Section 1: For the physics, I implemented the bare minimum, and four conventional physics classes that I put in a physics package, so that I can easily replace them with a 3rd party physics library later. This basic engine includes 2D vectors for forces and speeds, and uses a standard tick(int milliseconds) method to translate physics objects according to these vectors. I put the call to physics.tick() in Processing's draw() method and used that as my main game loop.

The most ugly part of my physics is the collision detection, which prevents an object from moving if its destination intersects with another object. This prevents the player's avatar from passing through the floor, but also means the player cannot move left or right across a surface without jumping, because the force from the left or right direction plus the force of gravity would result in a vector pointing diagonally down and to the left or right. Since that is not a legal move, no translation takes place. Only when the player is not on a surface (for example after jumping) can the player move left or right.

Section 2, DeadlockingForkExample: The first modification caused deadlock by having the two threads synchronized on a on one static Object each, and then while still holding their first Object, synchronizing on each other's Object. That is, Thread 1 obtained Object 1 and Thread 2 obtained Object 2. Then Thread 1 tried to obtain Object 2 while Thread 2 without releasing Object 1, and Thread 2 tried to obtain Object 1 without releasing Object 2.

Section 2, ThirdThreadForkExample: This modification adds a third thread and changes the busy boolean to an AtomicBoolean owned by the first Thread and accessed by the other two Threads.

Section 2, ProcessingWithNotify: In this modification I was trying to make the draw() and keyPressed() methods of a PApplet interact in the way Thread0 and Thread1 interacted in the original fork example. I was surprised when the process hung at .wait() because .notifyAll() was never called. I discovered that draw() and keyPressed() are actually called in the same thread, concluding that this main thread must check some event queue for keyPressed events, and some other thread must write the events to the buffer when the user presses a key. This means that I won't be able to change the priority of the keyPressed task and will need to keep the keyPressed() method light or else I will slow down the main game loop.

Section2, SynchronizedBlockForkExample: In this example I was experimenting hoping that if Thread 1 used synchronized on an Object when accessing it, but Thread 2 didn't used Synchronized when accessing the Object, Thread 1 would force Thread 2 to wait, but Thread 1 wouldn't cause Thread 1 to wait, essentially giving Thread 1 priority over the Object. I wanted my main game loop in part 4 to have that priority over certain client-specific threads. Unfortunately, instead there was no interaction between the threads as if neither one used Synchronized.

Section 3: The client and server send strings in the form of byte arrays back and forth, each on a single thread. The client's thread is a loop that accepts a client connection, sends a message to the client with the number of clients connected, reads the client's greeting, and then says goodbye to the client. The client has one thread that simply connects to the server, reads a message, says hello, reads the server's goodbye message, and then closes the socket.
I also created a test (tests.HW1Section3Test.java) that runs the server on one thread, then starts each of five clients on a new thread, one every 2 seconds. I have this exact same test setup for sections 4 and 5 as well[1] (Appx 1).

Section 4: I made both the client and server PApplets that send input data and world object data to each other. I made the server a PApplet so that I could first test that the world objects were showing the correct behavior on the server before worrying about how the objects were then transmitted to the

1 I wanted to make the PApplets' frames not appear on top of one another using frame.setLocation(x, y), but could not make it work.

client. Once the client establishes a connection, the player's inputs are registered using Processing's keyPressed() method, and then communicated to the server in the form of Input objects. The Input class has the Serializable marker to be sent on the client's object output stream. The input stores the user's input in the form of an enum of the possible actions the player can take (jump, move left, and move right). The Input class also has a number of properties that the server uses to process the Input, but that the client doesn't need to set beforehand. These are marked as Transient to make the object as light as possible to send.

The server accepts clients in a separate thread created in Processing's setup() method. Each time a client connects, the thread creates a ClientHandler and adds it to a ConcurrentHashMap. It also starts a new thread that solely listens to the client for incoming input and adds it to a queue in the ClientHandler.

I decided not to use a queue from the java.util.concurrent library because I didn't want anything to slow down the main game loop that accesses the queue to process the input. Concurrent queues involve overhead to protect against concurrency issues, and I didn't want to use Synchronized(queue) either because I never wanted the main game loop to wait on a client's input queue to unlock after being written to in the client listening thread[2]. With large numbers of clients, the number of threads accessing the queue becomes large and the chance of the main game loop waiting on clients to release the lock on the Synchronized(queue) is too much.

The methods addNewInput(Input) and getNewInputs() in the ClientHandler class contain the design I chose instead. The queue is implemented by giving each Input a reference to the next Input instance in the queue. The ClientHandler has pointers to the first and last Inputs in the queue. Since one thread adds to the end of the queue, and the other thread accesses the beginning of the queue (and immediately sets that reference to null to mark the queue to be overwritten), the only issue arises when both threads access the same Input at the same time, rather than the whole queue. If the main game loop accesses the last Input in the queue to find it has no next Input, it will stop processing that Input buffer, but if the client listening thread then writes a new Input to the last Input's next reference, that new Input will never be processed. However, this requires the listening thread to access the queue before the main game loop, but not finish adding the Input until just as the main game loop has arrived at the final Input, something I decided was unlikely enough.

When the game loop processes an Input, it adds the corresponding 2D Vector of force to the physics engine, and then processes X milliseconds of physics with gameEngine.physics.tick(X), where X is the amount of time since tick(X) was last called. Then the main game thread sends each client the coordinates, dimensions, and ID of each rectangle in the game world.

The client has a separate thread that solely listens to the server for Rectangles. It then stores the Rectangles by ID in a ConcurrentHashMap. The Rectangles would then be accessed by the main game loop (Processing's .draw() method) and drawn in turn. To handle access from multiple threads, I decided the overhead of using ConcurrentHashMap to store the Rectangles was acceptable for the client, because unlike the server, this overhead would not be multiplied, and also because the client didn't have too much else to handle.

Section 5: In the client's setup, in addition to the server listening thread, a thread is started that sends Inputs to the server whenever there is new Inputs to send[3]. The Inputs are shared between threads in a ConcurrentLinkedQueue. The queue structure is to reflect the structure they will be stored in on the server. However, the java.util.concurrency implementation was used for the same reasons mentioned above for the ConcurrentHashMap.

On the server side, a new client-specific thread is started in the ClientHandler constructor that

2 This is why I wanted the method of giving threads priority to work in the third modification of ForkExample

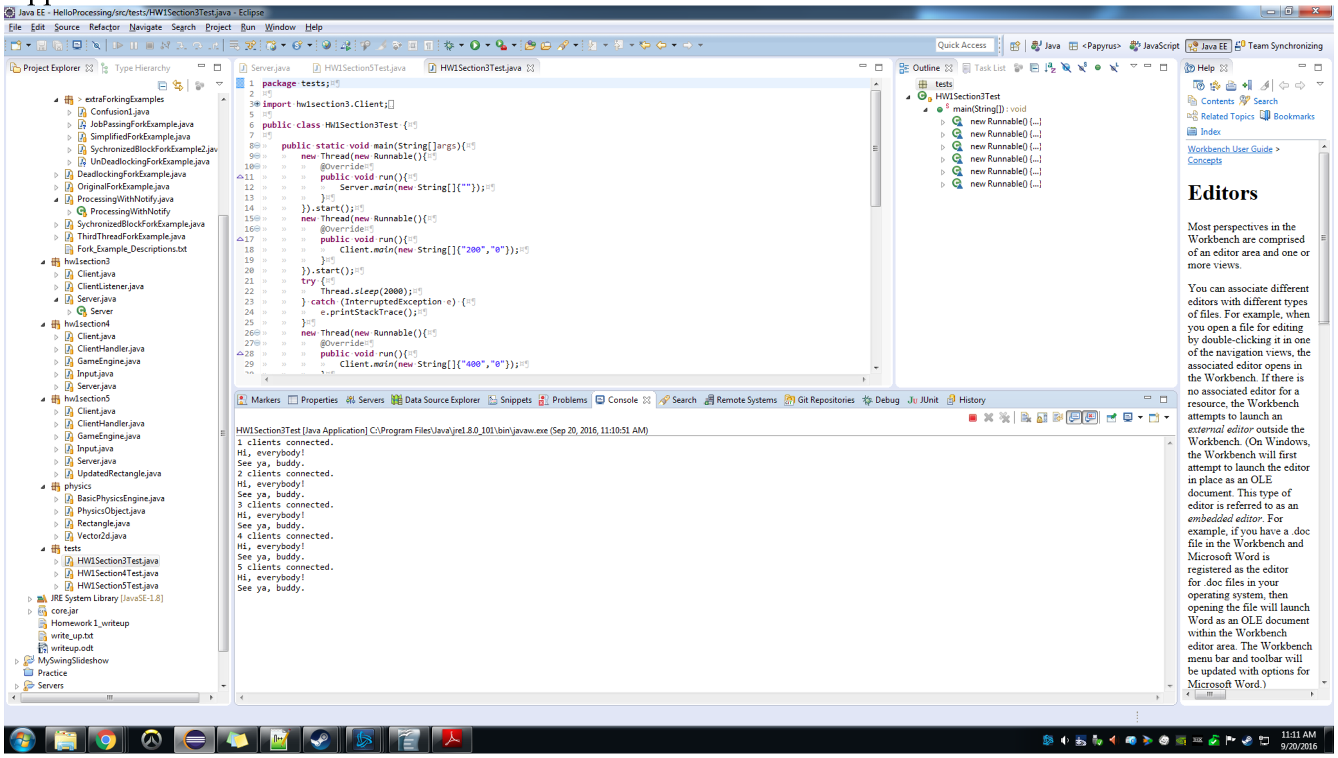
3 If not for the fact that draw() and keyPressed() are called in the same thread, I might have put the sending of data in keyPressed() for my final design.

sends Rectangle updates to that client. The Rectangles are wrapped in an UpdatedRectangle object and stored in a HashMap. The UpdatedRectangle class has a isNew boolean, which the sending thread checks so that it only sends new updates to the client. Right now all rectangles are added to the HashMap in each iteration of the game loop, but in the future, the game engine can have logic to only add updates for Rectangles that that client can see, or only Rectangles that have changed since the previous iteration.

The HashMap was used here so that the main game loop would not be slowed down by the overhead of ConcurrentHashMap or waiting for the client thread to release a lock on the data structure. The code where each thread accesses the HashMap was surrounded by a try catch, so that in the event of a concurrency exception, the only result will be that the client does not get that Rectangle's correct update for that frame.

Having the client-updating in a separate thread also prepares the task to handled by the Worker/Job design pattern in the future, so that the server can assign the priority of updating clients.

Appx-1:

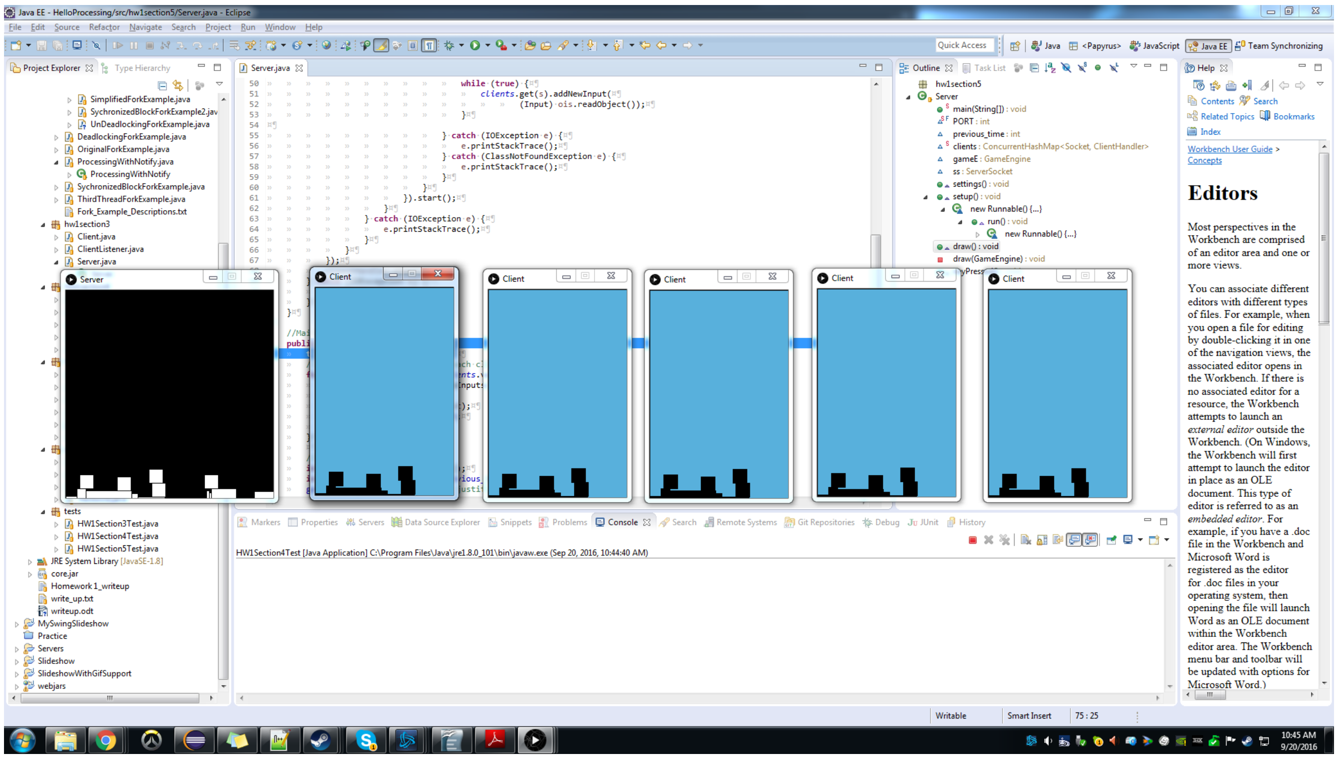


```java
package tests;

import hw1section3.Client;

public class HW1Section3Test {

	public static void main(String[]args){
		new Thread(new Runnable(){
			@Override
			public void run(){
				Server.main(new String[]{""});
			}
		}).start();
		new Thread(new Runnable(){
			@Override
			public void run(){
				Client.main(new String[]{"200","0"});
			}
		}).start();
		try {
			Thread.sleep(2000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		new Thread(new Runnable(){
			@Override
			public void run(){
				Client.main(new String[]{"400","0"});
```

```
HW1Section3Test [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Sep 20, 2016, 11:10:51 AM)
1 clients connected.
Hi, everybody!
See ya, buddy.
2 clients connected.
Hi, everybody!
See ya, buddy.
3 clients connected.
Hi, everybody!
See ya, buddy.
4 clients connected.
Hi, everybody!
See ya, buddy.
5 clients connected.
Hi, everybody!
See ya, buddy.
```

Output generated by HW1Section3Test

Appx-2:



```java
										while (true) {
											clients.get(s).addNewInput(
													(Input) ois.readObject());
										}

									} catch (IOException e) {
										e.printStackTrace();
									} catch (ClassNotFoundException e) {
										e.printStackTrace();
									}
								}
							}).start();
						}
					} catch (IOException e) {
						e.printStackTrace();
					}
				}
			});
```

```
HW1Section4Test [Java Application] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (Sep 20, 2016, 10:44:40 AM)
```

PApplets opened by HW1Section5Test