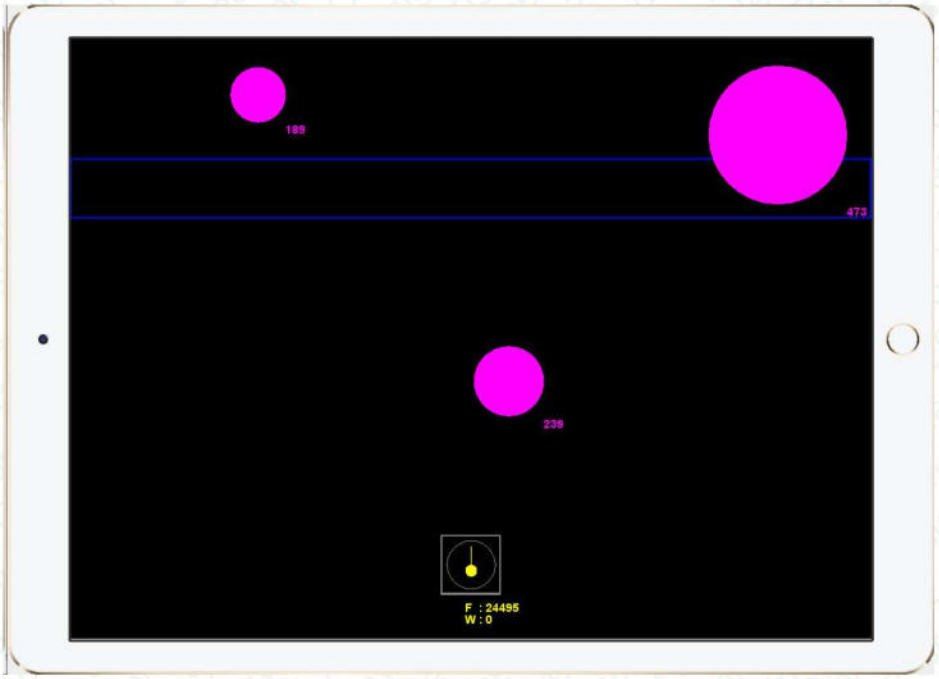
ASSIGNMENT 1 :: A FIRST VERSION OF HORNETS HFD



INTRODUCTION

This term we will be studying object-oriented graphics programming and design. Simple video games provide a good platform for understanding these concepts and you will be developing successively more complex variants of a simple 2D top-down video game over the course of the term.

This term our game is called **Hornets HFD.** Hornets HFD is set in the near future where an explosion of brush fires is wreaking havoc in our community. Fires are spontaneously erupting all over campus and you are tasked with using your firefighting helicopters to extinguish all of the fires and save the campus. You will pick up water from the American River and use that water to fight fires throughout the campus. You will have to make repeated trips back to the river to get more water and the fires will continue to grow in size until they are fully extinguished. As the game increases in complexity over the term you will have to avoid other objects and manage your fuel and other resources carefully in order to win the game.

The goal of this first assignment is to develop a simple but fully playable version of the game and to introduce you to programming in Codename One. You will be building on this throughout the semester and refactoring your code continuously. To that end, we are not hyper concerned about getting everything right the first time around. It's far more important that you get it working and then work on improving your working solution by continuously refactoring to cleaner code as we progress through the term.

For this version we will use the keyboard to control a single helicopter in a simplified graphical display that uses the entire screen. In later versions we will add additional screen components and controls as we learn how to build more complex GUI interfaces.

HIGH LEVEL PROGRAM STRUCTURE

We're going to start out with a basic structure that we will build on over the course of the term. Some of this structure will remain in place throughout the project, other parts of it you will change and adapt. Don't be afraid of this, refactoring is an important part of good coding.

Pay close attention to the structure description below. If class names are specified specifically in this document, then you are required to use those exact class names in your project.

There are three hierarchical components that form the primary structure of the project.

Class AppMain

At the highest level we have the class **AppMain**. This class is generated for you by the Codename One maven plugin and provided in the starter zip. You are going to basically change one line in this class for your game. The purpose of this class is to manage the high-level aspects of our application and setup and show the initial Form for your application. In the case of our game, this is nothing more than instantiating a new **Game** object.

Class Game

The **Game** class encapsulates the idea of the game and manages the user input and display output. Your Game class must extend the form class so that it provides a canvas on which we can display our graphics. In addition, it must implement the runnable interface so that an interval timer can invoke a *run()* method that will call methods in our **GameWorld** class.

We are implementing a simple variant of MVC, or, Model View Controller. The Game class serves as the controller because it handles the flow of input controls and dispatches these commands to the **GameWorld** class, which, as we will discuss in more detail later, is our model. In MVC the model contains the rules that govern the application. This is often referred to as the *business logic* of the application. In this first version of the game our Game class also serves as the View as it provides the basic canvas for rendering our game. We'll talk more about the MVC pattern later in the course.

Class GameWorld

All of the rules in our game are implemented in the GameWorld. This class holds the state of the game and determines win/lose conditions and instantiates and links the other *Game Objects.* The GameWorld does not know anything about where user input comes from or how it is generated.

At this stage we are not overly concerned that we are purely and properly implementing MVC. We do, however, want to start thinking about *separation of concerns*. We should be able to change our input from keyboard to touch, or swap in a completely different output view type, i.e., a textural view, without touching much code in our game model.

The interaction of these classes is discussed further later in the document.

GAME OBJECT CLASSES

In addition to the classes described above you will have some additional classes that represent game objects. For this version of the code each game object will be its own base class, i.e., derived only from **Object**. In future versions of this project, you will build a hierarchy of game objects. The reason that we are doing it this way is so that you get some specific practice in refactoring and can see a concrete example of the tradeoffs between the complexity of inheritance hierarchies as opposed to the simplicity of a flat object structure.

You will need each of the following game object classes. All of these game objects must have a location on the screen. For this project use the **com.codename1.ui.geom.Point** class to represent this location. Consult the CN1 developer's guide for detail on this class. You will want to take heed that this will be a key aspect of this course. Developers must be able to learn new APIs on their own. Each game object class must also have a *draw(Graphics g)* method that will draw the object on the provided graphics context *g*. We will discuss this further later. Each class will have additional methods that implement behavior for that class.

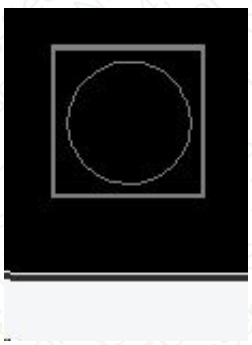
Later in this document I will discuss the basics of object behaviors and private data, but for now, let's jump into the various classes that will represent the game objects.

Class River

This class represents the American River. For this first version of the project, we will abstract the river as a simple open blue rectangle approximately one third from the top of the game window. You should derive the width of the river from the height of the game window so that it is a reasonable width no matter which device and/or orientation is selected to play the game. The length of the river extends across the entire width of the game window. An important aspect to think about with the **River** object is that a helicopter must be above the river in order to collect water from the river. For this *behavior* you will want to think carefully about the coupling between the River object and the Helicopter object. There is more detail on this further down in this document.

Class Helipad

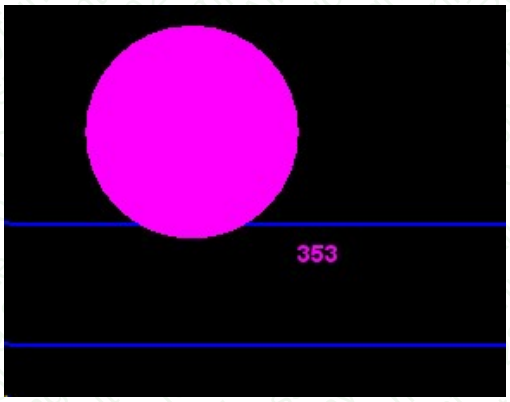
This class represents the starting and endling location of this first game. The helicopter will take off from the helipad and after putting out all of the fires will have to land back on the helipad in order to end the game. For this game there is no actual notion of altitude. A helicopter is landed on the helipad when it is within the bounds of the helipad and not moving.



The helipad is represented on screen by a gray square with a gray circle centered within the square. There should be a gap between the circle's edge and the square edge. The exact relationship is not strict and you may adjust for taste. However, the circle must be centered and there must be a clear and visible gap between the circle and the square. This is so that you get some practice in working with the coordinate system. It must be obvious that there is a difference and that you correctly computed the centering of the circle within the square for your submission to meet this requirement. The border of the square must have a thickness of five pixels.

The helipad should be centered along the width of the screen and should be roughly one half of its width above the bottom edge of the screen. Feel free to adjust slightly to make sure that your Helicopter fuel and water readout is clearly visible on startup.

Class Fire



This class represents a single fire burning on the abstract campus. For now, there is no relationship to buildings or other objects. The fire is represented on the screen by a magenta circle. For this version of the game, you will create three fire objects. Two above the river, one on the right and one of the left, and one below the river roughly centered vertically and horizontally in the space below the river. Each of your fire objects must have some slight random variation in both size and position when placed so that each time you play the game both the position and size are slightly different. It is your job to manage this variation so that the game is fun and playable. This will be a recurring theme throughout this course and it should be among your first lessons in the following idea:

Just because the code works does not mean that the project is complete.

In addition to displaying a circle, each fire circle must display its width to the lower right of the circle. The reason for this is in case the fire circle gets very small; the non-zero width will be immediately visible to the player.

As long as the width is greater than zero, the fire is still burning. Interactions between the fire objects and the helicopter are discussed later in the document. However, you will need to provide methods to grow the fire, and to fight the fire with some amount of water. As long as the fire is still burning it will grow over time. To reduce the size of the fire you need to dump some water on it and the amount of reduction is a function of how much water is dropped on the fire.

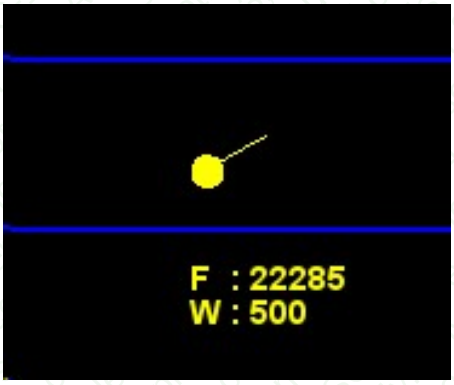
When you place the fire, its location is specified by the upper left-hand corner of the *bounding rectangle* of the circle. This means that the location of the circle is not at the center of the circle. When the fire grows or shrinks, you must maintain the original center of the fire. This will require you to think about CN1 graphics coordinates while growing and shrinking the fire.

You will need to store the fire objects in a Java Collection object so that the fires can be iterated over at different points in the game. You must do this! While you are allowed to repeat the constructor calls for the three different fire objects for the convenience in placing them in their required locations, this is the only time in the game where you may do this. At all other points you must treat each fire as simply one of the fires in the collection of Fire objects. It is a requirement that discrete Fire objects cannot be class level fields. They can only be local method variables in your *init()* method in GameWorld.

Later in the course we will use a more robust technique to instantiate the locations of our fires and we may not need to instantiate them in the same way within our *init()* method in order to place the fires at their desired location.

Class Helicopter

This class is the most complex game object and represents the main *player character* of this version of the game. The helicopter is represented as a small filled yellow circle with a line emanating from the center of the circle pointing in the direction of the helicopter's heading.



In this project the heading of an object is the compass heading specified in degrees. When the helicopter is initially place on the map it is placed facing a heading of zero degrees, or, due north, and an integer speed of zero. There are some complications relating to compass heading that are discussed later in this document. As the heading changes the line must rotate to point in the direction of the new heading. As with the helipad, you should derive the size of the helicopter object from the dimensions of the screen. More on this later. As long as it looks reasonably similar to the drawing and behaves as described herein, you will be fine.

Below the helicopter you must display the current fuel and water capacity, as shown above. These move with the helicopter but remain in the same position relative to the helicopter body. The Helicopter is initially centered on the Helipad so it is a good idea to pass the necessary coordinates into the Helicopter's constructor. You must pass in the center of the helipad as opposed to the helipad's coordinates. You may want to think about the order in which you create the objects to avoid issues here. Note, you cannot derive the coordinates of the helipad based on its placement rules for this game. In other words, if the location of the helipad changes, no changes to the helicopter code should have to be made. You are allowed to compute the adjustments necessary to center the helicopter on the helipad based on the center of the helipad that is passed in. If this doesn't make sense at the moment, then just go ahead and use those coordinates that are passed in and you should be able to immediately see what needs to be done. This leads us to some additional advice that you would do well to heed throughout this course:

Don't be afraid to experiment and try things out. More importantly, give yourself enough time for this experimentation as it is an important part of the discovery and learning process.

The water and fuel properties of the helicopter are integer values. The initial fuel value is set for playability and must be specified in the GameWorld class. The water is initially zero and is increased when the player instructs the helicopter to drink from the river. The details of this are discussed further below. The helicopter also has an integer speed that is also initially set to zero. The helicopter speed increases and decreases with break and acceleration commands, but there is a maximum speed of 10. The helicopter will ignore requests to go faster than the maximum speed or slower than zero.

We will discuss all of these objects further in a later section on game behavior. For now, let's take a look at all of the commands that we need to play this first version of the game.

GAME COMMANDS

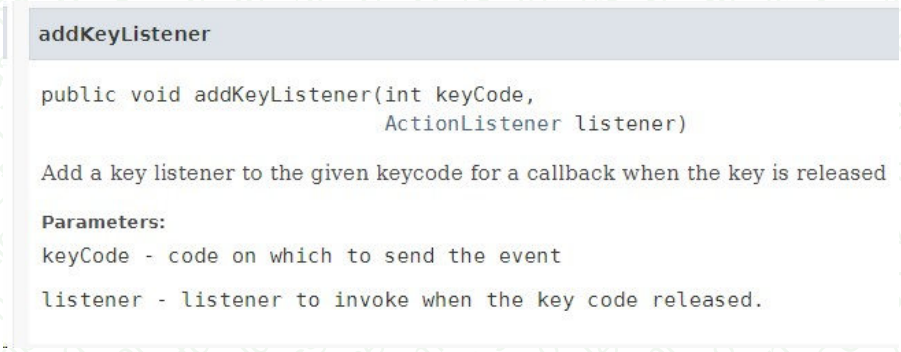
For this version of the game all of the game input is via keyboard commands. Later on in the course we will talk in detail about things like command objects and event driven operation. For now, we are just going to use these features as given below. I will give you a basic explanation of the behavior and some boiler plate code to implement the commands.

The Commands

Left Arrow	(-93)	Changes the heading of the helicopter by 15 degrees to the left.
Right Arrow	(-94)	Changes the heading of the helicopter by 15 degrees to the right.
Up Arrow	(-91)	Increases the speed of the helicopter by one.
Down Arrow	(-92)	Decreases the speed of the helicopter by one.
'f'		Attempts to fight a fire by dumping water on the map.
'd'		Drinks water when over a water source.
'Q'		Quits the game. (Note uppercase)

If you have never built GUI based Java software before then you might be used the idea that you need to *poll* the keyboard in order to see if any keys have been pressed and then, if so, determine which key had been pressed and then act on that keypress. You might use the Scanner class along with a case statement or any other of the various ways that keyboard input can be obtained in Java.

With modern GUI applications we rely on what is known as *event driven programming.* In event driven programming the GUI framework has a mechanism to register a listener for keypress events with respect to a particular GUI component. In our case, it is our main Form, or our Game class.



addKeyListener

```
public void addKeyListener(int keyCode,
                           ActionListener listener)
```

Add a key listener to the given keycode for a callback when the key is released

Parameters:

keyCode - code on which to send the event

listener - listener to invoke when the key code released.

Figure 1. addKeyListener method definition from the CN1 class reference.

Take a look at the addKeyListener() definition from the Codename One class reference for the Form class. This method takes a key code as well as a *callback* object of type ActionListener that will be invoked whenever that key is pressed while our main form is active. It seems a bit complicated to define an object just to link a keypress with a method. We're going to talk much more about what an ActionListener object is later, but, for now, we are going to take advantage of some *syntactic sugar* that has been a part of the Java language since version 8.

For each command in our game, we want to let the form know that it must respond to that keypress by calling a specific method within our game. Since the *game mechanics* are all handled by the GameWorld class, our listeners will all invoke methods in our GameWorld object.

Let's start by taking a look at what this will look like in your GameWorld class.

```
class Game extends Form implements Runnable {
    private GameWorld gw;

    public Game() {
        gw = new GameWorld();

        addKeyListener( keyCode: 'Q', (evt) -> gw.quit());
```

Take a look at the call to addKeyListener. The first parameter is the key code, this is an integer, such as the integers in the table next to the arrow keys, but we can also use a specific (single) quoted character. So, whenever the user presses shift Q, the ActionEvent that is the second parameter will do its thing. But what exactly is that syntax? This is what is known as a Lambda in Java 8. It represents an object and the addKeyListener line above Is equivalent to the following, slightly more complex, code. Note that we aren't doing anything here with the ActionEvent evt, but it is necessary in the syntax above because the ActionListener defines a single method that takes exactly one parameter. So, in our shortcut syntax, we must provide that parameter even though we ignore it.

```
addKeyListener( keyCode: 'Q',
    new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            gw.quit();
        }
    });
```

For the moment we don't need to worry at all about this more complete syntax, we will revisit it later. Our commands will all take the same form, an integer keycode will invoke a parameterless method in our GameWorld class. So, for each of the commands on the previous page, you will want to define a method in your GameWorld class to perform the appropriate action whenever that key is pressed. Your quit() method, defined in your GameWorld class, will look like this:.

```
public void quit() {
    Display.getInstance().exitApplication();
}
```

I'm showing you this because it points out an important aspect of programming in CN1 that is different from standard Java. Notice that the quit() method does not invoke the standard System.exit() method that you may be used to calling. This is because CN1 needs to exit gracefully and shut down all of the resources associated with our application. The remainder of your commands should invoke an appropriately named method in GameWorld that modifies the game state in response to the command.

GAME MECHANICS

You should have watched the basic game play video to get a basic idea of the gameplay for this version of the game. Note, we will be changing both the game play as well as the win/lose conditions throughout the term. At each stage the gameplay will reflect the learning goals for the module.

Game Play Overview

The game begins with the helicopter stopped and resting on the pad. For this version of the game the engine is running and so the helicopter is consuming some fuel as soon as the game begins. The player will use the navigation keys to move the helicopter towards the river. At the river the helicopter will slow down and take on (drink = 'd') water from the river. The helicopter will then navigate towards one of the fires. When the helicopter is over the fire the player will drop the water to fight = 'f' the fire. The player will then fly back to the river to pick up more water and continue to fight all fires until they are put out. At this point the player must return to the landing pad and bring the speed of the helicopter back to zero. Once this happens the game will end and a dialog box will appear to report the player's score which is defined as the remaining fuel, and to give the player an opportunity to play again. If at any time during the mission the player runs out of fuel then the game is over and a dialog box will appear letting the player know that they have lost the game and, again, giving them an opportunity to play the game again.

River Mechanics

The river behavior is simple. It can supply an infinite amount of water to helicopters. As long as the helicopter is over the river, and the helicopter's speed is less than or equal to 2, then the helicopter will be able to drink water in units of 100 at a time up to a maximum amount of 1000. That is, each press of the 'd' key will increase the water on board the helicopter by 100 units but only up to 1000 units. After 1000 units, further presses of the 'd' key will have no effect. For the purposes of this assignment, the helicopter is "over the water" as long as the center of the helicopter is between the upper and lower boundaries of the river.

Fire Mechanics

The fires have several properties that make them a bit more complex than the river. Each fire will grow over time by some small amount. You want this amount to be large enough to add challenge to the game, but not so large that it makes the game unplayable. You should implement a grow() method in your fire object that gets called randomly as the game is being played. As the fire grows it must remain centered on the same point. You will have to think about this carefully when you implement your grow() method. Hint: think about this in terms of what is the smallest integer amount that you can increase the fire in the horizontal and vertical directions and implement this change in your grow method. You may then call this method more or less frequently to achieve the desired playability.

The fire is also reduced in size when the player helicopter is over the fire and *fights* the fire by pressing the 'f' key. You will want to achieve this with a shrink(int water) method that reduces the size of the fire based on how much water is dumped on the fire. See the section below on helicopter mechanics for more detail on this. Again, you must maintain the center of the fire so you will want to think carefully about how to reduce the size. It's ok if some of the water is wasted and does not go to reducing the fire.

Think about playability when choosing how much water will reduce the fire by one unit. The player should have to return several times to extinguish large fires.

For the purposes of this assignment, the player helicopter is over the fire whenever the center of the helicopter is inside the circle represented by the fire. One thing that you may want to think about is that for very small fires, it will be challenging to fly the helicopter inside the perimeter of the circle. Because of this, you may implement a method that indicates that the helicopter is *near the fire.* This method should be robust against false positives. That is, it should not be possible that a helicopter is near two small fires in this version of the game. It should also be robust against changing dimensions. That is, you should derive the notion of *nearness* from the dimensions of the device that the game is playing on. Don't over think this, but put some effort into making sure that the game plays well when fires shrink to a very small size but aren't extinguished.

There is more discussion about helicopter and fire and river interaction later when we discuss the issues of object-oriented programming with respect to this assignment.

Helicopter Mechanics

The helicopter is somewhat more complex and will evolve significantly over the course of the term. For this assignment we want to focus on simple movement and firefighting mechanics. The **Helicopter** object has a number of properties in addition to the location property. The helicopter's state includes fields for heading, speed, fuel, and water. Make sure that you review the section on object-oriented programming in this assignment before you start coding setters and getters arbitrarily.

Movement

The helicopter moves forward with increasing speed as the up arrow is pressed. You should define a maximum speed that the helicopter does not exceed. For this version of the game, increase the speed by 1 for each press of the up arrow and implement a max speed of 10. Adjust all other timing factors, e.g., how often GameWorld's tick() method is called, to work with these choices. Pressing the down arrow reduces the speed to a minimum of zero. Note that this is not realistic because a helicopter can fly backwards. In later versions of the game, we will adjust this behavior somewhat. In addition to more realistic flight mechanics, we will link the speed change to the device's refresh rate so that the game plays well on both fast and slow devices.

The right and left arrow keys will adjust the *heading* of the helicopter. In this first version we will keep this very simple and simply change the heading by fifteen degrees to the left or to the right based on which key was pressed. A heading of zero degrees represents due north and you will want to make sure that your code respects this convention. It is very important to realize, however, that most of calculations that you will need to execute will expect that zero degrees lies along the X axis. Moreover, the various trigonometric functions such as sine and cosine expect the argument to be in radians, and not degrees. Finally, because we are using the standard device coordinates that causes the Y axis values to increase in a downward direction, a naïve implementation of *turnLeft()* and *turnRight()* will cause the helicopter to turn in the wrong direction. Make sure that your implementation responds to the controls as expected and accept that will be refactoring some of this code later when we modify our coordinate system to more naturally reflect the real world.

Turning or changing speed has no immediate effect on the helicopter's movement. These actions merely change the state of the helicopter by updating the helicopter's speed and heading fields. The helicopter's

position is updated when its *move()* method is invoked from the *tick()* method in the GameWorld class. This movement is reflected on the screen when the helicopter's *draw()* method is invoked from GameWorld's *draw()* method.

Fire Fighting

Fighting fires is accomplished by dropping water onto the map. When the player presses the 'f' key the helicopter will drop its entire load of water onto the map regardless of the proximity of any fire. The water drop will only fight a fire if the helicopter is flying over a large fire or is near a small fire. This should be implemented as a *fight()* method in your helicopter class. See the section on object-oriented programming for more detail.

While there is some more to discuss regarding the helicopter, it has more to do with coding structure than game mechanics so let's move on to discussing the programming structure of the game in more detail.

DETAILED PROGRAM STRUCTURE

```
1   package org.csc133.a1;
2
3   import com.codename1.system.Lifecycle;
4   import com.codename1.ui.*;
5
6   public class AppMain extends Lifecycle {
7       @Override
8       public void runApp() {
9           new Game();
10      }
11  }
12
13  class Game extends Form implements Runnable{
14
15      @Override
16      public void run() {
17          // ...
18      }
19  }
20
21  class River{
22      //...
23  }
24
25  class Fire {
26      // ...
27  }
28
29  // etc ...
30
```

Figure 2. AppMain.java basic structure

Before we start talking about the internal structure of this project, let's clarify some things about programming in Java related to common misconceptions as well as some unusual requirements for this assignment.

Just One File Please!

While this will change with the very next assignment, for this assignment, I want you to put all of your code in a single Java file called **AppMain.java**. It is a common misconception that each Java source file can contain only one class. This has never been true. It used to be the case that there could be only one **public** class in each file, but even this requirement doesn't hold anymore under certain conditions. Nonetheless, we will adhere to that for this project.

1) Use the supplied AppMain.java file from the starter package

2) Create each of the classes in this document by typing their definitions in the same file below the AppMain class as shown in the figure. Note that only class AppMain is public.

There are several reasons for doing it this way this time that will not apply later. First, I want you to have a constant sense of how much code that you're writing over time for this first assignment. My example version of this project was under 400 lines of java. I want you to keep that in mind. If you've written 2000 lines of java, you're overdoing it.

On the other hand, if you haven't written 100 lines before we get to the third week, you are very far behind. Second, adding more files adds cognitive load that is more easily managed once you have the structure of the program in your mind. I often start small projects this way and it can save you time at first. Part of the reason for this is that deleting classes that you don't want doesn't involve removing files. Finally, you may be doing peer review of the work of others and this will help to speed that up for you and your class mates.

AppMain Structure

Class AppMain is usually created either by the CN1 plugin or the CN1 Maven plugin when you create a new CN1 project. For this project it is given to you in the starter bundle. You can easily create your own starter projects using the CN1 online project template creator or by using a Maven archetype from the command line. AppMain is going to load your main form which, for this class, is your **Game** class. It does nothing else for this project and so you may directly copy the start() method shown in the previous figure. If we had other elements of the application, e.g., monetization via ads, that was a part of the application but not directly related to the game code, we would want to initialize those classes and include some logic here.

Game Structure

The **Game** class inherits from CN1's **Form** and implements the *runnable* interface. Inheriting from Form gives our game many properties including the ability to listen to keyboard events as well as giving us a canvas on which we can draw our game objects. Most of what you will need in this first project for class **Game** is shown in Figure 2. The Game class holds a reference to a **GameWorld** object (line 9) so that it can dispatch messages (function calls) to our game world. Our constructor initializes our game world and then initializes the key listeners that will respond to commands and initiate action. Only one of these is shown for you here, you will have to enter the remainder of them. Think about this and try to see how this isolates how the player interacts with the game, e.g., keyboard commands or touch input, with the action that happens in our game world, e.g., moving a player sprite or ending the game. We'll look more closely at MVC, or model/view/controller, later, but this aspect is a part of the controller.

This is followed by the code to setup a **UITimer** which will discuss at more length when we talk about event driven interaction. For now, accept what this does and we will discuss why this is important and some of the more detailed and nuanced ways that we can accomplish these goals later. The UITimer declaration, shown on line 23, requires a ***Runnable*** class as a parameter. A UITimer setup on a ***Runnable*** class invokes that class's run() method after some elapsed time specified as the first parameter to UITimer.schedule() as a number of milliseconds. The second parameter to schedule() determines whether the call will be repeated. In this case, it will. So a UITimer will invoke the run() method every 100 milliseconds. The third parameter to schedule() is the Form that the timer is bound to. Don't worry about that detail for now. Just note that the object passed to both the UITimer constructor and to schedule() is the current Game object as identified by the ***this*** parameter.

Take a look at the run() method on line 36. This method does only two things. First, it calls tick() in our GameWorld. This means that every 100 milliseconds our GameWorld is given the opportunity to update its state. It moves objects around the map and determines the nature of any interactions between them. It also calls repaint(). The repaint() method *asks* the system to redraw the screen by invoking the paint method.

```
1  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2  // Game
3  //  Game extends Form and provides the high level interface for dealing with the
4  //  game's UI. For now, it is the view in MVC.
5  //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6
7  class Game extends Form implements Runnable {
8
9      private GameWorld gw;
10
11     final static int DISP_W = Display.getInstance().getDisplayWidth();
12     final static int DISP_H = Display.getInstance().getDisplayHeight();
13
14     public static int getSmallDim() { return Math.min(DISP_W,DISP_H); }
15     public static int getLargeDim() { return Math.max(DISP_W,DISP_H); }
16
17     public Game() {
18         gw = new GameWorld();
19
20         addKeyListener('Q', (evt) -> gw.quit());
21         // other key listeners
22
23         UITimer timer = new UITimer(this);
24         timer.schedule(100, true, this);
25
26         this.getAllStyles().setBgColor(ColorUtil.BLACK);
27         this.show();
28     }
29
30     public void paint(Graphics g) {
31         super.paint(g);
32         gw.draw(g);
33     }
34
35     @Override
36     public void run() {
37         gw.tick();
38         repaint();
39     }
40 }
```

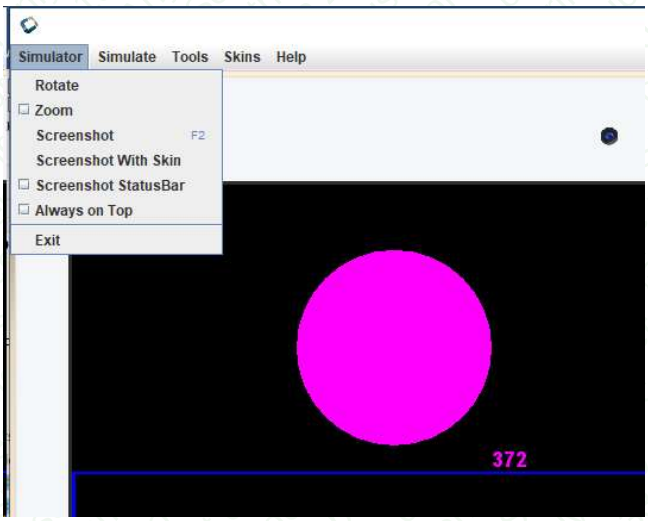
We generally don't directly invoke the paint() method in CN1 programing, and for that matter, in most of these types of GUI frameworks. We ask the framework to invoke the paint() method by calling repaint(). We'll discuss this more later. For now, know that attempting to call the paint() method on a form or container directly can result in unpredictable app behavior. In particular, we need access to a **Graphics** context and this framework is what grants us access to this through a Graphics object that is passed to us through a *callback*. A callback method is a method within our code that framework will call. We don't start CN1 apps with a public static void main method. We start the framework giving it our application as a parameter, and the framework calls methods within our application. This is something called the Dependency Inversion Principle in action. Sometimes the dependency inversion principle is referred to as the Hollywood principle, i.e., "don't call us, we'll call you.

This takes us to the paint(Graphics g) method starting on line 30. The system gives us a **Graphics** object, **g**. We should never save this and try to reuse it outside of the direct call tree of paint(). In other words, it's ok to call super.paint(Graphics g), in fact we should generally do that first so that the system can repaint other elements

of the form. We can also call our child draw() methods from within our paint method passing in the current Graphics object g. For this project, we will ask our GameWorld to draw all of its objects on the Graphics context g by invoking gw.draw(g). You will have to implement this method in your GameWorld class.

Because our Form provides the canvas on which we draw our game objects, it is also the View in MVC.

Screen Size



We are going to use the full width and height of the screen for this version of the game. While the simulator can switch between many different devices, you should start with the ***iPadAir2*** skin so that you have a large screen to work with. That said, you are required to use the screen size to adjust the size and positioning of your on-screen game objects. To keep things simple in this first version, you do not have to adjust the size and positioning while the game is playing. That is, you may treat the current size and orientation as constants in your game while it is playing. You must, however, make sure that all of the game objects are positioned correctly on startup when the game first starts no matter which orientation (portrait or landscape) is chosen. We will see later that this is not the way things work in mobile apps in the real world and that we need to be able to adjust dynamically to orientation changes.

You can change the orientation of the device by choosing Simulator/Rotate from the simulator menu. When you switch orientation, your game elements may not be visible as their placement has not changed. When you restart your game, however, the game elements should be placed correctly with respect to the new orientation.

Constants and Static Methods

You might have noticed lines 11 through 14. These lines specify some constants and some static methods. You should have already been exposed to these ideas but let's review briefly. The two constants DISP_W and DISP_H provide the full width and height of the display of the simulator. The keyword **final** means that, once initialized, these values cannot change. Hence, they are constants. The keyword **static** means that there is only one instance that is associated with the class, and not with instance objects of that class. You can access these constants in your code by referring to Game.DISP_W or Game.DISP_H. In future versions of this game, we will see that it's not really a good idea to make these constant values, however, it's fine for now. More importantly, this example demonstrates some of the standards for defining constant, e.g., all uppercase identifiers that employ *snake case.*

The two methods on line 14 and 15 are convenience methods for obtaining the current smaller and larger dimensions. These are not necessary but you may find them helpful. That said, there is an important aspect here that you will need to pay more attention to in upcoming versions of the game. We are using the min and max functions in java's Math package. When we are building mobile applications to distribute, however, we

must pay attention to the differences between what is available to the simulator and what is available to the runtime environment in CN1. As we will see later, you have to use special versions of some of the math functions if you want to build your application for distribution as the Math package is not included in CN1's runtime.

Screen Origin and Coordinate System

The origin, (0,0), of the screen is located in the upper left-hand corner with positive values increasing to the right and down. The lower right-hand corner is thus (DISP_W - 1, DISP_H - 1). We will see later that this is inconvenient and we will learn ways to impose a more natural coordinate system. For this first version of the game, we will be working directly with integer coordinates in order to keep things simple. For each game object, the position of the object is the upper left-hand corner. This corresponds directly to how the graphics primitives in CN1 are referenced. For example, if you draw a rectangle then you need to supply an X and Y for the location as well as a width and a height. The X and Y will refer to the upper left-hand corner. Note that for the interaction of some objects you will have to compute the center of the object so that the objects are placed in the correct location with respect to some other objects. For example, the helicopter must initially be placed centrally with respect to the landing pad.

GameWorld Structure

The GameWorld class provides the model in our MVC pattern. It manages the changing state of our game as we interact with it. Your GameWorld must declare and initialize all other game objects, manage the initialization of the game, determine when the game is won or lost, and draw all of the objects in the game world by invoking their draw() methods. Your GameWorld must have an init() method that is distinct from the constructor. The init() method is invoked whenever a new game must be played. It is perfectly reasonable to simply recreate all of the game objects in this init() method. The init() method creates all of the new state of the world including the positioning of each of the game objects.

As we have seen from the Game class structure, GameWorld has a tick() method that is called to update the state of the game. In this method you need to move your helicopter and check the status of each of the fires. If the fires are not put out, then they will slowly grow over time. It's important to keep in mind what information each class can access.

If the game as reached either a win or lose condition, the tick method should invoke a *modal dialog box* as shown in the demo video to offer to quit or replay. Look at the Dialog class in the CN1 class reference to see

how to implement this. If the player wins, their score is their remaining fuel and you will want to display that in the body of the dialog box. Otherwise, you can tell them that they ran out of fuel. You may change the language of the dialog box to suit your personality, however, you must include an option to exit the application as well as an option to play again.

You may create helper methods as necessary but they should not be public unless required. Later we may move these methods to a separate class, for this game, you should not create any additional classes other than what is specified in this document.

Your draw() method will need to draw all of the game objects as well as the black background. For this game, we are going to brute force the playing surface and use the entire area of the form. Note that this is not really good practice but our goal here is to delay some of the learning details of CN1 while allowing you to get a functioning game going. To that end, your first task should be to draw a black background for the entire game. You do not need to use the special paintBackground() method, simply draw a black rectangle, or, call clearRect() on the graphics context g. You will want to spend some time looking at the methods that are available in the Graphics class in the CN1 reference so you can figure out how to draw each of the different graphics primitives, e.g., circle, square, lines, required for this game.

Game Objects Structure

While many of the details of the game objects are discussed earlier in the document, there a few additional details and constraints noted below.

Each object has color. For this version of the game this is hard coded and you do not need to be able to change the color nor do you need to make this a field variable of the object. All of the game colors for this version have been chosen so that they have a predefined constant in the **ColorUtil** class, e.g., **ColorUtil**.**BLUE**. So you do not have to think about color models or the format of the color, just pass the defined constant into the setColor() method of the Graphics object g.

The Fire objects will have a somewhat complex interaction with the rest of the game. Think carefully about defining public methods that effectively model the behavior of the fire. Is it still burning? Is it near a helicopter? Would a fire know this? What about a helicopter? What's the most natural place to put this method? What data is needed for this method to work? What is the most natural method argument?

Helicopter fuel level is initially set to an initial value of 25000 and this should be defined as a constant in your GameWorld. Fuel is consumed at the square of the speed plus 5. Thus, the helicopter consumes much more speed at the fastest speeds.

While the initial fuel level is defined in the GameWorld, constants for maximum speed and maximum water should be defined in the helicopter class. For now, we will set maximum speed at 10 and maximum water at 1000, but we may adjust these for better gameplay in the future.

Private Data and Setters and Getters

In this course, all mutable, i.e., settable, fields must be private. For example, the location of each object is settable so it must be private data. This means that you may need setters and getters to obtain the value of this data externally or to set the value of this data. You do not always need setters or getters, however, and it's important to try to think in terms of behaviors. We will have more to say about this later, however, you should not assume that it's best to just add setters and getters for each variable. This is unnecessary and is reasonably equivalent to just making the data public. Unnecessary setters and getters indicate lower quality code and may negatively impact your grade.

As an example, consider a simple game that has a character that walks around a map. This character has a location and when you initially create the character you will need to put it somewhere on the map. This doesn't mean that we need a setter for the location data, however, as we can either provide a constructor with initial location data that determines where the character will *spawn*, or, we can choose this point, either fixed or randomly, in a constructor without parameters. You may think that moving the character around the map requires a setter. But think about this. We may not want the character to just move from any position to any other position on the map and if we can arbitrarily set the location field, then a bug may cause that to happen.

Instead of a setter for our character's location, let's think about character behaviors. One behavior that our character may need to have is to walk about the map. So, instead of creating *setLocation()*, we can create *walk()*. Note that the latter method name describes a behavior in terms of the *domain* of the project. That is, it describes a behavior of a character in terms of what it means with respect to the real-world object that our class represents. Now, if we really do want to have the character jump to another location, we can implement a *teleport()* method. Both of these methods give us the ability to more precisely constrain the behavior. For example, when you walk, there may be restrictions on how far or how fast you can walk. The same with teleporting, although, the restrictions will most likely be different than walking.

Hopefully you are getting the idea, you will need to think carefully about how you modify an object's private data. You want to think, as much as possible, in terms of behaviors of the object and to minimize the *coupling* between objects. We'll have more to say about coupling, and its close cousin *cohesion*, later in the course.

Some specific examples of object interactions

While the following discussion is not intended to be complete, it will give you some idea of how to start thinking in more detail about the differences between procedural and object-oriented programming. There is more to good object-oriented programming than just binding all of our data together with the methods that operate on it in something called a class. A key thought is that object-oriented programming models objects in the real world and our knowledge and understanding of those real-world objects should be reflected naturally in our code. We should be able to read the code and, as much as possible, believe that it naturally represents the real-world objects that we are modeling.

River and Helicopter Interaction

The River will not have access to the Helicopter object which will be created in the GameWorld class. Consequently, rather than creating complex logic in the GameWorld class that asks the River and helicopter what their locations are, consider how you create a method that returns a Boolean to that indicates whether or not a Helicopter is able to take or drink water from the River. What would you name this method? Is this a property of the River, or the Helicopter? What if we changed the name to *canDrinkFrom(River r)*? This should be clear now that this method should be a helicopter method. Should we define a method in the River called *underneathHelicopter(Helicopter h)*?

> *Thinking about these methods, what do they imply about needing getters or setters in the two classes involved?*

There's not one correct answer here. Making good choices for data access takes practice and it's rare that you get it right the first time. Further, every choice leads to some compromise somewhere and had we chosen a different route, we would have had different compromises. This is known as *the tyranny of the dominant decomposition.*

Helicopter and Fire Interaction

Similarly, you will want to think carefully about the helicopter and fire interaction. How does a helicopter fight a fire? What is necessary for this to occur and what values in the various objects must change when this is a success? Should your helicopter have a fight(Fire f) method? What advantages does passing in the Fire object give you?

Helicopter and Helipad Interaction

What about the helipad? How do we know when a helicopter has landed on a helipad? What does each object need to know in order to answer this question? Should we simply compute coordinates and answer the question in our GameWorld by using standard getters and setters for the various object's locations? Or, does it make sense to define a method hasLandedAt(Helipad helipad) in our helicopter?

These kinds of questions will be a recurring theme in this course. Generally, when students first encounter more serious object-oriented programming they think in terms of procedure. As our code becomes more complex, we want to think more in terms of encapsulation and data hiding. For readability and understandability, we want to think in terms of object behaviors that are natural for the object.

There is often some conflict between the behaviors of the real-world object and that of the model of that object. Our job as programmers is to balance these conflicts as best as we can. Ideally someone who understands the real-world domain will find our code to be a readable and natural representation of the problem.

CODING STANDARDS AND GIT REQUIREMENTS

Many of the required coding standards for this project have been defined throughout this document. In addition, you must adhere to the following. Note, these are only starter guidelines and you should have learned these in your previous courses. You will be learning the basics of Clean Coding in this course and your grade will, in part, reflect the degree to which you adopt those standards.

1) Class names always start with an upper-case letter
2) Variable names always start with a lower-case letter
3) Non-Constant identifiers use camel case
4) Constant identifiers use upper snake case
5) All code is neat and properly indented
6) You are restricted to an **80-character width**
    a. I want you to break habits that you may have developed of writing very long lines of Java code. You must learn to limit your width to aid readability.
    b. Java allows you to break lines in places you might not have thought about. Learn to structure your code more vertically with carefully placed line breaks that do not change the semantics of the Java language.
    c. The reason that it's so important for this first project is that it makes peer review and grading easier. Lines longer than 80 characters will break in speed-grader making your code less readable.
    d. This is a hard requirement for this assignment and it will impact your grade!

In the clean code discussion, the authors will warn you that commenting is a code smell. I'm reasonably certain that your other instructors have told you to comment excessively. We will be focusing in this course on writing clean and self-documenting code. However, this is not always possible and, where necessary, you must communicate your intent with comments. For this first assignment I don't want you to worry too much about this. I do not want to see comments on every line, but, feel free to do your best adding comments that you think are helpful. As we move through the course, however, you will need to adapt your style and learn to see comments as an indicator that your code might not be sufficiently readable.

Meta-Comments

For this submission I definitely want you to leave meta-comments. These are communications to your instructor about what your intent and choices were. Tell me about your thought process while writing your code. This should be a natural part of your coding. If you practice this while you're coding, then you will learn to develop text that can go in documentation at a later time. For future versions of the project, you will most likely have to submit a writeup that includes much of this discussion. You definitely want to practice communicating about your design process now!

Git

You are required to use Git in this course. You only need use it from within the IntelliJ IDE and you do not need to use github or advanced features such as branching. However, you are required to regularly commit to the local Git repository in your project directory and you are required to submit your git log with your submission.

The zip file of your project directory will contain a full copy of your repo and all of its history. The purpose of this is twofold. First, it's just a good habit to make and a valuable tool to practice using. It can save your bacon by allowing you to undo project breaking mistakes easily. Second, it communicates how you are working on the project to the grading team. It will be used as a measure of your engagement with the course and is one of several ways that your engagement will be assessed. You should make regular commits while you are coding and you are expected to include meaningful commit messages. It is expected that your git log will reflect steady progress over time on each of the projects in this course.

SUBMISSION

For the full submission of this project, you will submit multiple files. Failure to submit any of the following will yield an automatic no-pass grade.

Your **AppMain.java** file. You will submit this as a .java file. Yes, this same file will be included in the zip below. Every semester someone second guesses this requirement, don't. My grader wants to see your Java file in SpeedGrader and your zip must be available for verification.

1. An MP4 video of you playing your game in the **iPadAir2 emulator skin** in **landscape** orientation. Your video file must be named **A1.mp4**. You only need to play the game once but you must win the game so that all features of the gameplay are shown in your demo. Your video is limited to five minutes in length. If you can't win in five minutes, practice until you can. The game is lost by doing nothing in about eight minutes so this should not be too difficult. You must upload the MP4 file. Links to external videos will not be graded. You may narrate the video if you wish, but it is not necessary.

   Your video file must be of reasonably high quality. You may not use your phone camera to record your screen. I expect you to use some type of screen capture software. I use OBS Studio which is free, open source, and runs on all major computing platforms and is able to capture the CN1 simulator window.

2. A screen shot of your git log for the project as taken from IntelliJ, or a text editor if you are not using IntelliJ. The screenshot must show as much of your log as possible and it must show git-hashes. See the video on setting up CN1 if this isn't clear to you. I don't want to see a text file, just take a screen shot. Please take this seriously, half-attempts on this requirement will prevent you from earning top marks.

3. A zip file of your entire project directory. Unzipping this file should create the project directory and so you should create the zip file from the parent if you are using the command line. Your zip file, when unzipped, should behave exactly as the starter zip that you downloaded. If you don't know how to do this, then now is a great time to teach yourself.

There will be a code check-in for this project at the two-week mark. For this check-in you must upload your AppMain.java, a screenshot of your application running, and a screenshot of your git log. You do not have to have your project fully working; however, you must have about one third of the project completed for a passing grade. This will be assessed by specific milestones that will be delineated in the rubric for the check-in.