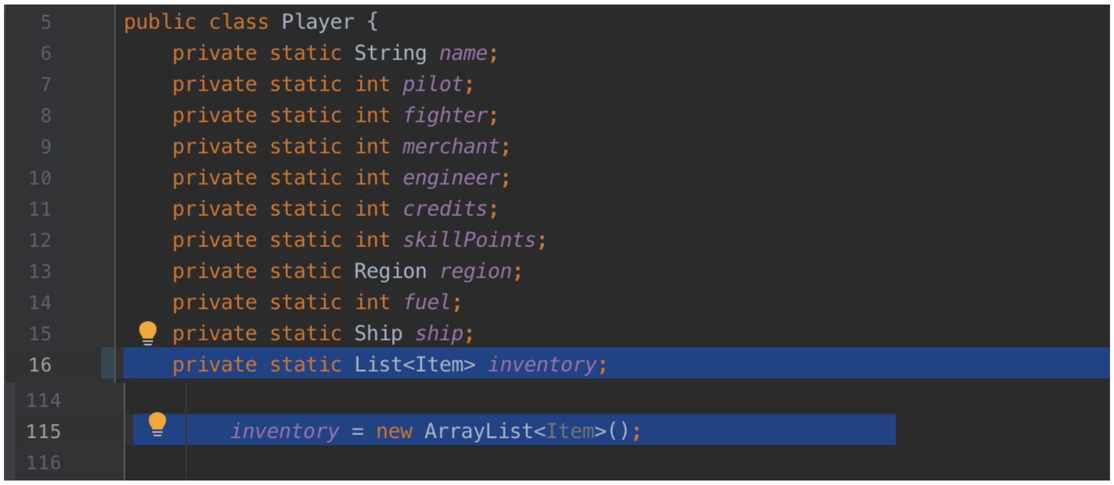
SOLID Principles

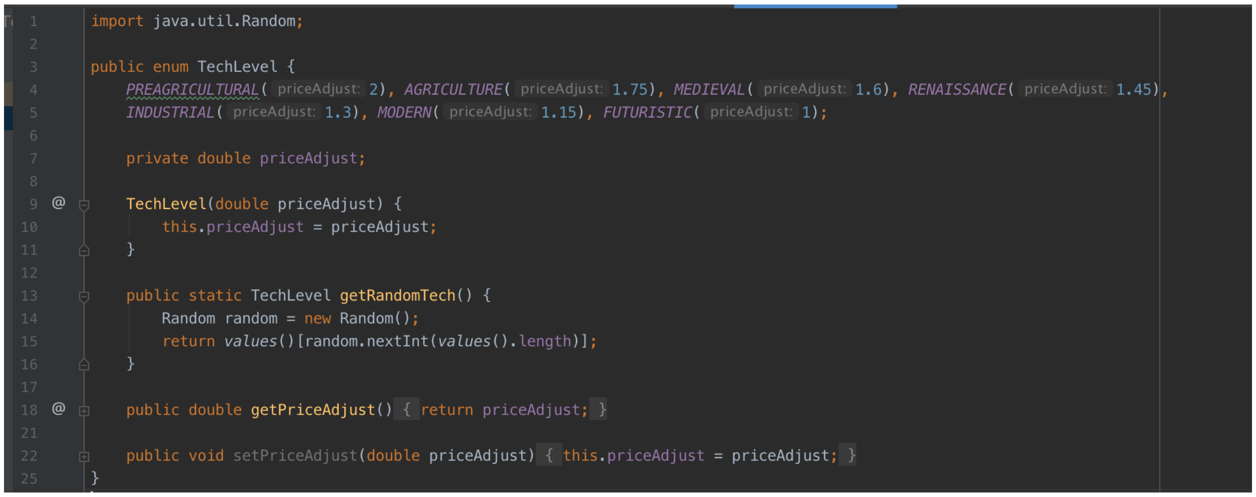
Dependency Inversion



```java
public class Player {
    private static String name;
    private static int pilot;
    private static int fighter;
    private static int merchant;
    private static int engineer;
    private static int credits;
    private static int skillPoints;
    private static Region region;
    private static int fuel;
    private static Ship ship;
    private static List<Item> inventory;

        inventory = new ArrayList<Item>();

```

Description: With the player's inventory, we have programmed to the interface, List and not the implementation of ArrayList. This demonstrates dependency inversion as we learned in class.

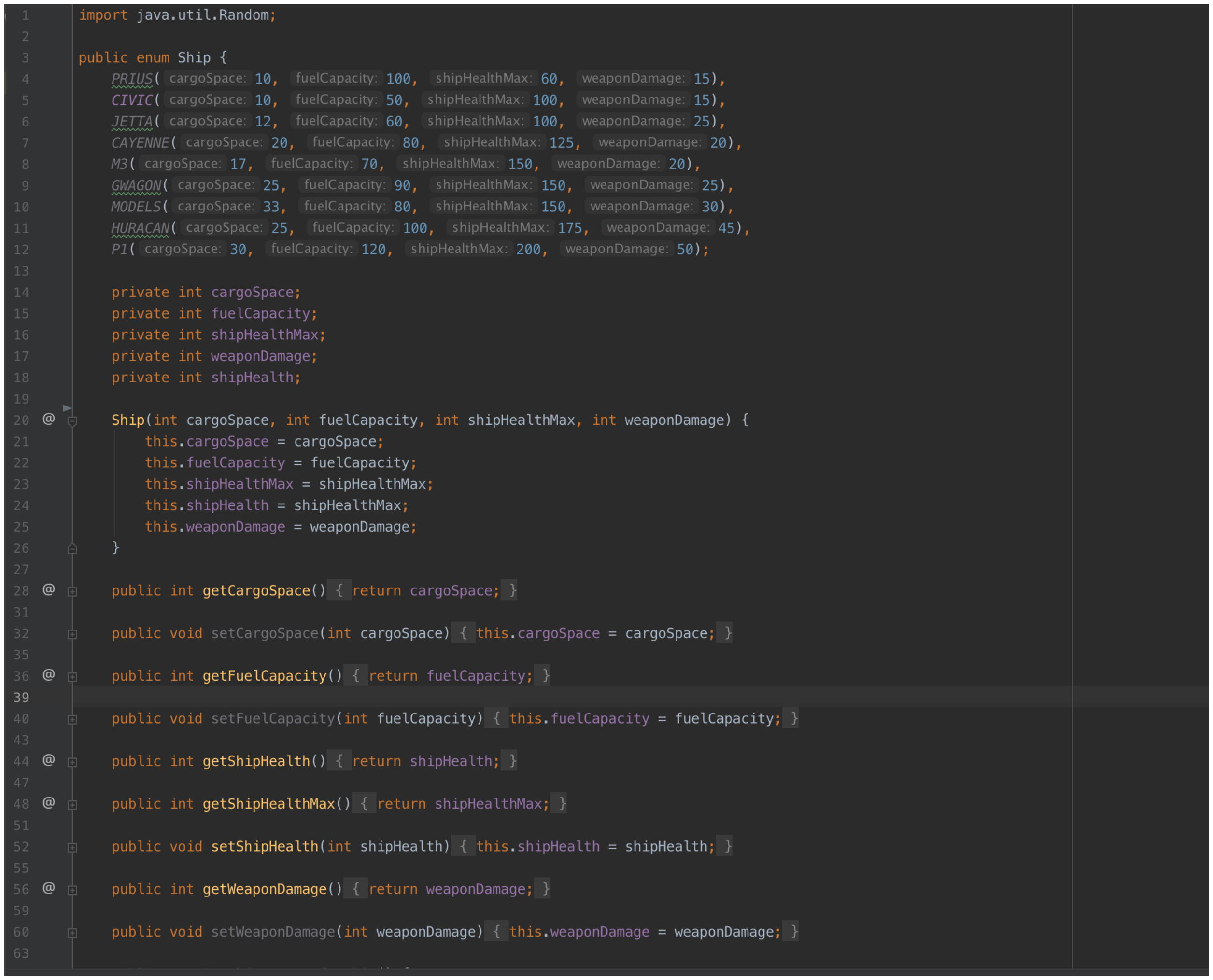
Interface Segregation



```java
import java.util.Random;

public enum TechLevel {
    PREAGRICULTURAL( priceAdjust: 2), AGRICULTURE( priceAdjust: 1.75), MEDIEVAL( priceAdjust: 1.6), RENAISSANCE( priceAdjust: 1.45),
    INDUSTRIAL( priceAdjust: 1.3), MODERN( priceAdjust: 1.15), FUTURISTIC( priceAdjust: 1);

    private double priceAdjust;

    TechLevel(double priceAdjust) {
        this.priceAdjust = priceAdjust;
    }

    public static TechLevel getRandomTech() {
        Random random = new Random();
        return values()[random.nextInt(values().length)];
    }

    public double getPriceAdjust() { return priceAdjust; }

    public void setPriceAdjust(double priceAdjust) { this.priceAdjust = priceAdjust; }
}
```

Description: This TechLevel class does not depend on other classes or unrelated methods. This class has three very simple methods that merely set a variable to adjust prices based on the tech level they are. This is a small, focused module, as opposed to being a large multipurpose module.

Single Responsibility Principle

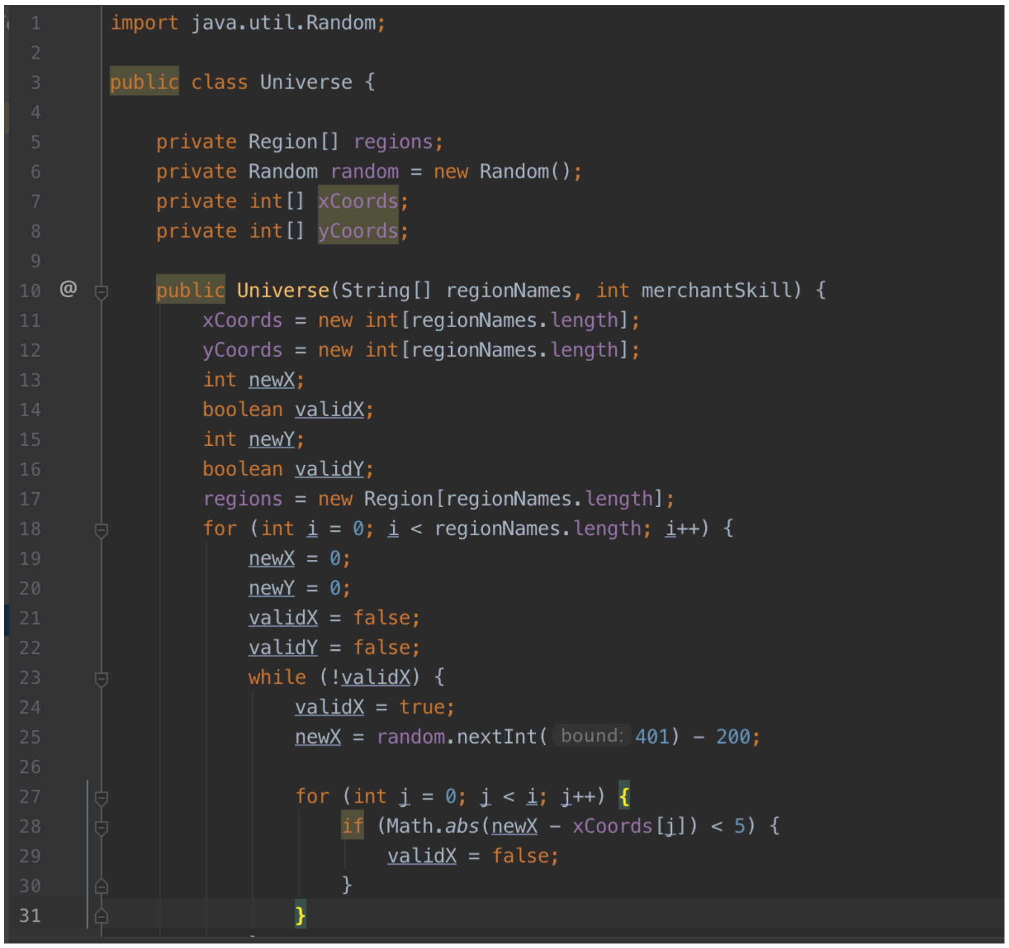


```java
import java.util.Random;

public enum Ship {
    PRIUS( cargoSpace: 10,  fuelCapacity: 100,  shipHealthMax: 60,  weaponDamage: 15),
    CIVIC( cargoSpace: 10,  fuelCapacity: 50,  shipHealthMax: 100,  weaponDamage: 15),
    JETTA( cargoSpace: 12,  fuelCapacity: 60,  shipHealthMax: 100,  weaponDamage: 25),
    CAYENNE( cargoSpace: 20,  fuelCapacity: 80,  shipHealthMax: 125,  weaponDamage: 20),
    M3( cargoSpace: 17,  fuelCapacity: 70,  shipHealthMax: 150,  weaponDamage: 20),
    GWAGON( cargoSpace: 25,  fuelCapacity: 90,  shipHealthMax: 150,  weaponDamage: 25),
    MODELS( cargoSpace: 33,  fuelCapacity: 80,  shipHealthMax: 150,  weaponDamage: 30),
    HURACAN( cargoSpace: 25,  fuelCapacity: 100,  shipHealthMax: 175,  weaponDamage: 45),
    P1( cargoSpace: 30,  fuelCapacity: 120,  shipHealthMax: 200,  weaponDamage: 50);

    private int cargoSpace;
    private int fuelCapacity;
    private int shipHealthMax;
    private int weaponDamage;
    private int shipHealth;

    Ship(int cargoSpace, int fuelCapacity, int shipHealthMax, int weaponDamage) {
        this.cargoSpace = cargoSpace;
        this.fuelCapacity = fuelCapacity;
        this.shipHealthMax = shipHealthMax;
        this.shipHealth = shipHealthMax;
        this.weaponDamage = weaponDamage;
    }

    public int getCargoSpace() { return cargoSpace; }

    public void setCargoSpace(int cargoSpace) { this.cargoSpace = cargoSpace; }

    public int getFuelCapacity() { return fuelCapacity; }

    public void setFuelCapacity(int fuelCapacity) { this.fuelCapacity = fuelCapacity; }

    public int getShipHealth() { return shipHealth; }

    public int getShipHealthMax() { return shipHealthMax; }

    public void setShipHealth(int shipHealth) { this.shipHealth = shipHealth; }

    public int getWeaponDamage() { return weaponDamage; }

    public void setWeaponDamage(int weaponDamage) { this.weaponDamage = weaponDamage; }
```

***Description*:** Ship only takes care of Ship related attributes such as cargo space, fuel capacity, ship health max, and etc. This class only contains getters and setters and does not meddle with other classes unnecessarily.

GRASP Principles

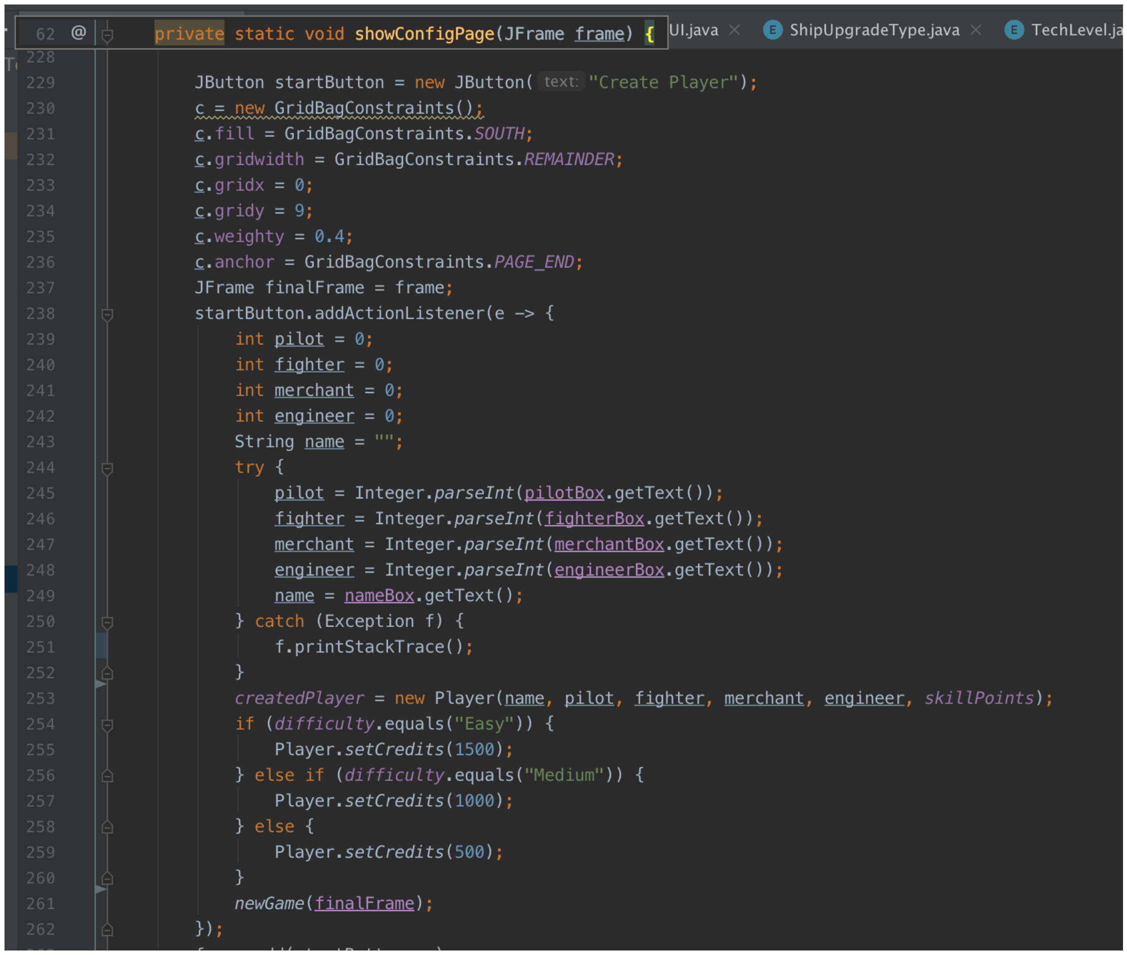
Information Expert



```java
import java.util.Random;

public class Universe {

    private Region[] regions;
    private Random random = new Random();
    private int[] xCoords;
    private int[] yCoords;

    public Universe(String[] regionNames, int merchantSkill) {
        xCoords = new int[regionNames.length];
        yCoords = new int[regionNames.length];
        int newX;
        boolean validX;
        int newY;
        boolean validY;
        regions = new Region[regionNames.length];
        for (int i = 0; i < regionNames.length; i++) {
            newX = 0;
            newY = 0;
            validX = false;
            validY = false;
            while (!validX) {
                validX = true;
                newX = random.nextInt( bound: 401) - 200;

                for (int j = 0; j < i; j++) {
                    if (Math.abs(newX - xCoords[j]) < 5) {
                        validX = false;
                    }
                }
```

Description: In order to make the Universe, this class must know the regions in order to assign x and y coordinates to add them to a single plane in this universe. Universe is the information expert on all regions in the game and therefore has the responsibility of organizing the regions.

Creator

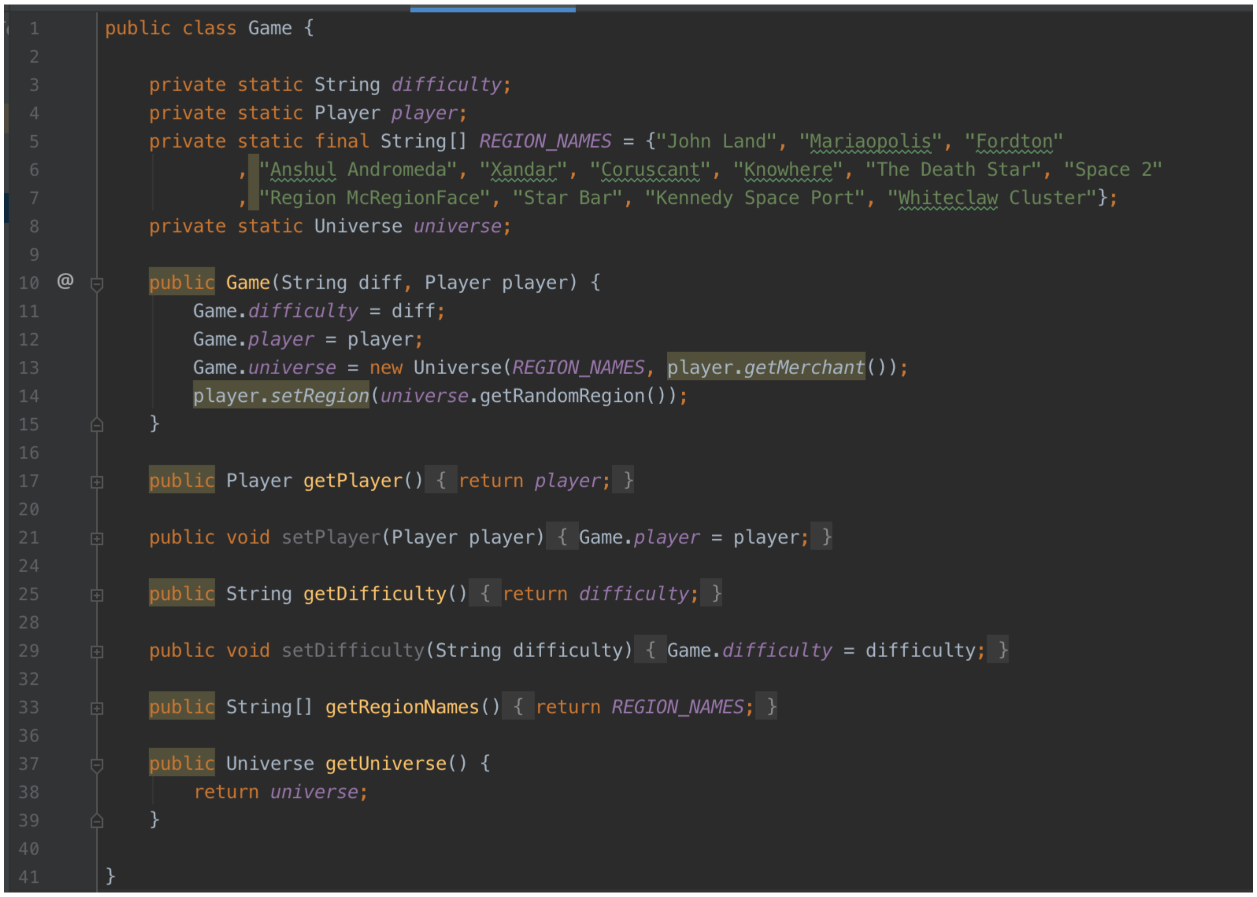


```
private static void showConfigPage(JFrame frame) {
```
UI.java ShipUpgradeType.java TechLevel.ja

```
        JButton startButton = new JButton( text: "Create Player");
        c = new GridBagConstraints();
        c.fill = GridBagConstraints.SOUTH;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.gridx = 0;
        c.gridy = 9;
        c.weighty = 0.4;
        c.anchor = GridBagConstraints.PAGE_END;
        JFrame finalFrame = frame;
        startButton.addActionListener(e -> {
            int pilot = 0;
            int fighter = 0;
            int merchant = 0;
            int engineer = 0;
            String name = "";
            try {
                pilot = Integer.parseInt(pilotBox.getText());
                fighter = Integer.parseInt(fighterBox.getText());
                merchant = Integer.parseInt(merchantBox.getText());
                engineer = Integer.parseInt(engineerBox.getText());
                name = nameBox.getText();
            } catch (Exception f) {
                f.printStackTrace();
            }
            createdPlayer = new Player(name, pilot, fighter, merchant, engineer, skillPoints);
            if (difficulty.equals("Easy")) {
                Player.setCredits(1500);
            } else if (difficulty.equals("Medium")) {
                Player.setCredits(1000);
            } else {
                Player.setCredits(500);
            }
            newGame(finalFrame);
        });
```

Description: Since the WelcomeScreen class has all the information provided by the player about their selected skill points, name, and difficulty, it naturally was given the responsibility to create the Player instance as well as the Game instance.

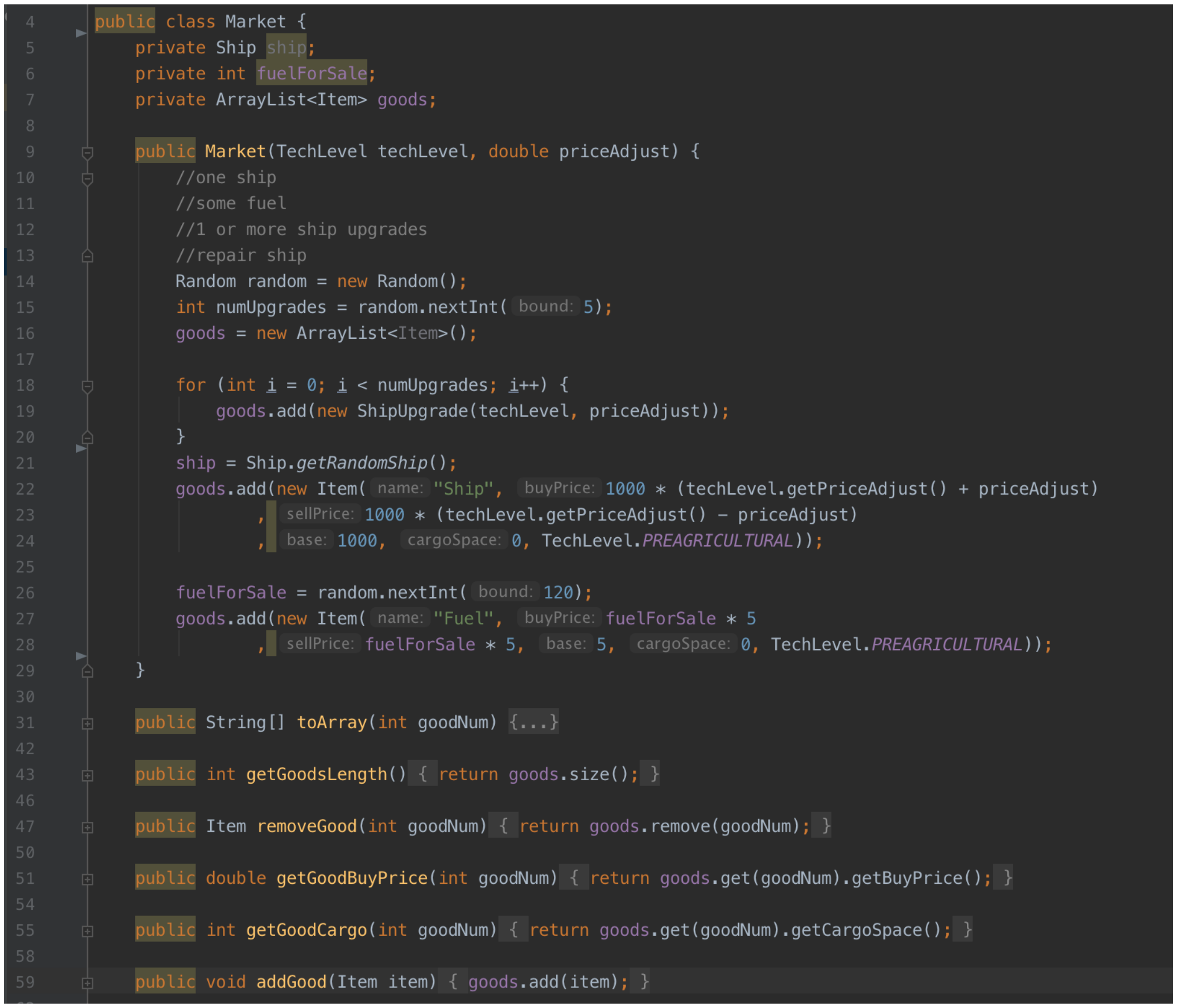
Controller



```java
public class Game {

    private static String difficulty;
    private static Player player;
    private static final String[] REGION_NAMES = {"John Land", "Mariaopolis", "Fordton"
            , "Anshul Andromeda", "Xandar", "Coruscant", "Knowhere", "The Death Star", "Space 2"
            , "Region McRegionFace", "Star Bar", "Kennedy Space Port", "Whiteclaw Cluster"};
    private static Universe universe;

    public Game(String diff, Player player) {
        Game.difficulty = diff;
        Game.player = player;
        Game.universe = new Universe(REGION_NAMES, player.getMerchant());
        player.setRegion(universe.getRandomRegion());
    }

    public Player getPlayer() { return player; }

    public void setPlayer(Player player) { Game.player = player; }

    public String getDifficulty() { return difficulty; }

    public void setDifficulty(String difficulty) { Game.difficulty = difficulty; }

    public String[] getRegionNames() { return REGION_NAMES; }

    public Universe getUniverse() {
        return universe;
    }

}
```

Description: The Game class sets the stage by assigning the difficulty, player, and the universe for the game session, and therefore controls the game's starting stages.

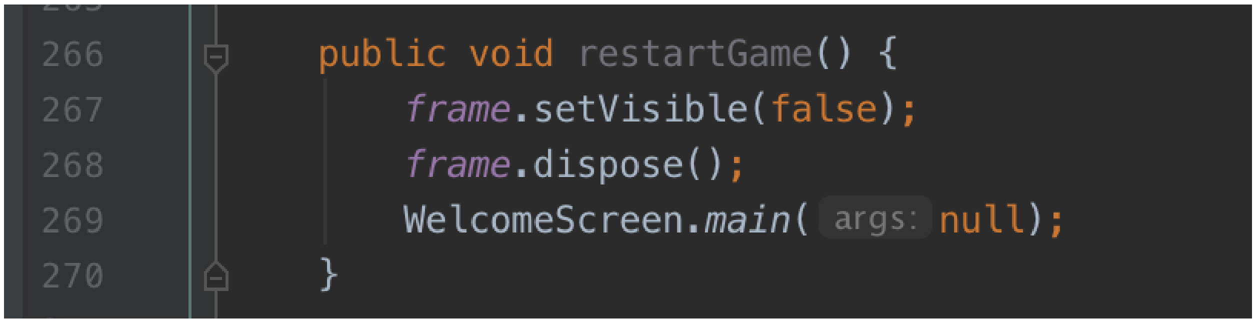
High Cohesion



```java
public class Market {
    private Ship ship;
    private int fuelForSale;
    private ArrayList<Item> goods;

    public Market(TechLevel techLevel, double priceAdjust) {
        //one ship
        //some fuel
        //1 or more ship upgrades
        //repair ship
        Random random = new Random();
        int numUpgrades = random.nextInt( bound: 5);
        goods = new ArrayList<Item>();

        for (int i = 0; i < numUpgrades; i++) {
            goods.add(new ShipUpgrade(techLevel, priceAdjust));
        }
        ship = Ship.getRandomShip();
        goods.add(new Item( name: "Ship",  buyPrice: 1000 * (techLevel.getPriceAdjust() + priceAdjust)
                ,  sellPrice: 1000 * (techLevel.getPriceAdjust() - priceAdjust)
                ,  base: 1000,  cargoSpace: 0, TechLevel.PREAGRICULTURAL));

        fuelForSale = random.nextInt( bound: 120);
        goods.add(new Item( name: "Fuel",  buyPrice: fuelForSale * 5
                ,  sellPrice: fuelForSale * 5,  base: 5,  cargoSpace: 0, TechLevel.PREAGRICULTURAL));
    }

    public String[] toArray(int goodNum) {...}

    public int getGoodsLength() { return goods.size(); }

    public Item removeGood(int goodNum) { return goods.remove(goodNum); }

    public double getGoodBuyPrice(int goodNum) { return goods.get(goodNum).getBuyPrice(); }

    public int getGoodCargo(int goodNum) { return goods.get(goodNum).getCargoSpace(); }

    public void addGood(Item item) { goods.add(item); }
```

Description: Market has one major task of aggregating all the items and creating a list of them with their pre-defined attributes. Market does not do anything not pertaining to the items to create an effective market. As a result, this class supports high cohesion because it doesn't do many unrelated things.

Low Coupling



```java
    public void restartGame() {
        frame.setVisible(false);
        frame.dispose();
        WelcomeScreen.main( args: null);
    }
```

Description: The Restart Game logic does not require resetting or otherwise handling the instance variables of any other game class, and just requires the main method on WelcomeScreen to be run after the frames are disposed of. This demonstrates that the other game classes do not overly rely on each other for their values.