1.1

One example of a visitor pattern in real life is a waiter/waitress at a restaurant visiting each customer to get their orders, being them food, and take their payment. Laws and social convention protect the ‘data’ because if the customer does not present money or their order to the waiter, then the waiter cannot do anything to either of them. The visitor pattern circumvents this protection.

```
public interface WaiterConduct {
        public void visit(customerToVisit c);
        // never have to recompile this public class customerToVisit {
        public void acceptService (Waiter w)  {w.visit(this);}
        // package fields: can be changed dynamically by visitors
         int cash = 100;
        String order = “Chicken Parm, Soda”;
        int tip = 15;
}
public class Waiter implements WaiterConduct {
        private String order;
        public void takeOrder (customerToVisit c) {
                System.out.println("Waiter is here");
                System.out.println("Order: " + c.order);
                order = c.order;
```

```
            System.out.println("Order taken");
        }
        public void takePayment  (customerToVisit c){
            System.out.println("Waiter is here");
            System.out.println("cash before = " + c.cash);
            c.cash -= 50;
            System.out.println("cash after = " + c.cash);
        }
public class RestauruantActions {
        public static void main(String[] args) {
            customerToVisit myCustomer = new customerToVisit();
            myCustomer.acceptService(new MyWaiter());
        }
}
```

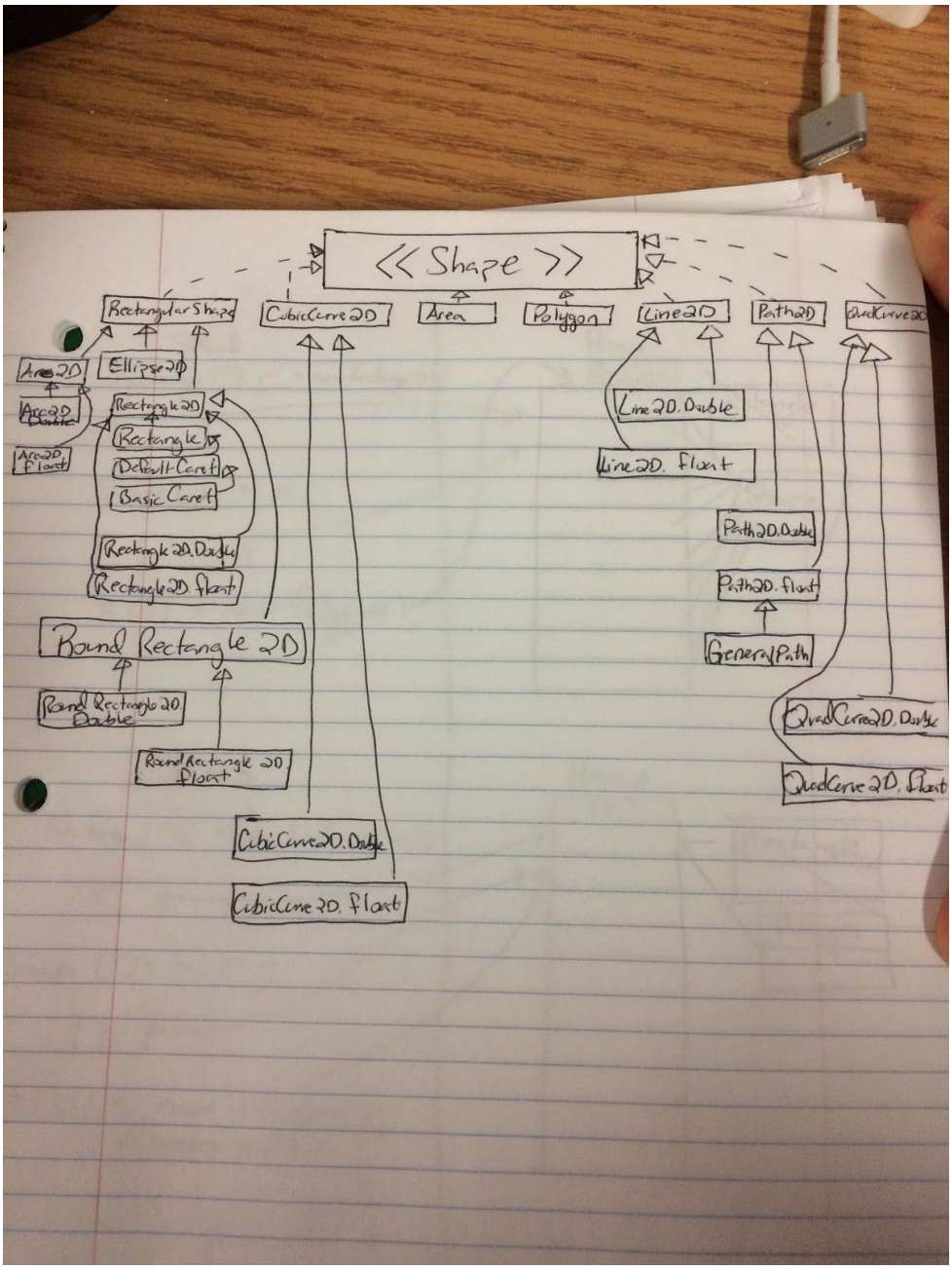
1.2

A real life example of the proxy pattern is when a customer walks into a pizza restaurant. The pizza is not immediately made, because that would be too expensive and may not be what the customer wants. Instead, the customer places an order, which acts as a proxy for the pizza object that is created, and the order allows the staff to reference the pizza directly from it. These data structures store their lengths and sizes as fields in order to allow the values to be recalled and not calculated every time they are needed.

```
a[].length //proxy of array a[]
a.size() //proxy of ArrayList a
a.length() // proxy of size of a LinkedList
```

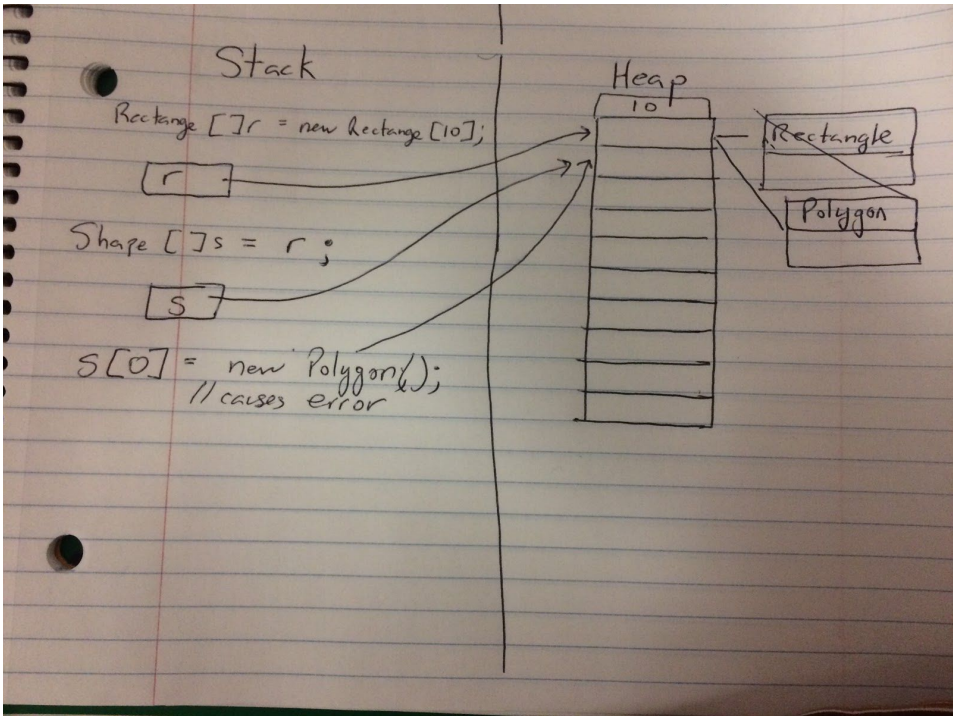
1.3

Point2D does not implement Shape because a point does not have an area and thus can not be treated as a shape. The methods contains() and getBounds() methods do not have sensible meaning for a point in space. There is no Shape3D interface because the methods that would be added to it are too similar to the ones in the Shape interface for it to make a difference.



1.4

This compiles because both Polygon and Rectangle implement Shape, so the compiler trusts the user that the code makes logical sense. The error Java.lang.ArrayStoreException is rendered.



1.5

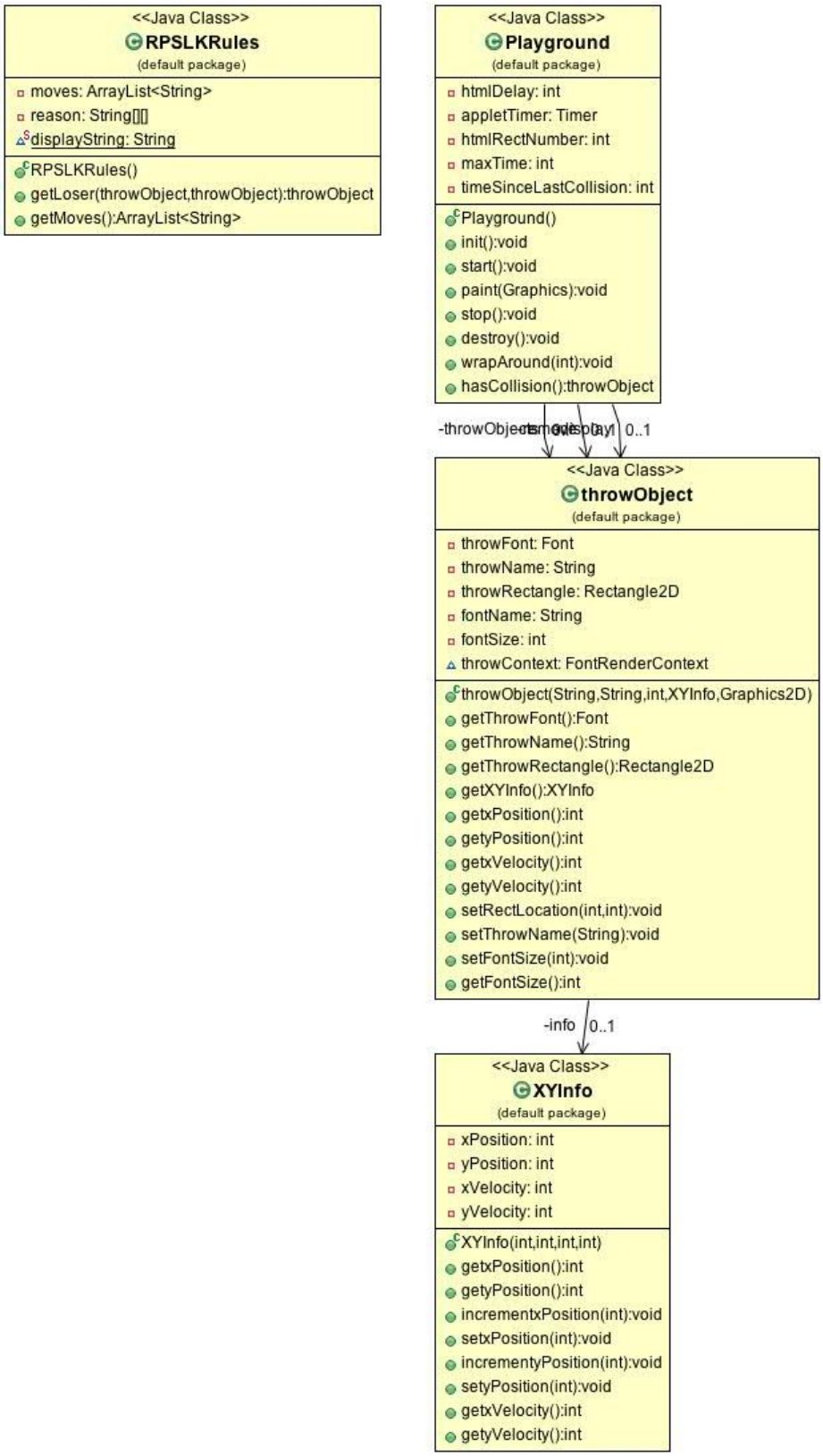
A tagging interface is one that does not list any methods, and therefore any class implementing it does not need to override any methods. They are used to define specific characteristics of classes that are used in the compiler, and give it expectations for compiling. Three examples of tagging interfaces include Remote, EventListener, and Cloneable.

- EventListener must be extended by any EventListener interface, it allows the program to respond to events.
- Cloneable is used to indicate to the clone() method that it is legal to make a field for field copy of an object in the class.
- Remote indicates that the classes methods can be invoked from a non-local virtual machine.

1.6

The collections framework has 3 fields, 55 methods, and 3966 lines. The applet source code has 593 lines and contains nine imports. A large framework is both a bad thing, and a good thing. One the negative side, it can be unwieldy and hard to implement, and often the developer will find that they are only using a small part of the framework anyway. Large frameworks put constraints on your programming and the developers themselves due to their large size and number of methods. On the positive end, developers can use a small part of the framework and ignore the rest, allowing a large amount of versatility in using the frameworks for projects.

1.7 UML


<<Java Class>>
RPSLKRules
(default package)
moves: ArrayList<String>
reason: String[][]
displayString: String
RPSLKRules()
getLoser(throwObject,throwObject):throwObject
getMoves():ArrayList<String>
<<Java Class>>
Playground
(default package)
htmlDelay: int
appletTimer: Timer
htmlRectNumber: int
maxTime: int
timeSinceLastCollision: int
Playground()
init():void
start():void
paint(Graphics):void
stop():void
destroy():void
wrapAround(int):void
hasCollision():throwObject
-throwObject
0..1
<<Java Class>>
throwObject
(default package)
throwFont: Font
throwName: String
throwRectangle: Rectangle2D
fontName: String
fontSize: int
throwContext: FontRenderContext
throwObject(String,String,int,XYInfo,Graphics2D)
getThrowFont():Font
getThrowName():String
getThrowRectangle():Rectangle2D
getXYInfo():XYInfo
getxPosition():int
getyPosition():int
getxVelocity():int
getyVelocity():int
setRectLocation(int,int):void
setThrowName(String):void
setFontSize(int):void
getFontSize():int
-info
0..1
<<Java Class>>
XYInfo
(default package)
xPosition: int
yPosition: int
xVelocity: int
yVelocity: int
XYInfo(int,int,int,int)
getxPosition():int
getyPosition():int
incrementxPosition(int):void
setxPosition(int):void
incrementyPosition(int):void
setyPosition(int):void
getxVelocity():int
getyVelocity():int