():void
draw(Graphics2D):void
draw(Graphics2D):void
for each entity
pollKeyboard():void
onKeysPolled(MultikeyEvent):void
Out-of-bounds, Collisions, and Blocking events are collected optimally. The concrete methods are invoked in sequence once the event arrays have been built.

Figure 8. The game loop sequence

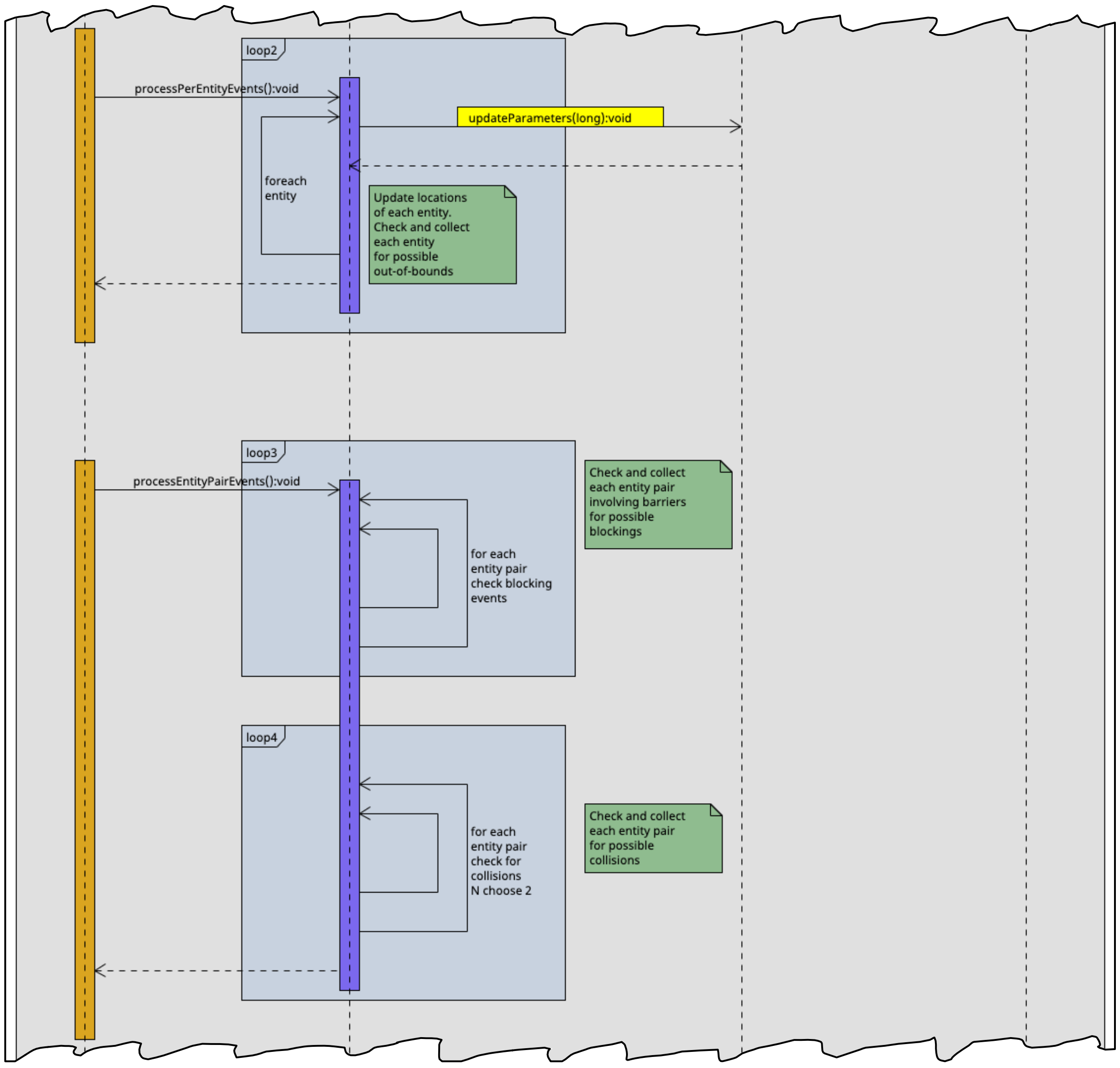


Figure 9. The game loop sequence continued

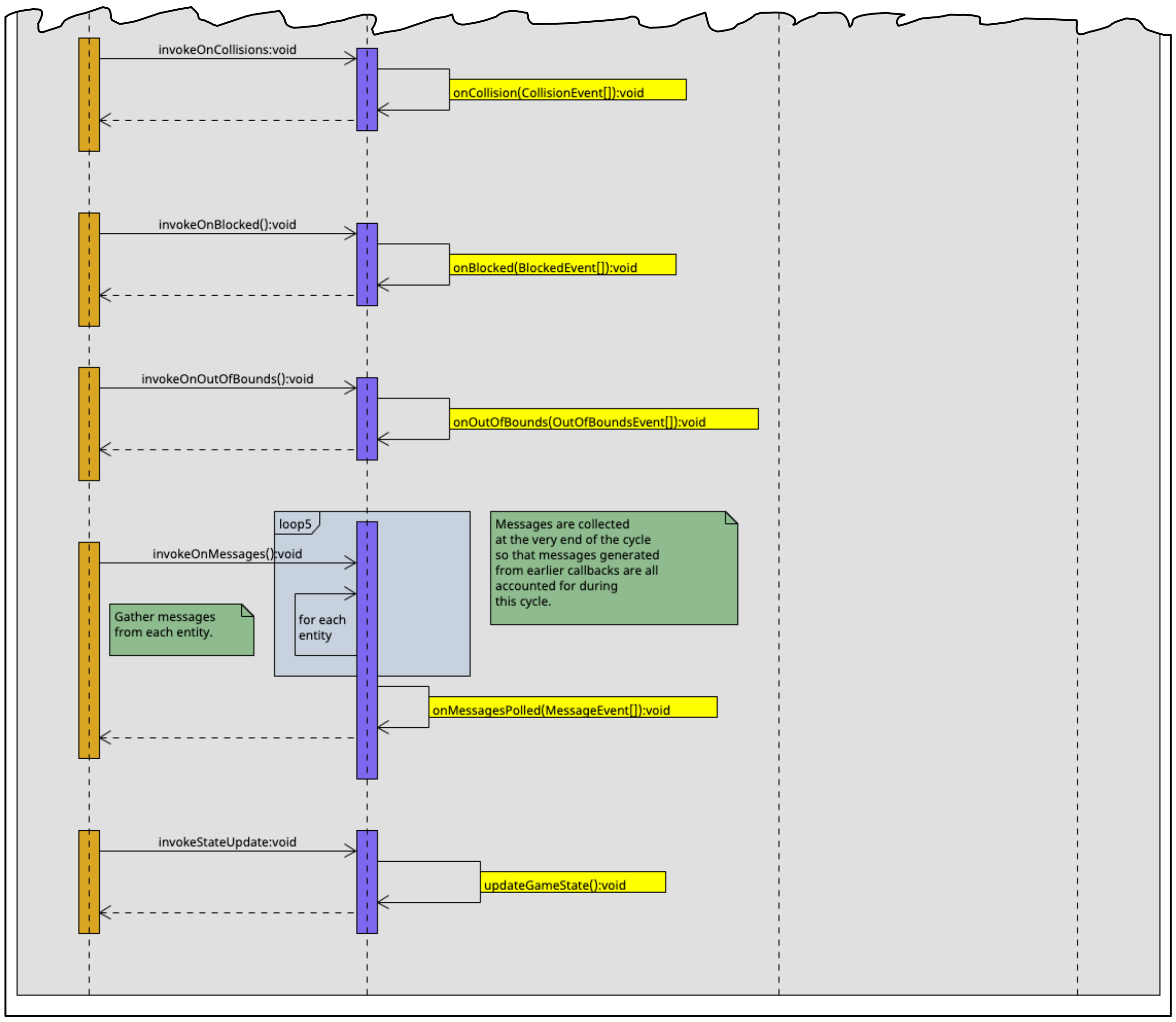

invokeOnCollisions:void
onCollision(CollisionEvent[]):void
invokeOnBlocked():void
onBlocked(BlockedEvent[]):void
invokeOnOutOfBounds():void
onOutOfBounds(OutOfBoundsEvent[]):void
loop5
invokeOnMessages():void
Messages are collected at the very end of the cycle so that messages generated from earlier callbacks are all accounted for during this cycle.
Gather messages from each entity.
for each entity
onMessagesPolled(MessageEvent[]):void
invokeStateUpdate:void
updateGameState():void

Figure 10. The game loop sequence continued

Learning Objectives

This course addresses the following objectives in the areas of game engine functionality, coding to a framework, object-oriented programming, and the creation of a full-fledged 2D video game.

Game Engine

GE1. Explain the purposes of the game loop activities including the timer, polling the keyboard, updating parameters, checking for collisions, checking for blocked entities, checking for out-of-bounds events, processing messages, and updating the game state.
GE2. Explain how the game engine uses the Model View Controller pattern.
GE3. Explain how the game engine simulates the physics of gravity.
GE4. Explain how the game engine employs callbacks.
GE5. Explain how the game engine draws a game's graphics.
GE6. Explain how the game engine performs animation.
GE7. Use the game engine's finite state machine classes to code a game.

Framework

FR1. Access the framework's API documentation.
FR2. Code the callback methods in the game loop including enlistEntities, setAppearance, onKeysPolled, updateParameters, onCollisionsPolled, on Blocked, onOutOfBounds, onMessagesPolled, and updateGameState.
FR3. Code a game controller and a model class.
FR4. Code animated sprites.
FR5. Code colliding sprites.
FR6. Code a seeking sprite.
FR7. Code sprites that rebound off each other.
FR8. Code sprites that bounce off screen edges.
FR9. Code sprites that bounce off barriers.
FR10. Code an incrementing scoreboard.
FR11. Code barriers.
FR12. Code scrolling scenery.
FR13. Code a ship firing a missile.
FR14. Code composite movement of a group of sprites.

Object-Oriented Programming

OO1. Demonstrate inheritance in a program.
OO2. Demonstrate the use of abstract classes in a program.
OO3. Demonstrate handling events in a program.
OO4. Demonstrate using an interface in a program.
OO5. Code an anonymous type in a program.
OO6. Code an inner type in a program.
OO7. Demonstrate the use of aggregation in a program.
OO8. Explain the Model View Controller pattern.
OO9. Define a finite state machine.
OO10. Code a state machine in a program.
OO11. Demonstrate the use of polymorphism in a program.

Video Game

VG1. Create a Space Aliens game similar to Space Invaders™.
VG2. Create an executable file of a game.

Learning Context

The website may be used in a traditional or flipped classroom mode, in which students have the opportunity to become familiar with the material outside of class and control their learning. In flipped mode, class time can be utilized for live problem-solving, targeted discussion, and interaction based on students' pre-work. The website should primarily be used on desktop computers where students will also code in an Integrated Development Environment (IDE) such as Eclipse or Visual Studio Code.

Schedule

The course is designed to include seven lessons shown in Figure 11, to fit the timeframe of a typical 14-week semester.

Lesson	Game topic(s)	Object-oriented topic(s)
1 Game loop	Game engine design Model View Controller (MVC) Game loop activities	Inheritance, abstract classes, MVC pattern
2 Vector dynamics	Using trigonometry and vector operations to navigate on screen	Event-handling, using an interface
3 Gravity and collisions	Using acceleration and inelastic collisions	Inheritance, abstract classes
4 Guided projectiles	Event-driven creation of entities that follow a defined trajectory	Anonymous inner types
5 Composite movements	More complex interactions	Aggregation
6 State machines	Using a state machine to model a multi-level game	State pattern
7 Executable app	Full Space Aliens game	Polymorphism

Figure 11. Lesson plan

Learning Tasks

Prerequisites

Learners should have the equivalent knowledge of two semesters of college Java programming.

Sequence

The game engine capabilities are introduced gradually as shown in Figure 12, in the context of building a complete Space Aliens game, shown in Figure 13, by the end of the final lesson.

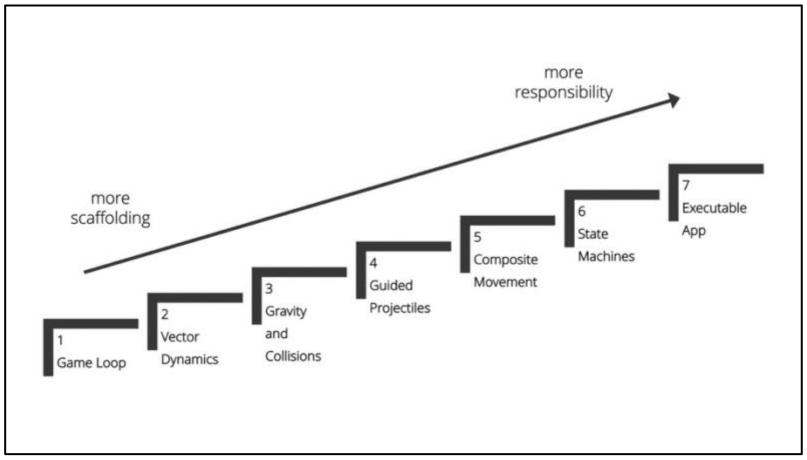


Figure 12. Learning tasks diagram

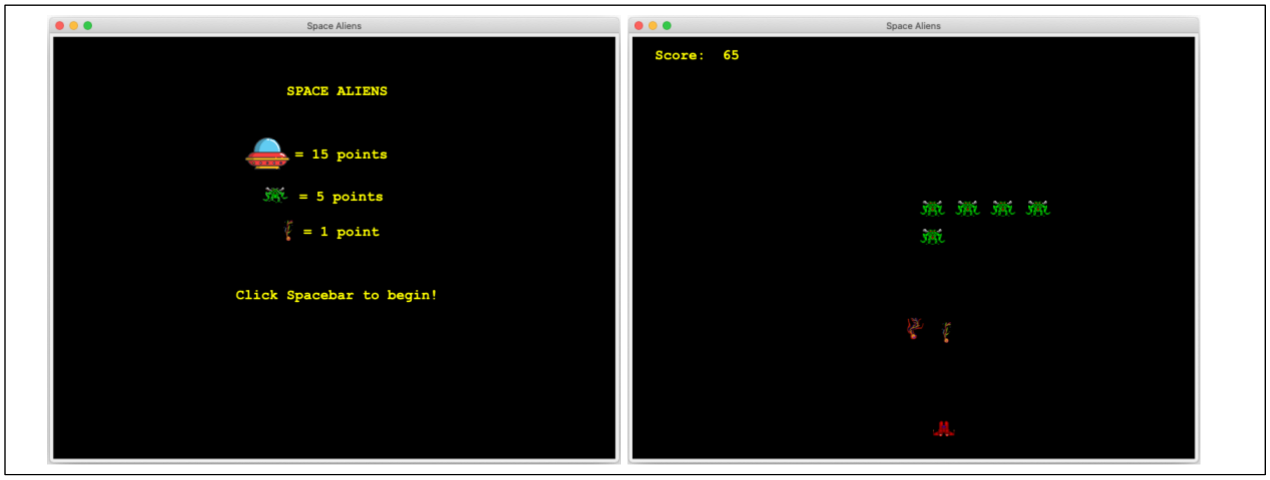


Figure 13. Student game implementation

Lesson Structure

Each lesson includes four key elements as shown in Figure 14, beginning with an interactive video, simulation, or H5P interaction. The interactive content is followed by a set of exercises for students to practice each skill incrementally and separately. Challenges then follow which combine the skills taught in the exercises into new applications or games for the student to create on their own. Each lesson concludes with links to additional optional resources, such as books, URLs, or videos, encouraging students to explore the lesson's material in greater depth.

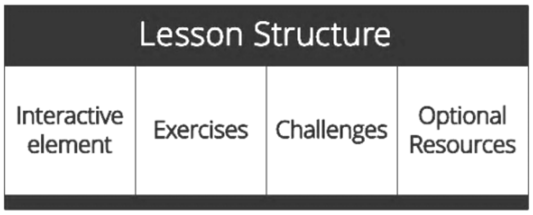


Figure 14. Lesson structure

Lessons

Computer Setup

You may use any IDE. We'll use a lightweight editor, VS Code. If you're not using VS Code, perform similar steps using your IDE.

1. **Install Java**
Select the latest Java Development Kit (JDK) for your operating system from Oracle: Java Downloads.
Follow the instructions to install it on your machine.

2. **Install an IDE**
Download the package for your machine: Visual Studio Code Download.
Follow the instructions to install it.

3. **Configure your IDE for Java**

Install some VS Code extensions to make it easier to develop Java.
In VS Code, select the 'Extensions' icon on the left, and search for and install the following extensions:
  MS Extension Pack for Java
  UMLet (for drawing UML diagrams)
In VS Code, open the command palette (⇧⌘P) or (Ctrl-Shift-P) and type 'java tips' to get started.
Visit Java in Visual Studio Code for more help.

4. **Git the game engine and exercise solutions**

In a browser, visit https://github.com/krish-pillai/jetdog-distro.git.
Find the latest release of the game engine framework and download the .zip file to your machine.
Unzip the file by double clicking it.
Open VS Code, select 'Open', and find and select the jetdog-distro-x.x.x folder that you unzipped.
Select ‘File’, ‘New File’ (or) to create a new java file and name it ‘myGame.java’.

5. **Configure the classpath**
Open the command palette (⇧⌘P) or (Ctrl-Shift-P) and type 'configure classpath'.
Let VS Code know where to find your source and build files.
You may also need to configure the Java runtime JDK.

6. **Reference the game engine library**
In VS Code, under Java Projects, right click on Referenced Libraries.
Find the jetdog-lib-x.x.x.jar file and select it.
(Or click the '+' next to Referenced Libraries and select the jetdog-lib-x.x.x.jar file.)
Open your myGame.java file, add a main() method:

```
public class myGame {

      Public static void main(String[] args) {
          System.out.println(“My First JetDog Program.”);
      }
}
```

and select 'Run'. Voila!

7. **Save your workspace**
Select 'File', 'Save Workspace As', and give your workspace a name.

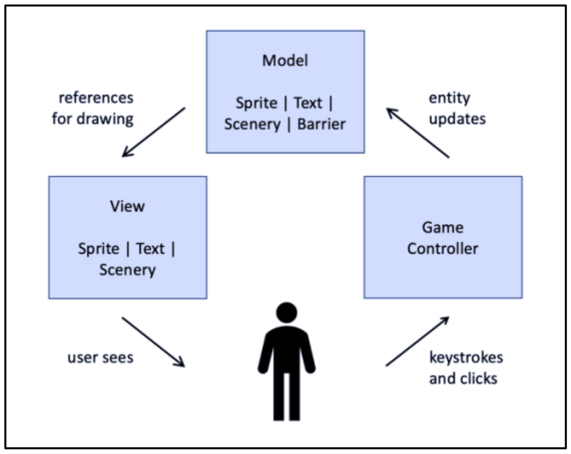
1 Game Loop

Objectives: GE1, GE2, GE4, FR1, FR2, FR3, FR4, FR5, OO1, OO2, OO3, OO4, OO8

- Explain the purpose of the game loop's activities.
- Create a game controller and a model class.
- Code the callbacks: enlistEntities, setAppearance, onCollisionsPolled, and onOutOfBounds.
- Demonstrate the use of inheritance and abstract classes.

Lesson

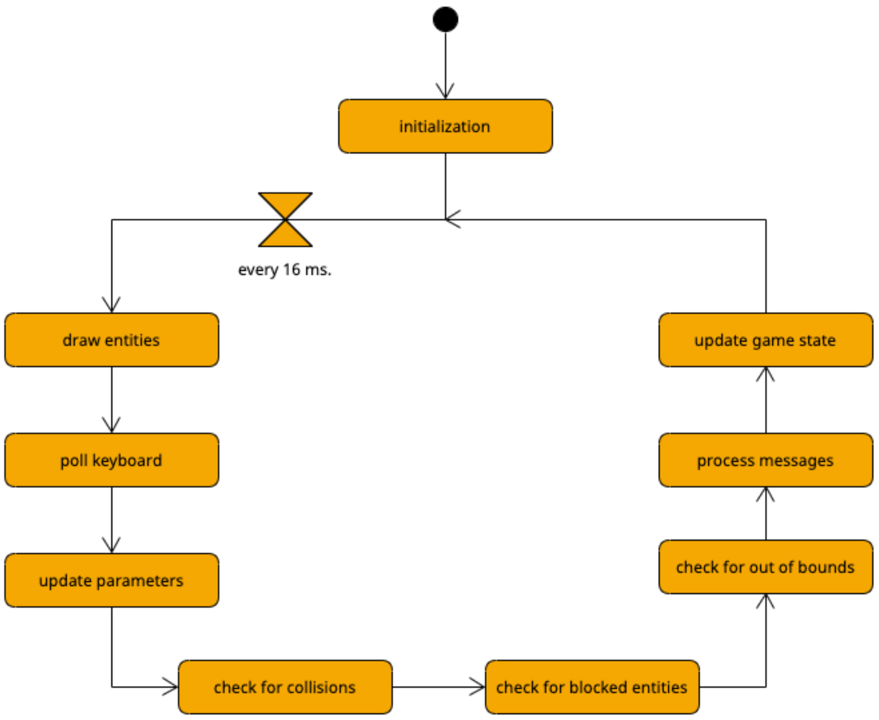
The game framework is organized using the model-view-controller design pattern. The user interacts with the controller using the keyboard and mouse. The controller contains a clock that executes the game loop logic every 16 ms. The controller keeps the Model objects updated with information associated with the state of sprites, text, and scenery. The model contains a reference to the View for drawing each of these entities. The View is what the user sees on the screen.



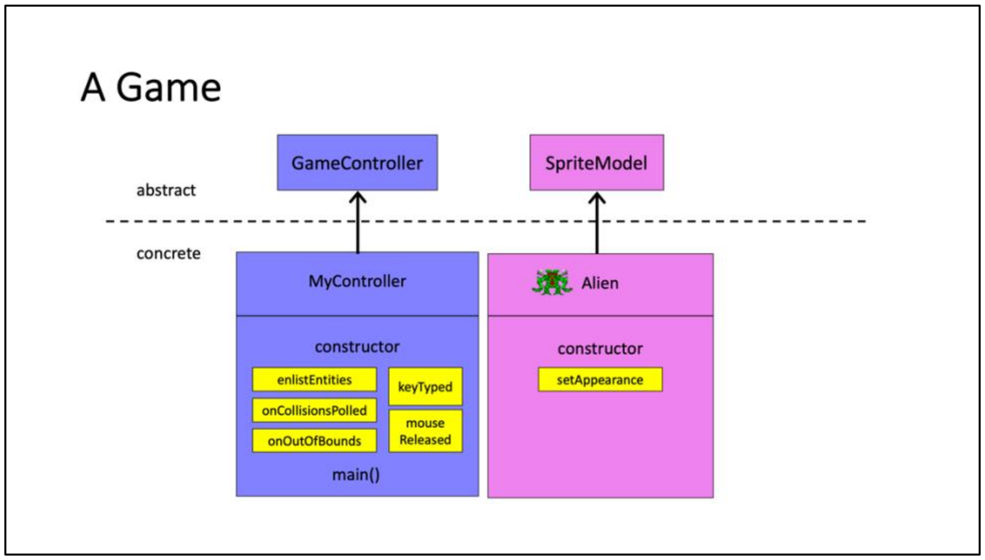
Let's take a closer look at the game loop. When the game begins, some initialization takes place, then the game clock ticks every 16 ms. During that time several tasks take place:

- Draw entities
- Poll the keyboard for keys pressed
- Process events
- Check for collisions
- Checked for blocked entities
- Check for entities out of bounds
- Check messages
- And update the game state such as updating the score

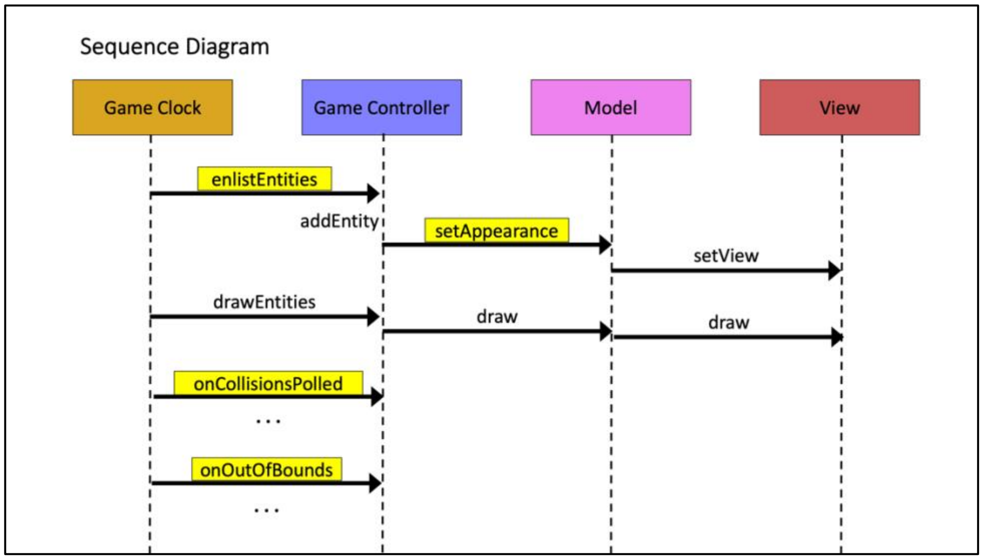
This happens every 16 ms continuously to maintain a typical video game refresh rate of 60 frames per second. This is all taken care of by the framework, so you don't have to worry about implementing it. However, there are some callback methods you are responsible for defining. Some of these methods are in the controller. Others are in the models.



Let’s take a closer look at how you would create your own game. The framework provides these abstract classes for you to extend. First extend the game controller to make your own concrete game. You’ll need to write a constructor as well as a main method. Next extend the model, in this case a Sprite Model, to create a concrete Alien class, or whatever creature you’d like. Then add the callback methods for each. For example, to tell the game what to do when the spacebar is pressed, you will write the onKeysPolled method in the controller.



It may help to look at a sequence diagram to see what happens when the game runs. The game clock calls your enlistEntities method in which you add all the sprites and other entities in your game. addEntity calls your setAppearance method in your model and that sets up the view. Next, the clock draws the entities, and the View handles drawing them. Then the rest of the game loop is executed, including those callback methods in your controller such as onCollision and onOutOfBounds, which you're responsible for implementing.



Exercises

1A_SimpleSprite	Make a simple sprite. Start by extending GameController and SpriteModel like this:
	
	public class SimpleSpriteExercise extends GameController
	
	Create constructors for both classes and generate the abstract methods for both. Next, write the main method in your controller class:
	
	`public static void main(String[] args) {`
	`    SwingUtilities.invokeLater(()->new`
	`SimpleSpriteExercise());`
	`}`
	
	Now you'll need to use the methods shown in the sequence diagram:
	
	Here's how it might look:
	
	`@Override`
	`protected void enlistEntities() {`
	`    alien = new Alien(400, 300);`
	`    addEntity(alien);`
	`}`
	
	`@Override`
	`protected void setAppearance() {`
	`    setView("alien-3.png", 1.0f, 3);`
	`}`
	
	For help, read about each class and method in the Javadoc.

1B_AnimatedSprite	Place a sprite on the screen and make it disappear when the spacebar is pressed. Use the methods in the sequence diagram: For help, try using MultiKeyEvent and KeyMap. Also read more in the Javadoc. Make it so that pressing the Escape key exits the game. Use onKeysPolled to check for the Escape key press. Once you have the Escape key working, you can add it to all your exercises.
1C_BouncingSprite	Place a sprite on the screen, set its x and y velocities, and start it moving. When it reaches the edge of the window, make it bounce and change direction. Use the sequence diagram: For help, try using event.getEdge() and ScreenBoundary. Read more in the Javadoc (link).
1D_CollidingSprites	Place two randomly-moving sprites on the screen and make them bounce off the walls. When they collide with each other, reverse their velocities to make them bounce away from each other. Activate a bounce sound when they collide: SoundEffects.BOUNCE.play(); For help, try using event.getA() and event.getB(). Use util.SoundEffects. Read more in the Javadoc (link).

More About onKeysPolled

Here are some important details on the abstract method onKeysPolled(int[]).

The onKeysPolled method is invoked for every clock tick. If a key is depressed for more than the cycle time of the game loop, it will be detected as a multi-key press. onKeysPolled is the right way to interact with a game since it is highly responsive to key strokes. It should not be used for single-shot operations. For example, moving an entity on screen based on keyboard inputs should be tied to the onKeysPolled() method. On the other hand, if you want to add an entity to the screen when a key is pressed, this mechanism would be unsuitable. Since human response times are slow compared to the cycle time of the game engine, a single keystroke will last over several game engine cycles and will result in multiple entities being added to the screen.

To avoid this, single-shot operations should make use of the standard Swing KeyListener. The onKeyTyped method of the KeyListener should be ideally used for single-shot operations. onKeysPolled is invoked every cycle of the game loop. When keys are depressed, the array argument passed to this method will contain the Swing defined key codes of all the active keys. The method gets invoked even if no keys are pressed. If there are no active keys, the argument to the method is an integer array with a single entry in it, which will be keycode zero. This proves a way for the game to take some action when no keys are active. For example, an onKeysPolled implementation that moves a ball to the WASD keys, but make it stop as soon as all keys are released, is shown below:

```
@Override
protected void onKeysPolled(int[] keyCodes) {
      for (int code : keyCodes) {
            switch(code) {
                  case KeyEvent.VK_W:
                        ball.setYVelocity(-speed);
                        break;
                  case KeyEvent.VK_A:
```

```
                                ball.setXVelocity(-speed);
                        break;
                        case KeyEvent.VK_S:
                                ball.setYVelocity(speed);
                        break;
                        case KeyEvent.VK_D:
                                ball.setXVelocity(speed);
                        break;
                        default:
                                ball.setXVelocity(0);
                                ball.setYVelocity(0);
                }
        }
}
```

Challenges

Code (1E_SpriteLab): Make two randomly-moving sprites that disappear with a sound when they collide.

2 Simple Vector Dynamics

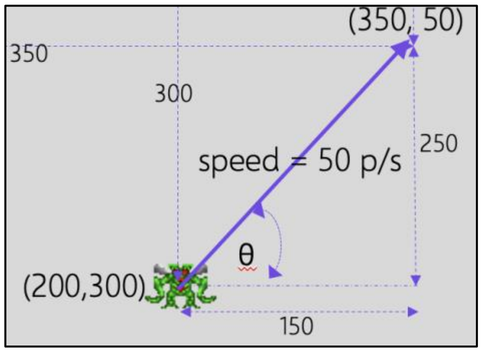
Objectives: GE4, FR1, FR2, FR3, OO1, OO2, OO3, OO4, FR6

- Explain how to use vector calculations to move a sprite.
- Code sprite behavior in the updateParameters callback.
- Use notifyController and the onMessagesPolled callback to specify game behavior.
- Demonstrate the use of event-handling, callbacks, and interfaces.

Lesson

How can we make a sprite move to a particular location? We'll use vector calculations for this.

Let's make this alien move directly to the cookie. We're using the Java 2D coordinate space. We need to tell the sprite both the magnitude and the direction of where to go. If we tell the sprite to go a certain distance (a scalar), it still needs to know which direction. If we tell it the direction (another scalar), it needs to know how far to go at that angle. While scalars, like distance or direction, have magnitude only, vectors have both magnitude *and* direction, so we'll use vectors to solve this problem.



First, let's find which direction (or angle) the alien needs to move. If the alien moves directly toward the cookie, we can create a right triangle and use trigonometry to break this distance into its vector components, one in the x direction and one in the y direction. Given the coordinates for the alien and the cookie, we can calculate the distance in the y direction to be y2-y1, or in this case, 250. In the x direction, x2-x1 is a distance of 150. Using trig, we can solve for the angle theta. The tangent of theta should be the opposite side over the adjacent side, or 250/150. Since we want the angle, we take the arc tangent of (250/150) = to get 59 degrees.

Now let’s find the velocities in the x and y directions. If we know the velocity of the alien toward the cookie, we can again use trig to calculate the component velocity vectors. xVelocity = V * cos(theta), and the yVelocity = V * sin(theta). Since our alien has a velocity of 50 pixels/second, Vx = 50 * cos(59) = 26 p/s, and Vy = 50 * sin(59) = 43 p/s.

Every time the game loop is executed and updateParameters is called on our Alien, we can use these vector calculations to program and set the new x and y velocities for the sprite. Since every 16ms, the Alien’s x and y location has changed, we need to recalculate the x and y velocities every time updateParameters is called.

There are two more things you will need to take into account. First, many of the Java Math methods specify parameters and return values in radians instead of degrees. You will need to make use of the fact that a circle of 360 degrees= 2*Pi radians. 90 degrees = Pi/2 radians.

To convert between the systems, remember that degrees = radians * 180/Pi, and radians = degrees * Pi/180. Note that 180/Pi is equal to 1 and is only used to convert. If you need to convert, you can use Math.PI.

For our example, a theta of 59 degrees = 59*(Pi/180) = 1.03 radians. Using the Java methods in java.lang.Math, Math.atan(250/150) = 1.0304 radians.

Second, we have to take into account the way software handles tangent and arc tangent functions. Since tangent is not a one-to-one function, arctan (Math.atan) has a range of -90 to 90, or -Pi/2 to Pi/2. Most software returns a value in this range. When we pass the x and y values into Math.atan(), the software loses track of the quadrant information, or the signs of the change in X and the change in Y. To adjust for this, use the arctan2 function (Math.atan2). This method takes y and x parameters and considers their signs, or the quadrant.

To understand more about the trigonometry involved, visit the links below.

(162) The 4 Quadrant Inverse Tangent (atan2) and Other Inverse Trigonometric Functions - YouTube

Sine, Cosine and Tangent in Four Quadrants (mathsisfun.com)

Exercises

2A_LayeredClouds	Make a scene with three clouds and a sun in overlapping layers. For help, use setScale() in your enlistEntities() method to make the sun big. Use setBackground() to change the background color to sky blue. Use the sun sprite, “sun-4.png”.
2B_PhantomBalloon	Make a scene in which a mouse listener registers a user’s mouse click, making a balloon appear for a short time only. For help, add a Mouse Listener to your controller and read about setTimeToLive.

2C_SeekingBalloon	Make a randomly-placed balloon move to the center of the screen using updateParameters and vector dynamics. For help,use GAMELOGGER.Info to write debug messages to the console, use the Helper utility. And don't forget to setSpeed to non-zero!
2D_PoppedBalloon	Make the balloon pop when it reaches the center of the screen. Use an explosion sprite for the pop. Add a sound effect (COLLIDE) when it pops. For help, use setGhost(true) on the pop to prevent a collision event being generated when the explosion sprite and the balloon sprite intersect. .

Challenges

Code (2E_CookieGame): Create a game where a blank display has an alien that pointlessly roams the screen searching for food. You should be able to drop a cookie on the screen by clicking on it. Once a cookie has been placed on the screen, the alien should make a beeline for the cookie and consume it. It should rest for fifteen seconds, content with its meal. After the alien has rested long enough, it should start moving around the screen again, searching for food until the feeding cycle repeats.

Draw: A sequence diagram of the cookie game. Include lifelines for your Controller, Alien, and Cookie. Draw the method calls for a use case sequence. Use pencil and paper or UMLet (link).

3 Gravity and Collisions

Objectives: GE3, GE4, GE6, FR1, FR2, FR3, FR4, FR5, FR7, FR8, FR9, FR10, FR11, OO1, OO2, OO3

- Explain how the game engine simulates gravity.
- Use the Physics methods: setCoefficientOfRestitution, setRotation, and rebound.
- Make sprites rebound off each other and screen edges.
- Handle scoring in the updateGameState callback.
- Make sprites bounce off barriers using the onBlocked callback.

Lesson

How can we make objects behave like they do in the physical world? We can use the Physics class provided by the game framework. We can make objects exhibit the effects of gravity, spin, and bounce off walls and each other with varying degrees of elasticity.

Let's say we want a soccer ball to fall and bounce on the floor and walls in a realistic way. We can create the sprite and give it an x velocity, then set the y Acceleration to a positive value. This downward acceleration simulates gravity. The x velocity will make it arc a bit the right or left as it falls.

To put some spin on the ball to make it look natural, we can call setRotationAngle every time updateParameters is called in the game loop.

To handle collisions with the east and west walls, floor, and ceiling of the game, we can use the onOutOfBounds method. We can use the ScreenEdge enumeration in the Configuration interface to check if the collision involves the North, South, East or West wall. Use the getEdge method on collision event entities for this, then modify the entity velocities depending on the edge involved and the desired behavior.

To make two objects bounce off each other, in onCollisionsPolled, we can call the Physics.rebound method on the two collision event objects. The rebound method changes the two objects' x and y velocities depending on each entity's x and y velocities at the time of the collision, their masses, and each entity's coefficient of restitution. You may set each object's properties using the setxVelocity, setyVelocity, setMass and setCoefficientOfRestitution methods on the entity.

Setting the frames per second to 100 is also recommended. To catch fast-moving events, you will likely need to decrease the velocity of entities or increase the frame rate.

How does the framework do this?

It makes use of restitution and the law of conservation of momentum.

Restitution is how much energy is left after a collision and affects the "bounciness" of objects. The coefficient (or COR) is the ratio between the starting and ending velocity after a bounce for an object. If the COR is 0, all energy is absorbed on impact. If the COR is 1, there is perfect elasticity (very bouncy). A COR greater than 1 would add extra fictional bounce after a collision. When two objects with different CORs collide, the lowest restitution is the one that is used. For example, if a soccer ball falls and hits a sandbag, they both use the restitution of the sand bag.

For collisions, the conservation of momentum means that during a collision, the total momentum of the system remains constant. We have:

ma * ua + mb * ub = ma * va + mb * vb

We know the COR = | vb - va | / | ua - ub |

From these two, we can derive the following equations to calculate the new velocities:

va = ma * ua + mb * ub + mb * CORa * (ub - ua) / (ma + mb)
vb = mb * ub + ma * ua + ma * CORb * (ua - ub) / (ma + mb)

These are the equations the framework uses to make the rebound method work.

Reference links:
Coefficient Of Restitution: Definition, Explanation And Formula » Science ABC
2D Collision Simulator (danimator.github.io)
Collision Lab - Collisions | Conservation of Energy | Conservation of Momentum - PhET Interactive Simulations (colorado.edu)
Coefficient of restitution - Wikipedia

Exercises

3A_SoccerBall	Make a soccer ball appear on the screen and use the left, right, up and down arrow keys (and WASD) to move it around. For help, use onKeysPolled.
3B_GravityBall	Make a soccer ball appear and be acted upon by gravity (a downward acceleration). It should bounce off the walls and floor as a ball would. Use a ScreenBoundary.

	For help, set the coefficient of restitution in setAppearance(), code onOutOfBounds, and set the rotation angle in updateParameters.
3C_ReboundingBalls	Make a single soccer ball appear each time the mouse is clicked, make them behave as in 3B, and make them rebound from each other using Physics when they collide. Make the game controller write a message to the console each time the ball hits the South wall. At the bottom of the screen, for the balls with a velocity of 0, with acceleration downward, and appearing almost at rest, turn off acceleration and call .dispose(). Write a message to the console "The ball has stopped." If you want, you can show the Collision Bounds width and height (hit box width and height) for each ball using showCollisionBounds. Later, it may be useful to adjust this. For help, use notifyController and onMessagesPolled.
3D_Scoreboard	Add a scoreboard to your game, initialized to "0". Each time a ball hits the south wall, increment the score. When the score reaches 10, change the score display to "GAME OVER". For help, use updateGameState().
3E_Barrier	Put a sloping barrier in the window and call rebound() in onBlocked() when a ball hits the barrier. Extra: If you want, go back and use a BarrierModel to fix the "stopped" balls in 3C.
3F_FlippedAirplane	Airplane that moves left and right to arrow keys, flipping itself to face the direction it's going. Spacebar makes it go up.

Challenges

Code (3G_JumpingSprite): Create a game where a green man (greenman-4.png) can move left and right with the arrow keys. Make the man able to jump using the spacebar giving him the acceleration of gravity. Create a bee that appears at the right edge of the screen, moves to the left, and then wraps around to appear at the right again. Use a SceneryModel (ufoscene-1.png) to create a static background for the game. Each time the bee crosses the screen and the man successfully avoids getting stung by jumping over it, a point is scored. Make a scoreboard that shows the total points. When 3 bee stings have been avoided, display "Game Over".

Tip 1: With most games, you'll have to tweak the velocities, acceleration, and sprite hit boxes. Use showCollisionBoudns() and setCollisionBounds() to experiment.

Tip 2: For this challenge, you may run into a situation where Barrier Model will come in handy to solve your problem.

Draw: A UML class diagram for one of your games (JumpingSprite or BeachPail). Include classes for GameController, SpriteModel, TextModel, your controller, and scoreboard. Depending on the game, include your classes for the jumping man and bee, or the beach pail and ball. Use pencil and paper or a software tool like UMLet.

Code (3H_BeachPailGame): Make a pail move left and right across the bottom of the screen using keystrokes. The game should begin by automatically releasing a single beach ball from the top of the screen. Move the pail left and right to collect balls. When the ball comes in contact with the pail, the ball should disappear, the score should be incremented, and another ball should be released. When 10 balls have been “collected,” display “Game Over”.

Tips: If you have objects created during the game loop, don’t do it in enlistEntities. enlistEntities is done once at the beginning for objects that are present at the beginning of the game.

For all exercises and challenges, try incorporating a few features of your own. Get creative and make it unique.

4 Guided Projectiles

Objectives: GE1, GE4, FR1, FR2, FR3, FR11, FR12, FR13, OO1, OO2, OO5, OO6

- Code realistic missile behavior in the onKeysPolled callback.
- Code an inner class.
- Code an anonymous class.

Lesson

How can we make a ship fire a missile using the game framework? We start by extending the SpriteModel class to create a Missile class. We can trigger the missile to appear using a keystroke like the spacebar for example. Then we’ll code the response in the onKeysPolled method:

We’ll create a Missile instance using the new() keyword, and pass it x and y location parameters related to the ship’s current location. You’ll have to use getxLocation and getyLocation and decide where the missile should appear in relation to these coordinates. Call addEntity on the missile and set it active (setActive) with a velocity (either setxVelocity or setyVelocity or both).

If you want to make the ship wait a brief moment between firing (This will make the game more challenging and realistic), you’ll need to add a timing system. You can use the current System time and a desired amount of delay between missiles, and then keep track of the last instant the missile was fired. In onKeysPolled, when the spacebar is pressed, get the System.currentTimeMillis(), subtract the last firing instant, and check if that difference is longer than your desired firing delay. If so, create the missile! Pew, pew!

Exercises

4A_ScrollingScenery	Make a screen with a scrolling scenery image. For help, use setxVelocity().
4B_MovingShip	Make a ship move left and right at the bottom of the screen in response to keystrokes.
4C_FiringShip	Make the ship move left and right and fire missiles.
4D_AnonymousAliens	Add to what you created in 4C. Make an alien move from right to left across the screen and wrap around. Make a second alien that moves right to left and when it reaches the west side of the screen, it disappears, generating a new

	alien on the east side at a random Y location, again moving left. Use anonymous classes.The Alien should use the anonymous inner type SpriteModel, since it uses a Sprite to represent its view.
4E_Barrier	Simple barrier exercise. On the bottom half of the screen make a barrier down the center and an alien that starts right to left. When the alien hits the barrier for the first time, it should crawl up and around and continue to the left. Design Decision These design decisions can impact reusability, extensibility, maintenance, which all affect cost and development time. What are the advantages and disadvantages of each design approach? Note that this type of barrier can be used in numerous ways including being placed horizontally on background scenery to make it look like someone's walking on a ledge.

Challenges

Code (4F_UFOMissileGame): Make a game that displays a UFO at the left side of the screen. The spaceship should respond to up and down arrow keystrokes. Create an alien or aliens that move right to left. When the alien reaches the left edge, a new alien should appear at the right edge at a random Y location. The ship should fire missiles eastward when the spacebar is pressed. When a missile collides with an alien, make an explosion appear briefly and make the alien disappear with a sound. Make both the alien and the ship disappear when the alien collides with the ship. Use the ufo-7.png and dot-4.png image files to implement your sprites. Use a static scenery background with fortresses to hide behind. Add Barriers. When the alien bumps into a barrier, it should crawl along it and around it.

Note: To get the pixel locations to draw the barriers in the exact locations shown in the scenery, set a mouse listener in your game controller, and make it print to the console the coordinates on mouse clicked.

Draw: A UML object diagram of the missile-firing UFO. An object diagram depicts a snapshot in time, so not everything in your code will be shown in the diagram. You may want to include your ship, an alien or aliens, the explosion, and/or the missile object. Use pencil and paper or UMLet (link).

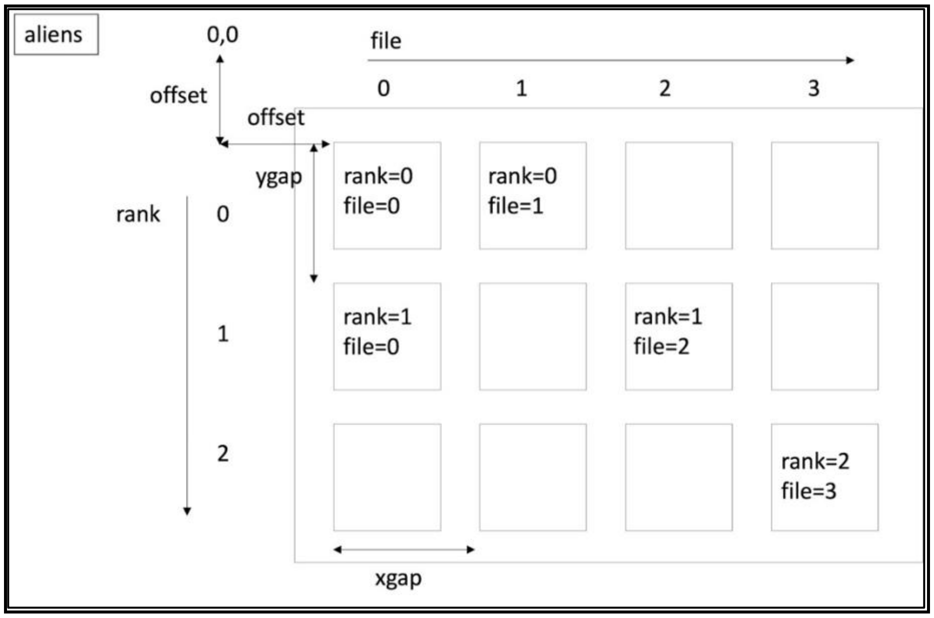
5 Composite Movement

Objectives: GE4, FR1, FR2, FR3, FR13, FR14, OO1, OO2, OO3, OO7, VG1

- Code a cluster of sprites that behave as a group.
- Demonstrate the use of aggregation.

Lesson

What if we want to have a group of sprites move as a single unit? This is a composite movement. We can't just extend a Sprite class, because that would give us only single instances of the sprite. We need a new class for the "army" of sprites, let's call it AlienArmy. This new class contains a data structure that groups sprites into a matrix. A two-dimensional array is ideal for this purpose. We can create an array and initialize it to contain new instances of a particular kind of sprite, an Alien in our example.



We can make the swarm move as a group when one of the army bumps into a wall. If an individual sprite in the army has an outOfBounds event, it can notify the game controller, and the onMessagesPolled can make the army move in a certain way. Perhaps an alien army will step forward and reverse direction. Perhaps a group of asteroids will bounce off the wall. To accomplish this composite movement as a group, create a method in your AlienArmy class and iterate through the matrix using nested for loops, commanding each sprite to behave as desired.

Exercises

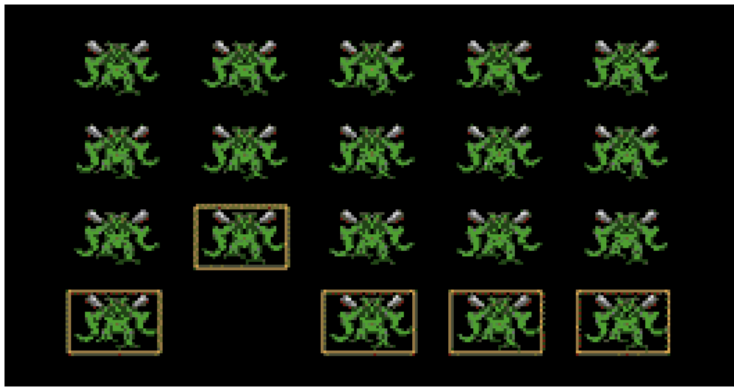
5A_RotatingShip	Make a ship in the center of the screen rotate when the left and right arrow keys are pressed.
5B_FiringRotating	Make the ship fire a missile in the direction it is facing. You'll have to use vector dynamics for this.
5C_SpaceRockCluster	Make two separate space rocks that appear and move in a random direction wrapping around the screen. Make a cluster of four space rocks that bounces as a group off the edge walls. ShowCollisionBounds on the rock in the cluster that is closest to the ship at all times.
5D_RockExplosions	When a missile hits a lone space rock or a space rock in the cluster, make an explosion appear and the single rock disappear. When the ship gets hit by a space rock, make the ship and rock both explode, then disappear. During this entire game, showCollisionBounds on the rock in the cluster that is closest to the ship. (same as you did in Exercise 5C)

Challenges

Code (5E_SpaceAliensTest): Code a Space Aliens game from the UML class diagram provided. The game should have the following features:

- A spaceship that can move left/right along the baseline using the A/D and arrow keys. It should fire missiles when the spacebar is pressed.

- An army of aliens that march towards the baseline (bottom of the screen). The army of aliens should march horizontally, stepping forward when it gets to the edge of the screen. When an alien is hit by a spaceship missile, it should explode and disappear.
- ShowCollisionBounds on the "frontline" of aliens. The frontline is always the highest ranking in each file. Note that a "front line" is not necessarily a straight line.



Test this out by trying to shoot an alien behind the frontline. Make sure the frontline remains unchanged.

Code (5F_SpaceAliens): Add the following features to your 5E_SpaceAliensTest:

- The frontline aliens should fire 'fireballs' back at the spaceship. Only the front line aliens should fire. In addition, an alien should not fire at another alien.
- The spaceship should explode if hit by a fireball thrown by an alien. It should also explode if bumped by an alien.

6 State Machines

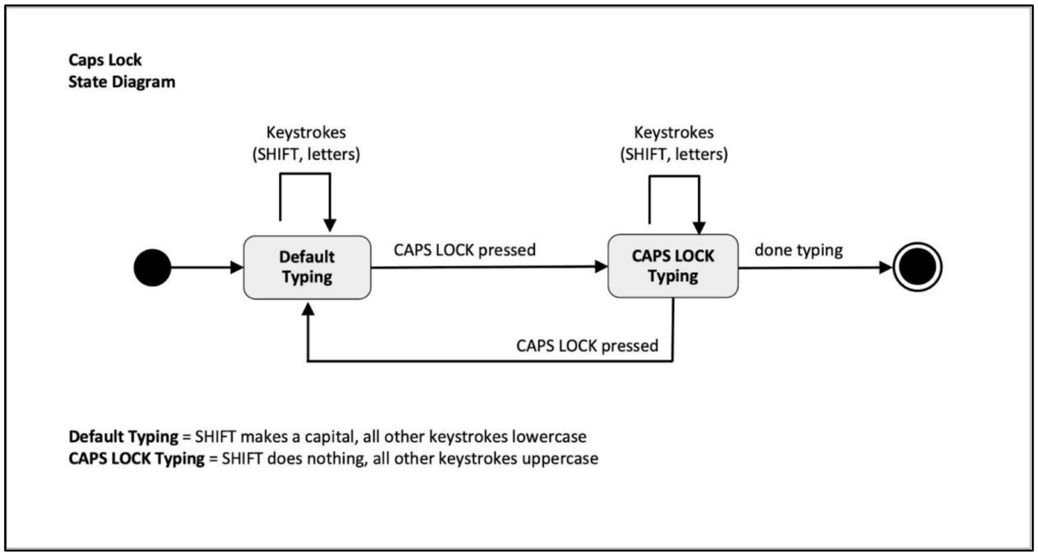
Objectives: GE5, GE7, FR1, FR2, FR3, FR4, FR5, FR7, FR8, FR11, FR13, FR14, OO1, OO2, OO3, OO7, OO9, OO10, OO11

- Code a game using a state machine.
- Demonstrate the use of the state pattern.

Lesson

One way to design software is using state machines. In event-driven systems like our game framework, the response to an event, like a keystroke or mouse click, can depend on the type of event *and* the internal state of the system.

For example, in this simple Caps Lock state machine, whether a key press results in a capital or lowercase letter depends on the state of the system at the time of the keystroke, that is whether we are in a Default Typing state or Caps Lock Typing state:


Caps Lock
State Diagram
Keystrokes (SHIFT, letters)
Keystrokes (SHIFT, letters)
Default Typing
CAPS LOCK pressed
CAPS LOCK Typing
done typing
CAPS LOCK pressed
Default Typing = SHIFT makes a capital, all other keystrokes lowercase
CAPS LOCK Typing = SHIFT does nothing, all other keystrokes uppercase

This is what we call a finite state machine. It represents the possible states and the pattern of events, states and state transitions in the system. We can draw a state machine as a directed graph with nodes representing states, and edges representing state transitions. Each transition depends on the incoming event and the current state. We can also represent the state machine as a table:

Event	**Current State**	**Result**	**New State**
Keystrokes	Default Typing	Lowercase unless SHIFT held down	Default Typing
CAPS LOCK pressed	Default Typing	Keyboard light on	CAPS LOCK Typing
Keystrokes	CAPS LOCK Typing	Uppercase	CAPS LOCK Typing
CAPS LOCK pressed	CAPS LOCK Typing	Keyboard light off	Default Typing

or

State	**Keystrokes**	**CAPS LOCK**
Default Typing	no change (default)	CAPS Typing
CAPS Typing	no change (caps)	Default Typing

In the Caps Lock state diagram, when we begin, we're in the Default Typing state in which all keystrokes result in lowercase unless we hold Shift to make an uppercase letter. Once we press Caps Lock, the keyboard light goes on and we transition to the Caps Lock Typing state in which all letters are rendered uppercase. While in the Caps Lock Typing state, we can press Caps Lock again, turning off the keyboard light and transitioning us back to the Default Typing state.

In the game framework, we can create an FSM (Finite State Machine) class to create a game play flow using states and events. Some common states to consider are Start Screen, Setting Up, Playing, Game Won, Game Lost, and Retry Screen. The states you choose to code depend on the structure of your game play, turn-taking, number of players, levels, and other factors you may design.

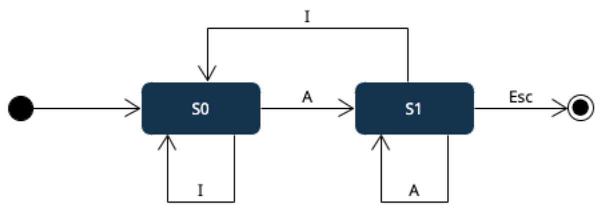
When implementing software using a state machine, it can make the code simpler and more reusable, reduce programming errors, and make it easier to test.

Exercises

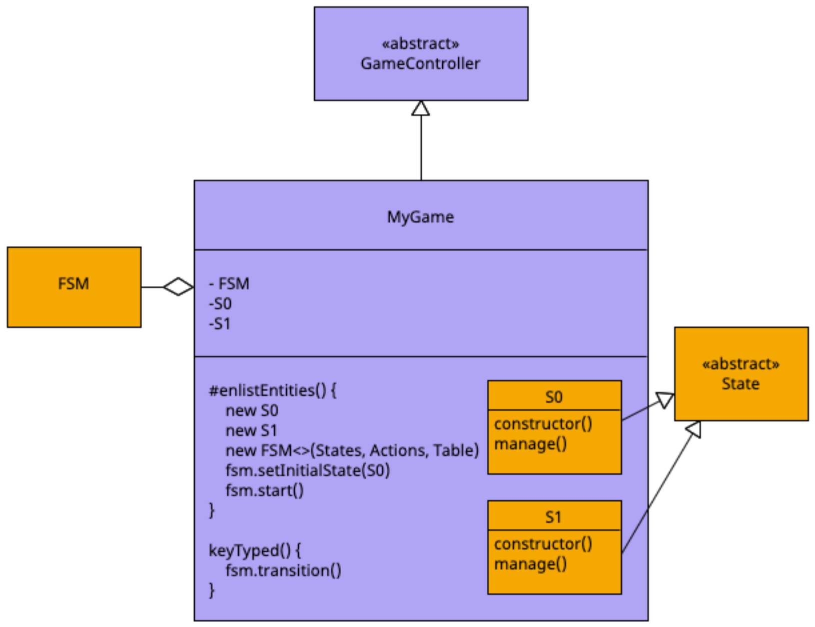
6A_TwoStates

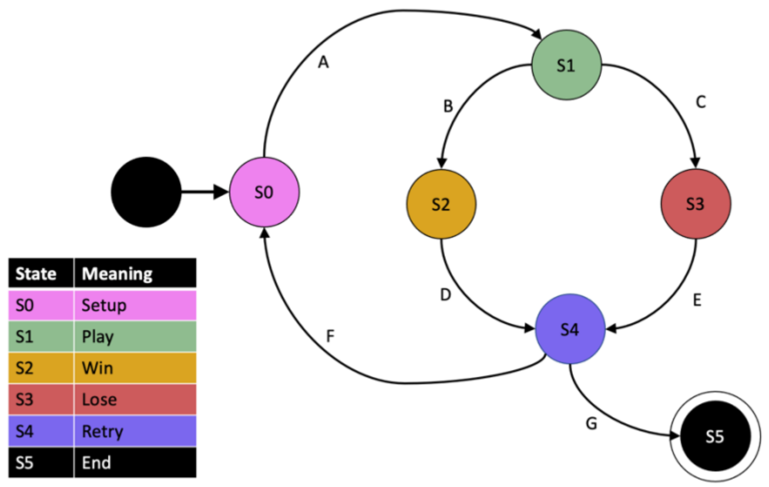
Make an alien that can be toggled Active or Inactive by user keystrokes. When the user presses ‘A’ the alien becomes active and animated. When the user presses ‘I’ the alien becomes inactive and motionless. Use the transition table and state diagram:

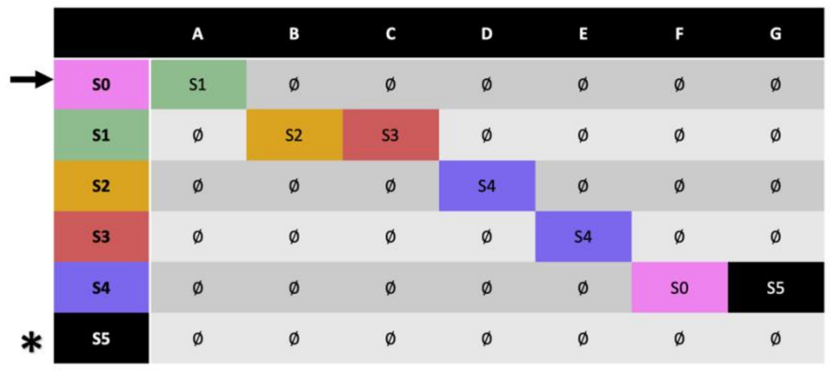
State	Press ‘A’	Press ‘I’
S0 (Initial state)	S1	S0
S1	S1	S0



Then create a Finite State Machine (FSM) class using the FSM class from the framework. You’ll also need to create two states using the framework’s State class.



	When there are just a few classes like this example, you could code them as inner classes. Later, when you are dealing with lots of classes, you may wish to even create a States container class to contain them all. This is encapsulation.
6B_ThreeStates	Draw a state diagram for a three-state alien, which can be Active, Inactive, or Moving depending on user keypresses. Use pencil and paper or UMLet. Then code to the diagram. When in the Moving state the alien should move randomly around the screen.
6C_BlockBreaker	The Block Breaker game is played as follows. A paddle at the bottom of the screen can move left and right using keystrokes. Blocks are positioned in a stationary matrix (2D array) near the top. A ball appears directly below the blocks and falls toward the South edge. The paddle must move to the ball to make it bounce off the paddle up toward the blocks. When a block is hit by the ball, the ball bounces off the block, and the block disappears. The game is won when all the blocks are gone. The game is lost when the ball gets past the paddle and strikes the South edge. Code the game using a paddle, an array of blocks, a bouncing ball, and an FSM class. Consider using some typical states: Setting up, Playing, Game Won, Game Lost, and Retry Screen. Here's a state diagram and transition table you can use if you'd like:  A B C D E F G S0 S1 S2 S3 S4 S5 State Meaning S0 Setup S1 Play S2 Win S3 Lose S4 Retry S5 End

	 Ø indicates no state change
6D_BlockBreakerPlus	Add barriers to your game for the ball to bounce off.

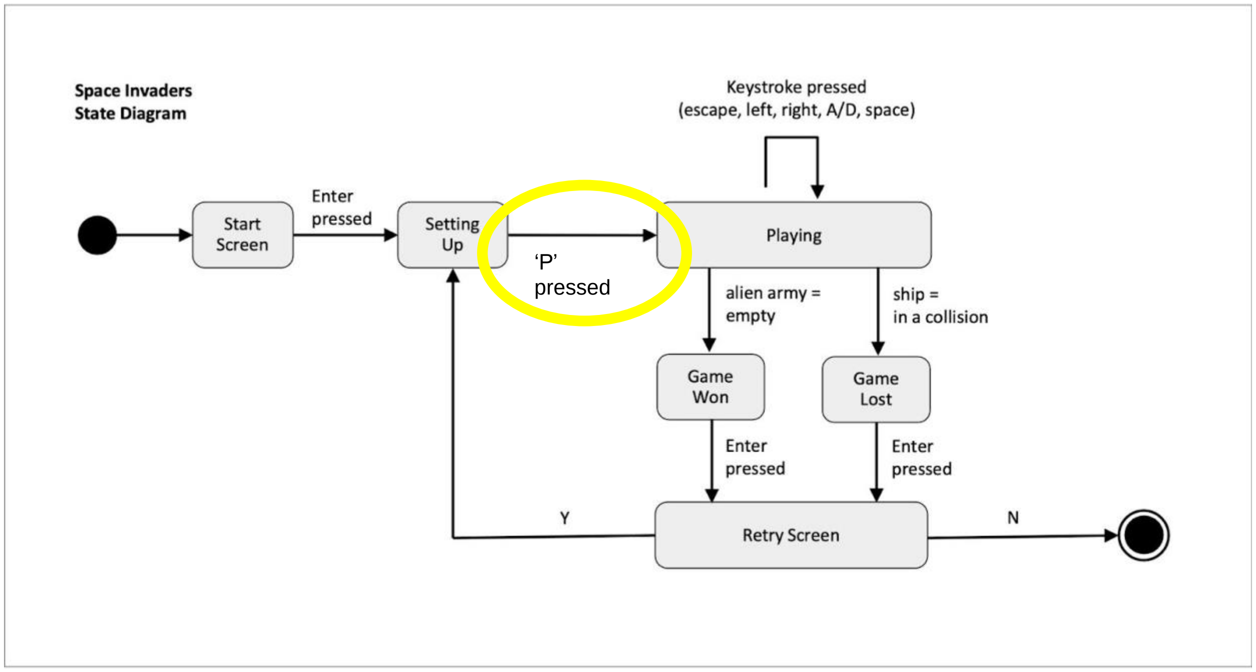
	A	B	C	D	E	F	G
→ S0	S1	Ø	Ø	Ø	Ø	Ø	Ø
S1	Ø	S2	S3	Ø	Ø	Ø	Ø
S2	Ø	Ø	Ø	S4	Ø	Ø	Ø
S3	Ø	Ø	Ø	Ø	S4	Ø	Ø
S4	Ø	Ø	Ø	Ø	Ø	S0	S5
* S5	Ø	Ø	Ø	Ø	Ø	Ø	Ø

Challenge

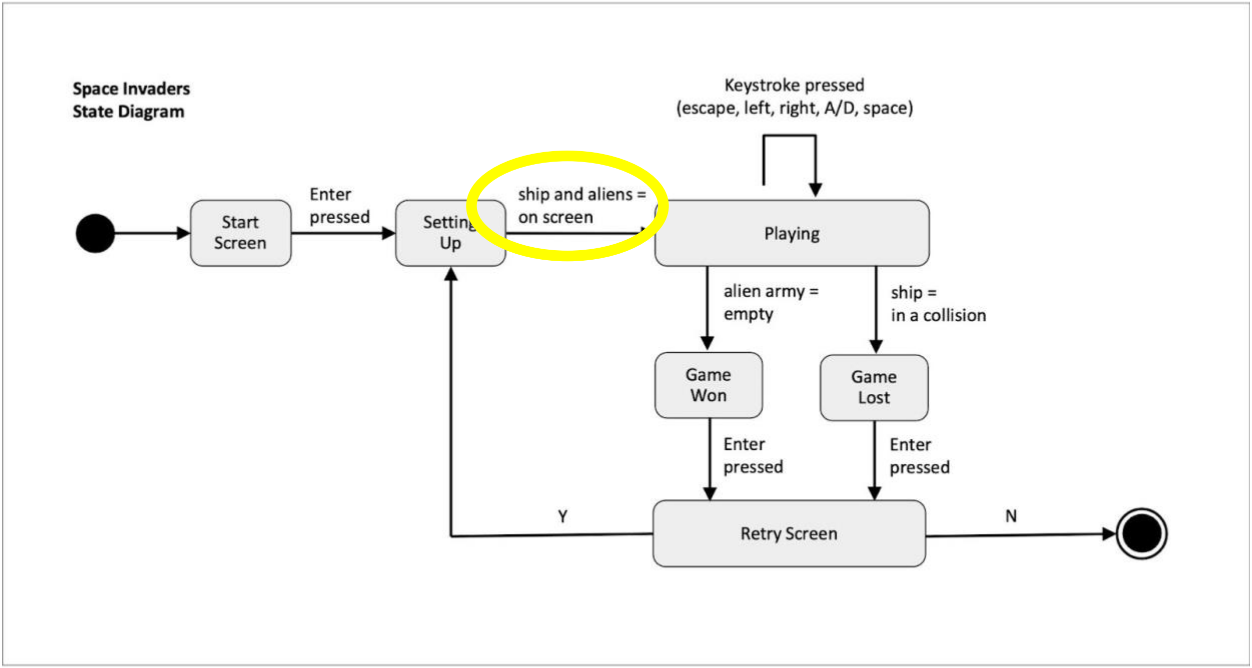
Code: (6E_SpaceAliensFSM) Enhance your Space Aliens code to make a full game of play using the following states: Start Screen, Setting Up, Playing, Game Won, Game Lost, and Retry Screen. Add barrier sprites for your ship to hide behind. Fireballs cannot penetrate the barriers. Also, your ship's missiles should not penetrate the barriers. Have two kinds of aliens in the alien group so that each type shoots a different kind of fireball. This is polymorphism.

Note: It is possible to have a transition from one state to the next that does not require an event or action. In this case, we are dealing with a monoid. This can be shown with an example of two different ways to design a state diagram.

The first diagram uses an action to trigger the next state:



The second diagram illustrates a transition monoid:



You may read about the formal definition of a state machine as well as monoids here:

https://en.wikipedia.org/wiki/Deterministic_finite_automaton#Formal_definition
https://en.wikipedia.org/wiki/Deterministic_finite_automaton#As_a_transition_monoid

The game engine's FSM is equipped to handle transition monoids by communicating the completion of a monoid state change to a state change event listener. Following is a detailed explanation.

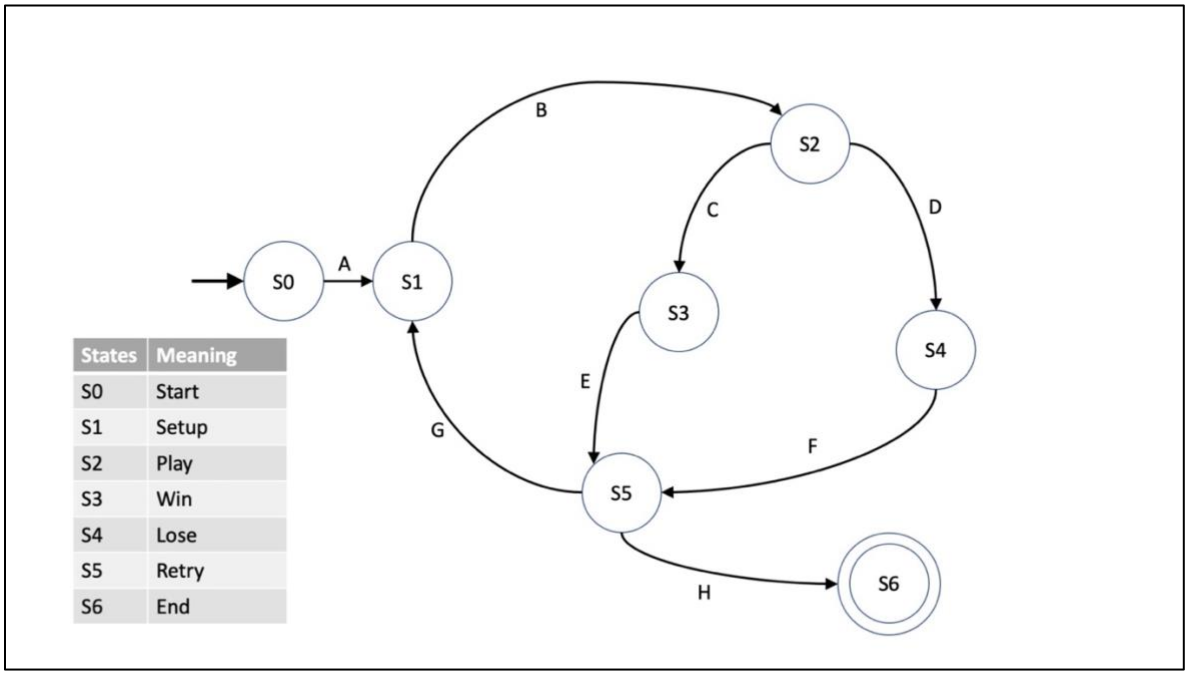
More on Finite State Machines

A Finite State Machine (FSM) or a finite automaton has a set of states it can be in. It transitions from state to state in response to external inputs or actions. The transitioning of a finite state machine from one state to the other is defined by a transition function δ(s, a), where 's' is the current state of the FSM, and 'a' is the action that causes the transition to occur. The framework provides a finite-state machine that can only be in one state at any one time. Such automatons are termed deterministic, as opposed to non-deterministic automatons that can be in more than one state at once. This definition makes the FSM provided a Deterministic Finite State Automaton (DFSA).

The FSM can be represented in two ways:

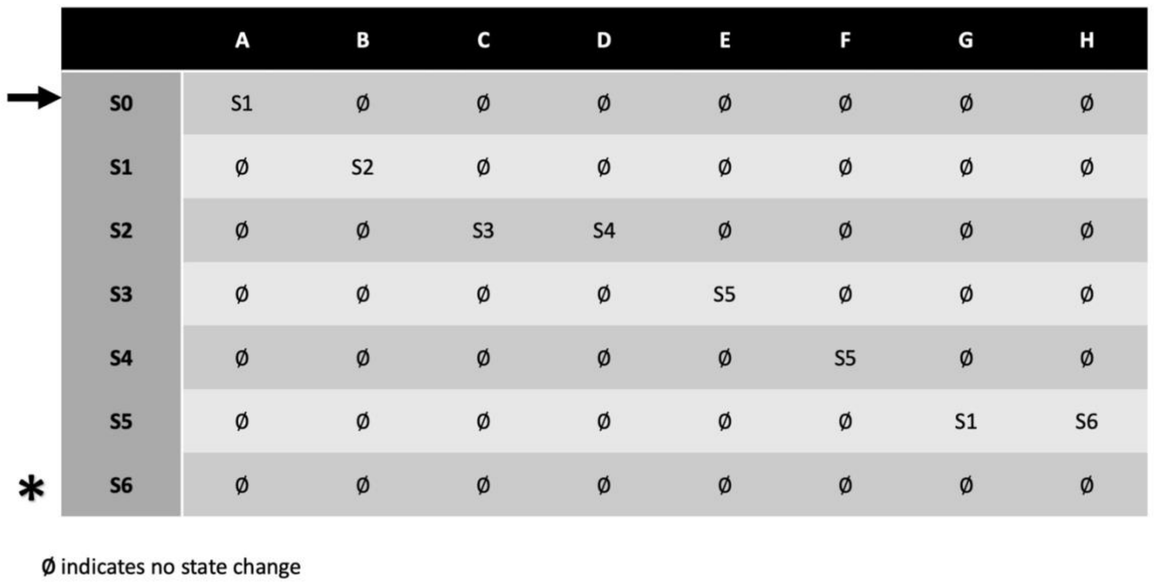
1. A transition diagram, which is visual and is a great way to design the FSM on paper
2. A transition table, which is a two-dimensional representation of all the transition functions, arguably harder to read that a visual representation

Transition Diagram
The figure below shows a transition diagram for a typical game. The game begins with a start screen S0, does the setup S1 and then transitions into the play state S2. The FSM stays in S2 while the game is in progress. If the gamer wins, the FSM transitions to state S3, and to S4 if the gamer loses. Once the outcome of the game is displayed on the screen, the FSM transitions to state S5, which prompts the gamer for a replay. If the gamer chooses to quit, the FSM transitions to the end state S6, indicated by the concentric double circles. If the gamer chooses to replay, the FSM transitions to S1 and the cycle repeats.



States	Meaning
S0	Start
S1	Setup
S2	Play
S3	Win
S4	Lose
S5	Retry
S6	End

Transition Table
The transition table representation of the FSM is typically represented as shown below, as a two-dimensional array of transition functions, δ(q, a). The entry in the table for a row corresponding to a state ‘q’ and the column corresponding to an input action ‘a’ is given by the function δ(q, a). The domain is the set of actions, and the range is the output of the δ(q, a) function. The transition table for the FSM is implemented in code as an associative array indexed by states. It is a generic type parameterized by the data type of the action class. Each entry in the array is an associative array of states that is indexed by actions. The description of the game state machine represented as a transition table is shown below.



	A	B	C	D	E	F	G	H
S0	S1	Ø	Ø	Ø	Ø	Ø	Ø	Ø
S1	Ø	S2	Ø	Ø	Ø	Ø	Ø	Ø
S2	Ø	Ø	S3	S4	Ø	Ø	Ø	Ø
S3	Ø	Ø	Ø	Ø	S5	Ø	Ø	Ø
S4	Ø	Ø	Ø	Ø	Ø	S5	Ø	Ø
S5	Ø	Ø	Ø	Ø	Ø	Ø	S1	S6
* S6	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

Ø indicates no state change

The start state is indicated by an arrow and the accepting or final states are indicated by a star alongside it. In this specific table ,the ∅ entries indicate an ignored action or a 'don't care'. If the FSM is in a state 'q' and reads no inputs, it remains in the same state. The transition table implementation accepts null entries.

Defining the FSM
The finite state machine is represented by a "five-tuple":

1. A finite set of states, denoted by Q
2. A finite set of input symbols represented by strings, denoted by Σ
3. A transition function δ for each state, corresponding to each action
4. A single start state q0 | q0 $\in$ Q
5. A set of final or accepting states F | F $\subset$ Q

A complete representation of an FSM 'A' can be shown as:

$$A = (Q, \Sigma, \delta, q0, F)$$

Creating an FSM
Once the FSM has been designed using a transition diagram, and the various states have been identified, the FSM can be realized in code.

Defining States
Each state of the FSM must be defined by extending the abstract State class. The recommended implementation would provide a constructor for each state that takes a reference to the game controller implementation. The FSM would have to reference the game controller to manage the setup of that state. The base class State uniquely defines the state name and provides room for a short description of the state as well. To organize and encapsulate all the states, provide a container class called States, within which all the states of the machine are defined as static inner classes. The States class represents the set of all states of the FSM (the finite set Q).

A state implementation for Exercise7A is shown below:

```
public class States {

        public static class S0 extends State{

                private Exercise7A controller;

                public S0(Exercise7A controller) {
                        super("S0");
                        this.controller = controller;
                }

                @Override
                public void manage() {
                        controller.showStartScreen();
                }

        }
        // Code removed for brevity
        // More states will be defined within this container class
        // …
        }
}
```

Defining the FSM
The FSM is defined by a class that takes the action data type (Type) as a parameter. The constructor of the FSM<Type> takes the state array, the actions for this FSM, and the state transition table as its input. The state transition table is also defined for a parameterized action type. To set up the game state machine mentioned earlier, which works with String type actions, the code would be as follows:

```
FSM<String> fsm = new FSM<>(new State[] { s0, s1, s2, s3, s4, s5, s6 },// States
             new String[] { "A", "B", "C", "D", "E", "F", "G", "H" }, // Actions
             new State[][] { // delta functions a.k.a transition table
                          // A, B,  C,  D,  E,  F,  G,  H
                          { s1, s0, s0, s0, s0, s0, s0, s0 }, // s0
                          { s1, s2, s1, s1, s1, s1, s1, s1 }, // s1
                          { s2, s2, s3, s4, s2, s2, s2, s2 }, // s2
                          { s3, s3, s3, s3, s5, s3, s3, s3 }, // s3
                          { s4, s4, s4, s4, s4, s5, s4, s4 }, // s4
                          { s5, s5, s5, s5, s5, s5, s1, s6 }, // s5
                          { s6, s6, s6, s6, s6, s6, s6, s6 }  // s6
                          });
fsm.setInitialState(s0); // defines initial state
fsm.setFinalStates(s6); // defines final state
fsm.addStateChangeListener(this); // add the state change listener
fsm.start(); // starts the FSM in its initial state
```

The methods invoked are self-explanatory. A variation of the FSM discussed is where the completion of a state transition triggers a subsequent state transition, without an input action. Algebraically, the input action on the state is rolled into the state as $\delta_a(q) = \delta(q, a)$. Such a representation is referred to as a transition *monoid*. When this manner of behavior is required, the FSM can communicate the completion of a state change to a `StateChangeListener` object. The listener is any class that implements the StateChangeListener interface, which lists the following function:

```
       public void stateChanged(StateChangeEvent event);
```

The FSM will notify the listener by invoking the callback function when a transition completes. This feature can be used to manage tandem state changes, where one state change triggers the subsequent one on completion. The addition of a state change listener is optional, and is to be used only when transition monoids are present in the FSM.

Using the FSM
Once an FSM has been instantiated and initialized for a specific action type, it is moved into the start state by invoking the `start():void` command. Following that, various actions can be input into the FSM by invoking the `transition(Action):void` method on the FSM instance. Upon invoking this method, the FSM will fetch the delta function for the current state and the specified action from its transition table, and invoke the `manage():void` method on that state. Since the transition table is a two-dimensional associative array, the lookup happens in near constant time. The FSM then settles on the new state and notifies the listener about the completion of the transition. The recommended place in the controller to trigger the state machine transition is the overridden `updateGameState():void` method provided by the game controller concrete class.

An example on triggering the transition on an action is shown below. The current action is emptied out at the end of each cycle to avoid redundant transition calls on the finite state machine.

```
@Override
protected void updateGameState() {
        // check if empty string
        if (currentAction.length() != 0) {
                fsm.transition(currentAction);
        }
        // avoid redundant actions
        currentAction = "";
}
```

In the state diagram shown for the game example, all actions are the direct result of user interaction to prompts, except for the action identified as “B”. The keyboard listener functionality can be used to map user inputs to corresponding action strings as shown.

```
@Override
public void keyTyped(KeyEvent e) {

        switch(e.getExtendedKeyCode()) {

        case KeyEvent.VK_Y:
                currentAction = "F";
                break;
        case KeyEvent.VK_N:
                currentAction = "G";
                break;
        case KeyEvent.VK_ESCAPE:
                dispose();
                System.exit(0);
                break;
        default:
                currentAction = "A";
                break;
        }
}
```

The `onKeysPolled` method should not be used for affecting state changes since the method gets invoked at the frame rate of the game, resulting in multiple redundant settings on the current action variable.

The transition from S1 to S2 happens without user interaction, and is a result of an internal transition completion. In other words, action ‘B’ is generated internally:

$$\delta_b(\text{“S1”}) = \delta(\text{S1}, \text{“B”})$$

This self-triggered FSM transition, with the game controller acting as an intermediary, is managed via the state change listener as shown below. The concrete game controller should implement the `StateChangeListener` interface to be able to listen in on the FSM.

```
@Override
public void stateChanged(StateChangeEvent event) {
	State newState = event.getNewState();
	if (newState.equals(s1)) {
		fsm.transition("B");
	}
}
```

A game that is built on sound object-oriented principles that is backed by an FSM will operate with provable consistency. The FSM class provided with the game framework is a generic type that is parameterized by the action that the FSM responds to. It can be instantiated for the action type used in the game. State machine-based designs are less prone to unresponsive stalls, since all transitions to end states are clearly defined and considered. It furthermore facilitates ”automata theoretic” testing of the control flow of the overall application.

7 Executable App

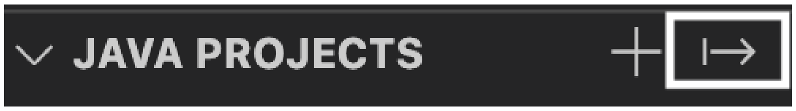
Objectives: VG1, VG2

- Create an executable file of your game.
- Demonstrate the use of polymorphism.

Lesson

Once you’ve developed your complete game, you’ll want to package it into an **executable** file for users. One way to do this is to export your package as a runnable *.jar file. The steps to do this are slightly different depending on which IDE you use (Eclipse, VSCode,etc).

VSCode
In VSCode, open the command palette (⇧⌘P) and type ‘java: export jar’. Or click the **Export Jar** button:



Eclipse
- Select “Package Export”
- Select “Java”
- Select “Runnable jar file”
- For run configuration, select “controller”
- For export destination, select “Downloads”
- Select “package required libraries into generated jar”

To **run the jar file**, from a Terminal, type:

```
java -jar filename
```

The jpackage Command (oracle.com)

Challenge

There are no Exercises for this lesson, only challenges.

Code: (7A_SpaceAliensComplete) Enhance your Space Aliens game to include the following features:

- A start screen which asks for and allows the user to enter their name.
- Occasionally a boss-alien (use your own sprite) should traverse the top of the screen, which should be destroyable by the spaceship with a missile.
- The user should be able to play multiple levels. Each level of play should have different game behavior (speed of aliens, boss-aliens, etc.). Be creative!
- A scoring system should be incorporated into the game, keeping track of aliens killed (1 point each) and the boss-alien (10 points) destroyed at each level.
- Display an end screen showing the current user's name and score.

Code: (7B_SpaceAliensPlus) Enhance your Space Aliens game per 7A and in addition, include the following bonus feature:

- At startup, display a screen showing the top three user names and high scores. Use a fake list to begin with.
- Allow the user to enter their name at the beginning. Then use persistent storage to keep track of the three highest scores on the machine to display upon restart of the game.

The game should have a class that contains all scores (ScoreBoard). This ScoreBoard class should be Serializable, which means it can be written to disk using the writeObject() method. Once written to disk, it can be read back with the readObject() method. See page 617 in *Absolute Java*.

Concepts

Algorithms

In a game engine, speed is critical. This is where choosing the right algorithm can help! An algorithm is the step by step procedure followed by a method. Collision detection is one algorithm accomplished in the checkForCollisions method of the Game Controller. If the algorithm used to process the entities can be improved or made faster, more entities can be processed within the game loop. The game engine utilizes nested for loops for this purpose, resulting in O(n^2) complexity.

This is Big O notation used to analyze algorithms. It represents the worst-case performance of a specific algorithm. Here is a graph of functions commonly used in the analysis of algorithms. As the number of entities (n) processed grows, the number of operations grows according to the function. In our case of collision detection, the n comparisons done in the nested for loops increase just as the graph of n^2 does. This means the computational delay can increase really quickly as more and more entities are involved. If we can find a better data structure and algorithm to accomplish the same task, we can improve the performance and potentially process more entities.

We also used nested for loops when we created our alien army array. Likewise, the algorithms we used to process that matrix have complexities of O(n^2). Many of the alternative algorithms with lower big O values don't work for games because these improved algorithms are predictive in nature. When the user can move the entities and change the velocities, it becomes impossible to predict! Do you think you can come up with a better algorithm?

Read more in *Absolute Java* pages 134-137 (algorithms in general), 672-683 (binary search), 388-391 (selection sort), and 701-708 (sorting)

APIs

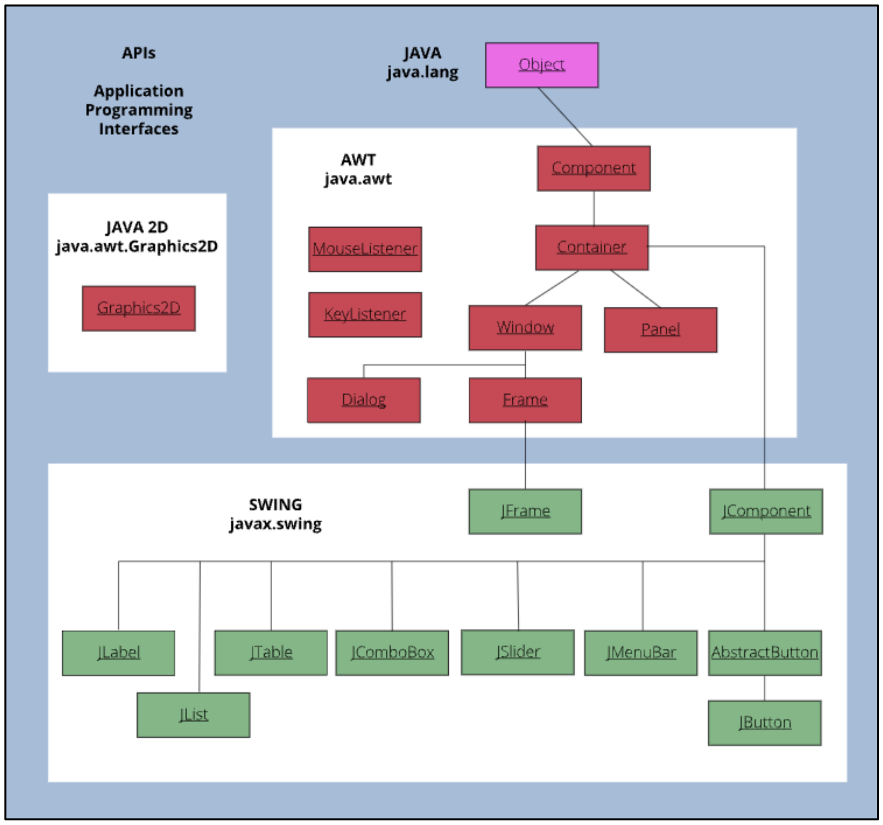
An **application programming interface** (**API**) is a software interface that provides a service to another piece of software. The instructions for how to use the API is called an API specification. When a programmer uses an API, they call the API methods. The API defines the method calls and how to use them.

Java Foundation Classes (JFC) is an API that provides services to this game engine and which you may want to use when creating your game. JFC bundles graphical user interface (GUI) components and services including Java AWT, Java Swing, and Java 2D.

- Java AWT API (java.awt)
    - AWT stands for Abstract Window Toolkit.
    - Platform specific
        - GUIs developed with AWT are platform specific (they have a look and feel depending on the operating system, that is, the GUI looks different on Mac and Windows)
    - Heavyweight
        - Utilizes the operating system, so can be slow
- Java Swing API (javax.swing)
    - GUI framework
    - GUI widget toolkit providing buttons, labels, menus, checkboxes, text areas, text fields, etc.
    - Built on top of AWT
    - Platform independent
        - The look and feel remains the same across platforms
    - Lightweight
        - Uses Java instead of the operating system, so it renders quickly
- Java 2D API (java.awt.Graphics2D)
    - Graphics API
    - Renders, manipulates and transforms complex 2D images and text

How are these APIs used in the game engine?

Swing: First, the GameController extends a **Swing** JFrame. Here you can see the JFrame class and some of the other classes provided by Swing.



References:
[javax.swing (Java SE 19 & JDK 19) (oracle.com)]
[java.awt (Java SE 19 & JDK 19) (oracle.com)]
[Graphics2D (Java SE 19 & JDK 19) (oracle.com)]

AWT: The game engine also uses an **AWT** WindowListener to respond to the window's status changes when it is closed, and it implements the **AWT** EventListener KeyAdapter interface to respond to keystrokes.

Java 2D: The game engine renders images and text using **Java 2D** Graphics2D, BufferedImage, AffineTransform, and FontMetrics objects which make use of **AWT** Font and Color objects.

Read more in *Absolute Java* chapters 17 and 18

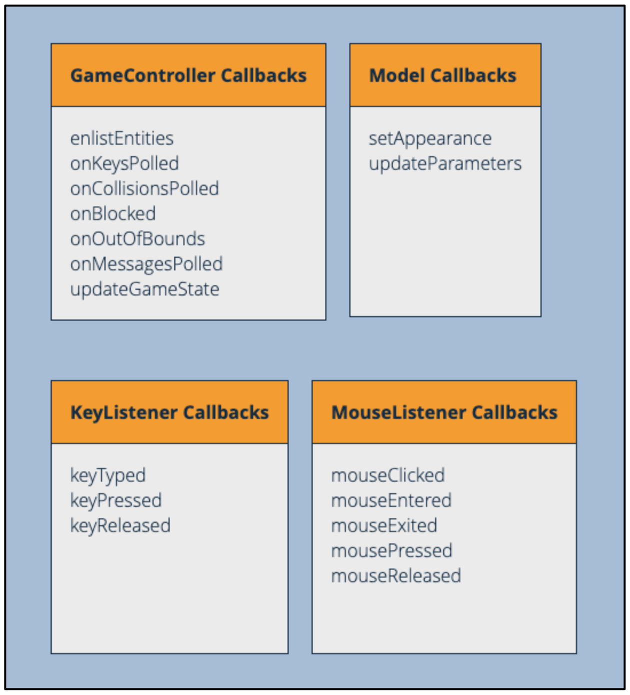
Callbacks

What is a callback?

A callback is a method that you write, but you don't invoke or call yourself. When you start your game, the framework calls your method. The framework "calls you back."

Let's look at an analogy. You go to the mechanic's shop to get your car's oil changed.
You leave the car at the shop. You walk three blocks away to the shopping center while the oil is changed. You know how to get back to the mechanic shop, but you don't initiate the trip back.
You wait for a callback from the mechanic. When your car is ready, the mechanic calls you back, initiating your trip back to the shop.

Here are the callbacks you need to implement in your game. If you add a Key or Mouse Listener to your game controller, you'll need to implement those callbacks as well. Callbacks are also called event handlers.



Read more in *Absolute Java* chapter 17

Data Structures

A **data type** is a grouping of data specified by a set of possible values and operations.

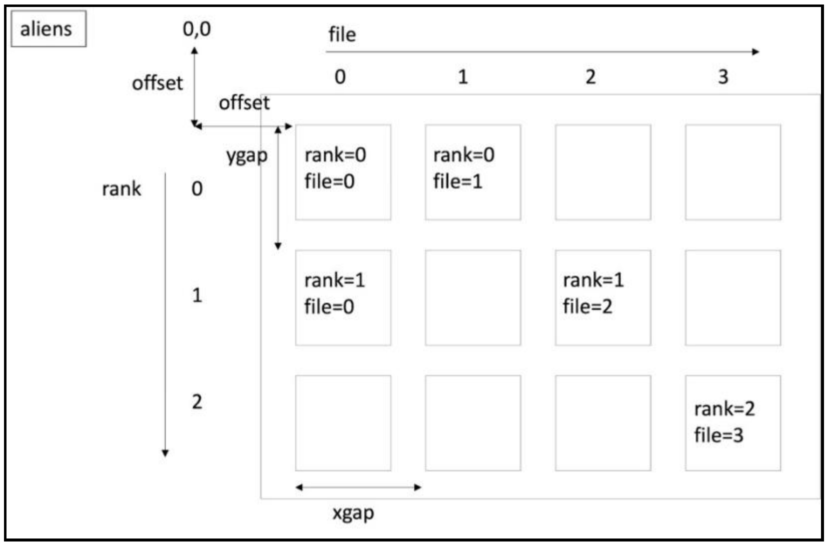
For example, an integer data type can hold a mathematical range of whole numbers, and allows operations such as +, -, *, and /.

An int is called a **primitive** type in Java because the Java language implements it for us. Some other common primitives in Java are boolean, double, and char.

Java also allows **user-defined** data types like classes and interfaces. Strings are user-defined since they are objects. But Java provides a lot of support for Strings, so you may almost think of them as primitives.

We use the term **Abstract Data Type (ADT)** when we refer to the model that explains how a data type, or collection of data, behaves. In contrast to this, we use the term **data structure** when we're talking about the physical implementation of the ADT in a computer language.

An **array** is a user-defined data type in Java. A one dimensional (**1D**) array is a single row, while a two dimensional (**2D**) array is a matrix. To create an army of aliens in a 3 x 4 matrix,



We first define the array named aliens:

```
SoldierAlien[][] aliens = new SoldierAlien[2][3];
```

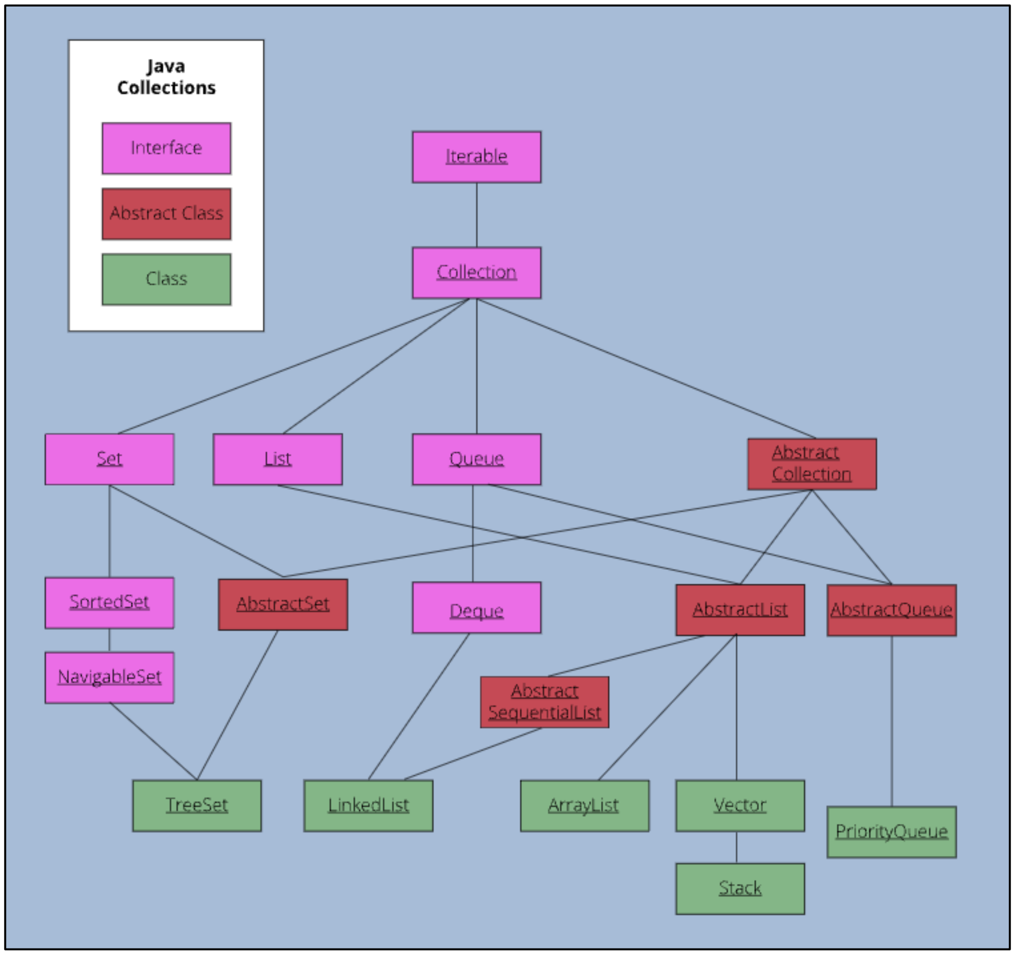
Then we may iterate over the array to fill it with SoldierAliens or search it for a particular SoldierAlien:

```
for (int rank = 0; rank < aliens.length; rank++)
      for (int file = 0; file < aliens[rank].length; file++) {
            aliens[rank][file] = new SoldierAlien();
}
```

A **hashmap** is another data structure used in the game engine. In the SpriteView, a hashmap stores data in a dictionary of key-value pairs. The animated sprite sheet images are stored this way so the code does not have to keep opening the files from disk.

Other examples of data structures are linked lists, queues, stacks, and trees.

The **Java Collections API** in **java.util** provides lots of data structures. This API gives you access to reusable data structures that have been tested and work, rather than risking creating them yourself. One commonly used structure is the ArrayList which is an ordered list that can shrink and grow.



The Collection classes make use of inheritance and interfaces. The collection classes are also **generic**, so for example, List<E> and ArrayList<E> contain elements of a generic type, E. This is how we're able to declare ArrayList<SoldierAlien> to create a list of soldier aliens. Collection classes also supply useful functions like iterating, searching, sorting.

Read more in *Absolute Java* chapters 6, 14, 15, and 16

Graphics

Entities

The game engine provides four basic **entity** types, Sprites, Text, Scenery, Barriers, and handles rendering their graphics on the screen for you.

To create game graphics, extend one of the entity classes, **SpriteModel, TextModel, SceneryModel, or BarrierModel.** (It is not recommended to extend EntityModel.) Use the **setAppearance** method to call **setView**(), creating a reference to the model's view. For example, if you're using a SpriteModel, the call to setView will extend the SpriteView class for you. Extend the GameController to make your game. In your game controller, use the **enlistEntities** method to create (using the new keyword) and call **addEntity**() for each entity model you would like.

Animation

The GameController instantiates and starts a GameClock which has a Timer and an inner class which extends TimerTask. Timer and TimerTask are found in the java.util package. Inside the TimerTask, SwingUtilities.invokeAndWait is used to call the drawEntities method on the Game Controller. So each time the Timer goes off, the call to drawEntities invokes the rendering of entities, creating the animation effect.

The Timer goes off every 16.67 ms, unless you override the GameController's rate. 16.67 ms was chosen based on the typical screen refresh rate in a video game, which is 60 fps. 1000ms/60 frames = 16.67ms per frame. So the game engine refreshes every 16.67 ms by default. Note that the computation of the location of entities is independent of the frame rate.

To override the default frame rate, pass the desired frame rate as an argument when you instantiate the GameController:

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new Game(100));
}
```

Drawing

The game engine uses a **graphics context** from the Java 2D API Graphics2D class. Built on top of the Graphics class, Graphics2D uses methods like draw, fill, drawString, and drawImage. It can also composite (with layering), transform (or rotate images), draw fonts, and antialias.

There are two ways that drawing graphics can be accomplished. One is called **system-triggered** painting; the other is **app-driven** painting. System-triggered painting uses the paint() and repaint() methods and happens automatically when the system thinks the GUI window is "dirty" or in need of a redraw. For example, a redraw might be triggered if another window moves on top of it.

This game framework, on the other hand, uses app-driven graphics. Because a game has to be updated, or drawn, constantly rather than as the system deems, the app-driven approach is needed. The game's EntityView is a JPanel and has a draw method in it. When the game controller's drawEntities method is executed, it calls each View, passing it the graphics context for that component. For sprites, it renders each frame from the sprite sheet, for scenery, it renders the image in its next position, and for text, it draws the glyphs. Once everything is drawn into a buffer, the framework then displays it all at once to the screen using the BufferStrategy.show().

Coordinates

The coordinate space utilizes the Java 2D user space, which starts at top left 0,0. The X coordinate gets larger as you move to the right. And Y gets larger as you move down the screen. Use these coordinates when instantiating sprites or using **setxLocation**() and **setyLocation**(). Use this coordinate system when specifying the location of your game entities.

Image Files

In the framework, image files are stored in the **res.drawable** folder. The **naming convention** for a sprite file is to give it name followed by a dash followed by the number of animation frames and ending with dot png. Save the file in the **png format**. If you want the sprite to be animated, create a **sprite sheet** with all the frames of animation. The framework will handle rendering the animation. All you need to do is add the **animationFrames** (persistence value) value as to the number of frames you want animated inside your **setAppearance** method.

Sprite Sheets

To create your own sprite sheets, first download and install GIMP.
GIMP - Downloads

Steps to make a sprite sheet:

1. Open GIMP,
2. Select GIMP, Settings, Folders, Scripts to see your scripts directory.
3. Go to GitHub, https://github.com/malteehrlen/tilemancer.
4. Download the code .zip file for the tilemancer script.
5. Unzip in your scripts directory.
6. Filters > Script-Fu > Refresh Scripts to check if tilemancer is properly installed. You should see a tilemancer option under Filters > Animation > tilemancer.
7. In GIMP, select File, New
8. Make a 100 x 100 file
9. Select a layer in the bottom right
10. Right click, add transparent layer
11. Add as many layers as you need.
12. Right click to delete the solid background layer.
13. Create your sprite - a different animation frame on each layer
14. Select Filter, Animation, tilemancer.
15. For Shape, select One Row Per Layer Group.
16. File, Export As, and save as a .png file. Use the naming convention, image-X.png, where X is the number of frames.

Scenery

If you make your own scenery image for scrolling, be sure to make the right edge match the left edge so the scroll will seem smooth.

References:
Lesson: Overview of the Java 2D API Concepts (The Java™ Tutorials > 2D Graphics) (oracle.com)

Double Buffering and Page Flipping (The Java™ Tutorials > Bonus > Full-Screen Exclusive Mode API) (oracle.com)

GraphicsEnvironment (Java SE 19 & JDK 19) (oracle.com) to check for supported fonts (java.awt.GraphicsEnvironment.GetAvailableFontFamilyNames or GetAllFonts.

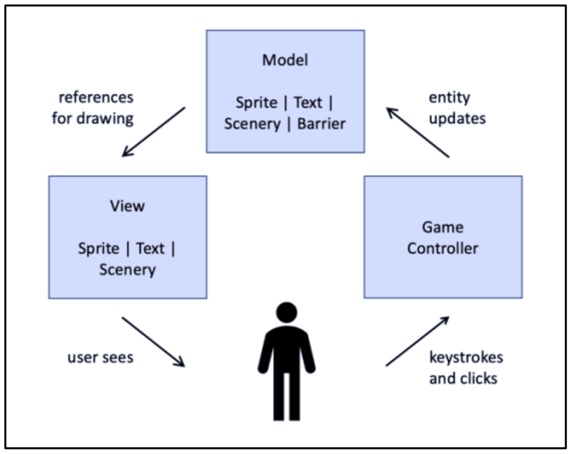
oracle.com/java/technologies/painting.html

Read more in *Absolute Java* chapter 18

MVC

Model-View-Controller

The game framework is organized using the model-view-controller design pattern. The user interacts with the controller using the keyboard and mouse. The controller executes the game loop logic. The controller keeps the Model objects updated with information associated with the state of sprites, text, and scenery. The model contains a reference to the View for drawing each of these entities. The View is what the user sees on the screen.



Read more in *Absolute Java* pages 700 and 988.

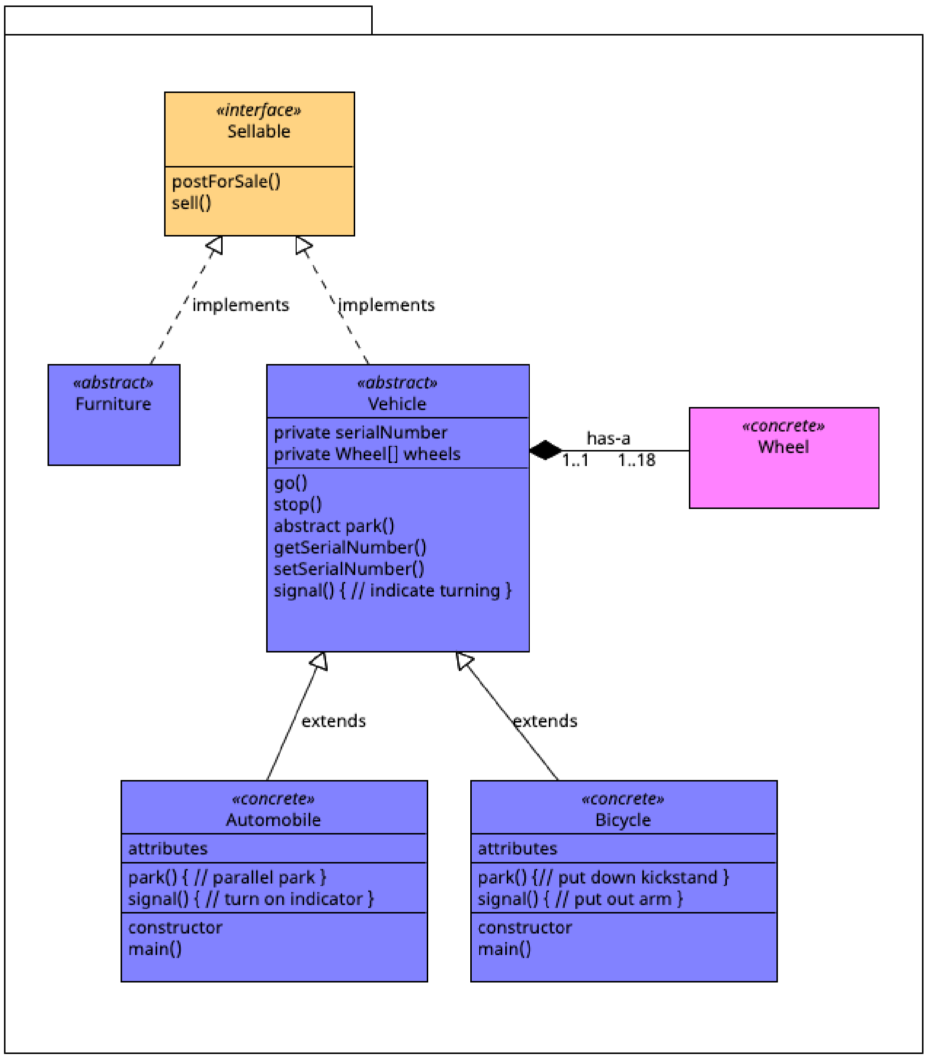
OOP

A vehicle is a **type** of object. An automobile is a type of vehicle. A bicycle is also a type of vehicle. Each of these is a **class**, or type, of object.

A Vehicle has **attributes** (like serial Number) and **methods** (like go and stop). These methods are **inherited** by subclasses like Automobile and Bicycle. They too have a serial number and can go and stop. Automobile and Bicycle may have additional attributes and methods.

For Automobile, openTrunk()
For Bicycle, ringBell()

Vehicle is a **super class** and Automobile and Bicycle are **subclasses**. They **extend** their superclass.



MyToyota is an **instance** of an automobile. MyBicycle is an instance of a bicycle.
Automobile myToyota = new Automobile();
Bicycle myBicycle = new Bicycle();
MyToyota and MyBicycle are **objects**. They are each an instance of a class.

We can get attributes from each, myBicycle.getSerialNumber()
And perform methods on each, myToyota.go();

These methods are available to Automobile and Bicycle by inheritance from their superclass, Vehicle.

These classes have **is-a** relationships. An automobile is-a vehicle. And a bicycle is-a vehicle.

Some classes have **has-a** relationships. A vehicle has-a wheel. In fact it can have up to 18 wheels.
An automobile has four wheels.
wheels = new Wheel[4];
A bicycle has 2. wheels = new Wheel[2];

The instances, myToyota and myBicycle, have wheels since they inherit them from their superclass.

Object-oriented programming requires thinking in terms of **objects** and **classes**.

Classes each require a **constructor** which is like a special method executed right after a new object of its type is created.

public Vehicle() { // constructor }
public Automobile() { // constructor }
public Bicycle() { // constructor }

Classes may be **abstract** or **concrete**. You can create an instance of a concrete class, but you can't create an instance of an abstract class. Abstract classes provide some functionality that should be extended to a concrete class, then instantiated. Abstract classes may or may not contain abstract methods requiring implementation in the concrete class. For example, if Vehicle is abstract, it could contain some abstract methods which must be implemented in its concrete subclasses. Vehicle may contain a park() method since all vehicles must be able to park. But each subclass will park in a different way. So they are required to implement the method as they see fit.

For an automobile, park() { // parallel park }
For a bicycle, park() { // put down kickstand }

In some sense, this requirement may be thought of as **overriding** the superclass' method.

Interfaces are used for functionality that applies across classes. Vehicles and furniture are both Sellable. Each class **implements** the Sellable interface which may contain methods such as postForSale() and sell().

While classes and constructors are often declared **public**, attributes and methods are often **private**, making them hidden or **encapsulated** for security and control. Attributes are often made accessible by **getter** and **setter** methods.

Polymorphism, the concept of providing multiple forms of something, is present in OOP in two ways: overriding and overloading methods.
First, polymorphism comes into play when there are classes related by inheritance. If a method in a subclass **overrides** a method in its superclass, then which method gets called on an instance depends on the type of object being referred to, not the type of the reference variable.

If Chair defines showDimensions() { // show height, width and depth }
And HighChair, overrides showDimensions() { // show h, w, d, and tray dimensions }

Chair c = new HighChair();
c.showDimensions() would show h, w, d, and tray dimensions because c has an object *class* of HighChair, not Chair.

This is determined at runtime, so it's called **late binding**.

In the game framework, polymorphism is used when the engine iterates over the entities (including sprites, text, scenery, and barriers). Each entity's type is determined through late binding.

Late binding contrasts **early binding** in which the type of object is determined at compile time:
HighChair c = new HighChair();

Anytime **overloaded** methods are used, this is also polymorphism, only using early binding. Overloaded methods have different method signatures and different parameters types. Overloading must use early binding on the type, not the class of object, if that parameter is an object.

Getting Started

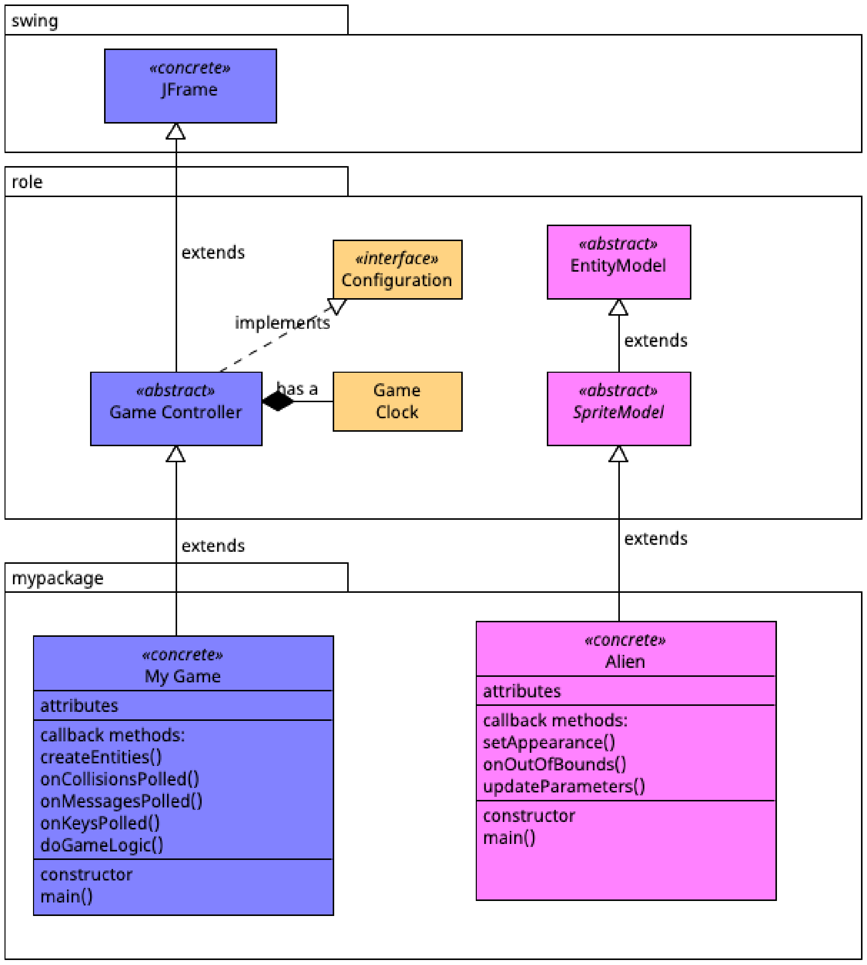
In the game framework, you will extend the abstract **GameController** and **SpriteModel** classes to create concrete classes of your own. You will implement, and in a sense, override the abstract methods from each abstract class. The GameController has a Game Clock which is the control loop for the game. The Game Controller extends JFrame, so your concrete game has all the functionality of the JFrame class as well.

public abstract class GameController extends JFrame implements Configuration

Game Controller also implements the Configuration interface, so your subclass also inherits the screen edge enumerations- North, South, East and West- from that interface.

Both classes in your new package (or folder) need a constructor which you will write. Your game class will also need a main() method which instantiates your game:

```
public static void main(String[] args) {
  SwingUtilities.invokeLater(()->new MyGame());
}
```



Inner and anonymous classes

Inner (or **nested**) classes are defined inside another class. This can make code more readable when classes belong together.

```
class Outer() {
      . . .

      class Inner() {
            . . .
      }
}
```

Anonymous classes are inner classes with no name:

new Inner() { }

Anonymous classes are expressions that must be part of a larger statement, such as when we pass anonymous classes as parameters in a method call. For example, during game play, you may wish to generate new instances, say, new Alien sprites. You can do this using an **anonymous** class. The object is created and passed all at once to addEntity:

```
private void generateAlien() {


  addEntity(new Alien(50, 50, 2) {


    @Override
    protected setAppearance() {
        setActive(true);
        setView("alien-3.png");
        setxVelocity(-20f);
    }

    @Override
    public void updateParameters(long elapsedTime) {
    }

    @Override
    protected void onOutBounds(OutBoundsEvent event) {
    }

  });
}
```

Note that anonymous inner types can be created from Interfaces as well.

Read more in *Absolute Java:*
Encapsulation, overloading, and constructors in chapter 4
Inheritance and overriding in chapter 7
Polymorphism in chapter 8
Abstract classes in chapter 8
Interfaces in chapter 13
Inner classes in chapter 13
Anonymous objects on page 298

UML Diagrams

The **Unified Modeling Language (or UML)** is a standard for visualizing software as well as other systems. It is important to know some UML so you can understand the architecture and behavior of software you will be working with and so you can communicate your software designs to others.

We'll look at five types of UML diagrams: **activity, class, object, sequence, and state diagrams**.

UML diagrams fall into two categories: **Structural** (static) and **Behavioral** (dynamic).

Class and object diagrams are structural.
Activity, sequence and state diagrams are behavioral.

There are some standard UML **symbols** for each type of diagram. However, keep in mind, you do not need to use all the features of UML in each diagram. Sometimes, your diagram will be more clearly understood when only the minimum information is included.

Let's look at some of the symbols available for each type.

A **class diagram** shows the static structure of a system. It shows how the classes are related.

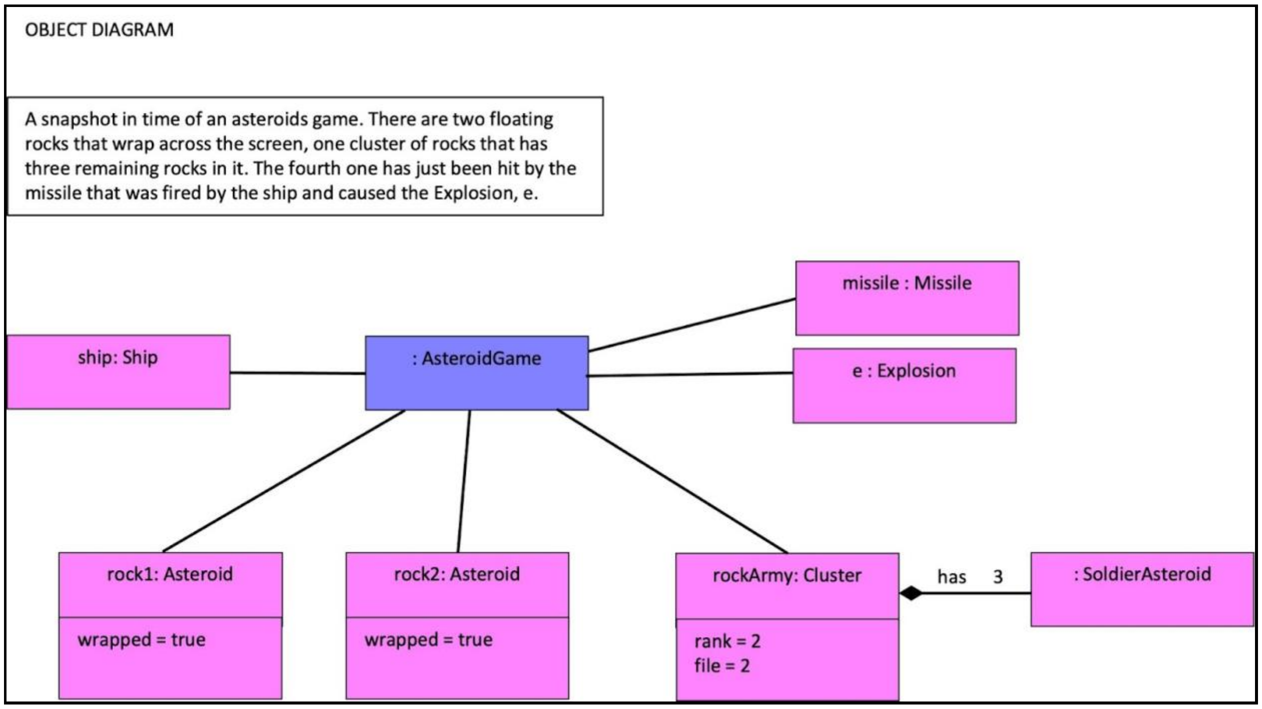
A class box has 3 sections: class name, attributes, and methods.
Abstract class names are in italics and have the <<abstract>> specifier.
Interfaces have the <<interface>> specifier.
Methods may be marked according to their visibility:
 + public, - private, # protected, or ~ package
Inheritance (is-a relationship) is shown with a hollow arrow pointing toward the superclass.
Composition (has-a) relationship is shown with a filled diamond on the whole class.
A hollow diamond indicates the whole class object may be destroyed without destroying the associated class objects. (aggregation or collection)
The cardinality may be shown along the association line.
Implementation (of an interface) is shown with a dashed line and open arrowhead.
If a class implements methods in the interface, the arrow to the interface should be closed but hollow.

The website home page for this game engine is an example of a class diagram.

An **object diagram** is a snapshot, or the static structure of a system at a particular time. While class diagrams show classes, object diagrams display instances of classes (or objects).

An object rectangle contains the object name followed by a colon and the object's class name.
Anonymous objects have no object name listed, and instead begin with a colon.
The object's attributes and their values are listed using an equals sign in the second section of the box.
Links are shown using the same symbols in class diagrams. (line, hollow arrow, filled and hollow diamond)

Try to avoid lines crossing each other, and keep it simple when you can.


OBJECT DIAGRAM
A snapshot in time of an asteroids game. There are two floating rocks that wrap across the screen, one cluster of rocks that has three remaining rocks in it. The fourth one has just been hit by the missile that was fired by the ship and caused the Explosion, e.
missile : Missile
ship: Ship
: AsteroidGame
e : Explosion
rock1: Asteroid
wrapped = true
rock2: Asteroid
wrapped = true
rockArmy: Cluster
rank = 2
file = 2
has 3
: SoldierAsteroid

An **activity diagram** is like an advanced flowchart. It can be used to model an algorithm.

We'll use the following symbols (although there are more):
Initial state, final state, action box, decision diamond

The game loop is shown with an activity diagram.

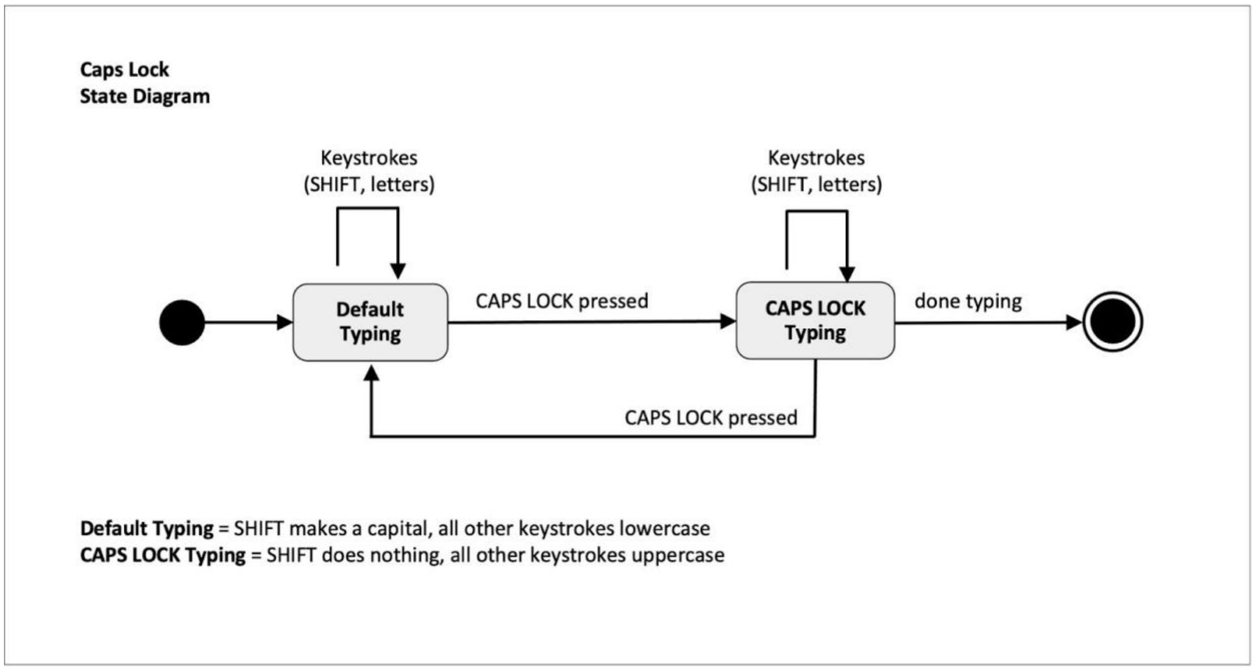
A **sequence diagram** shows interactions among objects in terms of an exchange of messages over time.

Lifelines show the passage of time as it extends downward
Person shows an entity that interacts with the system (such as a user pressing a key)
Objects are shown as rectangles
Rectangles also show activation, or the time an object is completing a task
Sequence diagrams and loops are shown with a box, labeled in upper left corner
Messages between objects are shown with a variety of links
Arrow with filled head = synchronous
Arrow with open head = asynchronous
Dashed arrow return messages
The word <<create>> over a message when a new object is created
An X on the lifeline when an object is destroyed

A **state diagram** shows the dynamic behavior of a system in response to stimuli or events.

Rounded rectangles are used for states.
Initial and final states are the same as for an activity diagram.
Arrows are marked with triggers, events, or guard conditions.
Diamonds may be used for decisions, and a circle with an X inside for escape from the state machine due to errors or other issues.

Caps lock is a great example.


Caps Lock
State Diagram
Keystrokes (SHIFT, letters)
Keystrokes (SHIFT, letters)
Default Typing
CAPS LOCK pressed
CAPS LOCK Typing
done typing
CAPS LOCK pressed
Default Typing = SHIFT makes a capital, all other keystrokes lowercase
CAPS LOCK Typing = SHIFT does nothing, all other keystrokes uppercase

Read more in *Absolute Java* chapter 12

IBM Documentation

User Interface

Windowing Toolkit

User Interfaces (**UI**) or Graphical User Interfaces (**GUI**) allow users to interact with the software through visual icons and sound indicators, in an easier way than a command line interface. Often, a user interacts with the software through GUI widgets like text boxes and buttons. **Swing** is a **windowing toolkit** used to create the GUI for the game engine. The Game Controller class is a Swing JFrame, with a default width and height, centered on your screen.
To change the title of your controller JFrame, call setTitle() in your game constructor. You can call methods on Frame or JFrame to customize your game window.

Event-driven Programming

GUIs are **event-driven**. There is an event dispatch thread (**EDT**) which handles tasks in the GUI's event queue. If other threads in the software try to access the GUI, there can be problems. The tasks (or events) are processed by the EDT one after the other serially and should be short so as not to freeze the GUI which must remain responsive. Long tasks should be done in the background, and not in the EDT.

Listeners
Tasks (or events) in the EDT are primarily
1) updates that cause user interface components to **redraw** themselves or
2) input events like **keystrokes** or **mouse clicks**.

The items in the game engine that redraw themselves are the Swing components (JFrame and widgets) and game entities. To make sure the Swing components are instantiated in the EDT, we use the

SwingUtilities helper class to pass the game controller code to execute in the EDT. In your game controller's main() method, we say SwingUtilities.invokeLater(new MyGame()).

Listeners are used to handle window, mouse, and keystroke events. The Game Controller adds a **window listener** to itself to respond to the game window's status changes when it is closed, and it implements a **multikey listener** so it can respond to keystrokes. This listener is specially designed to handle multiple keys getting pressed at the same time, like moving a sprite diagonally. If you want your game to respond to a particular individual key press, add your own KeyListener instead. The multikey listener communicates multiple key presses to the GameController through the onKeysPolled method. The multikey listener is only for active play. If you want your game to respond to mouse controls, implement **MouseListener** (java.awt.event.MouseListener) when you extend GameController and code the mouse methods from the interface.

Callbacks
When a listener object is created, it gets registered with a component. When events happen on the component, the listener methods are "called back" for execution. This is why listener methods are called **callback methods**. Some callback methods in your game controller are the mouseClicked, Pressed, Released, Entered, or Exited if you've implemented MouseListener. The windowClosing method is a callback implemented when the game controller creates the window listener as an anonymous class. The MultiKeyListener in the game engine contains callback methods for keyPressed and keyReleased which accumulate the keys pressed, even multiple key presses. These callback methods are handled as the events queue and are processed by the EDT.

Many of the **abstract methods** you are responsible for implementing are essentially callbacks as well. You must define them, but you don't call them directly. In this case, the Game Clock calls them. For example, pollKeyboard and onKeysPolled are methods that get called from the game loop. pollKeyboard is handled by the framework, and pollKeyboard then calls onKeysPolled, which you are responsible for defining in your game.

Sound Effects
Another aspect of the user interface are the sound indicators that are part of the game experience. Sound effect audio files are located in the **res.audio** folder. The SoundEffects **enumeration class**, which uses javax.sound.sampled, provides a way to play, loop, and stop sound files and set their volume. Some basic sounds are provided like **WIN, LOSE, COLLIDE, STARTUP, SHOOT, and BOUNCE**. These capabilities are inherited by your game when it extends the Game Controller.

Reference Links:
Event dispatching thread - Wikipedia

javax.swing (Java SE 19 & JDK 19) (oracle.com)

Listener API Table (The Java™ Tutorials > Creating a GUI With Swing > Writing Event Listeners) (oracle.com)

LineListener (Java SE 19 & JDK 19) (oracle.com)

Read more in *Absolute Java* chapters 17 and 18

The End