

**SYSC 2004  
Winter 2018  
Lab 8**

**Objectives**

1. Abstract Classes
2. Interfaces

Provided: JUnit test files for Player.java and all the Square (sub)classes

Submission Requirements: (Exact names are required by the submit program)

- **Without a submission you will not get a mark**  
GoSquare.java, GoToJailSquare.java, IncomeTaxSquare.java, Player.java, PropertySquare.java, Square.java

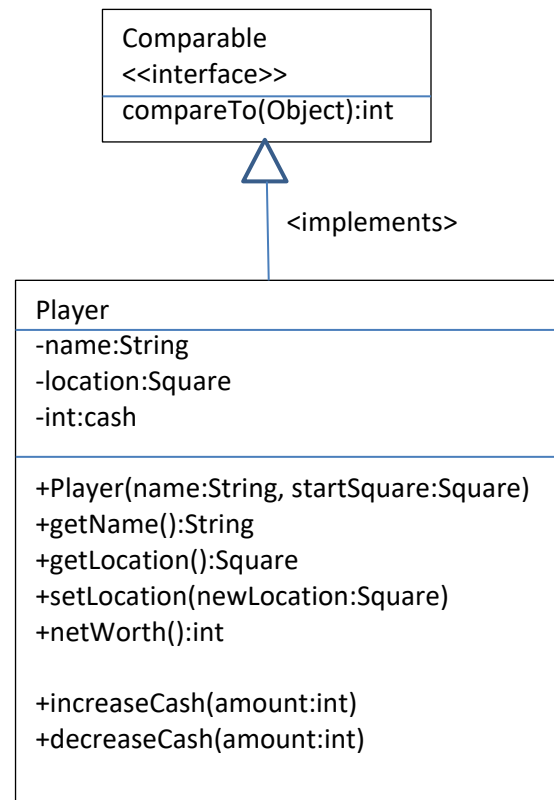
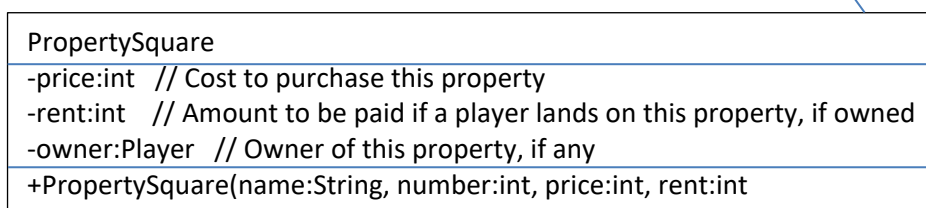
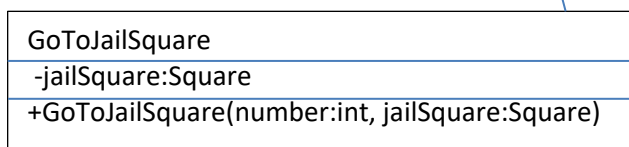
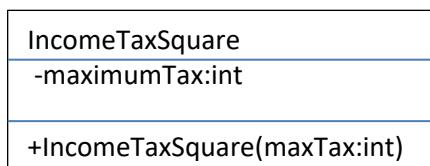
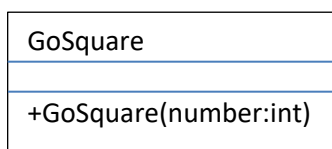
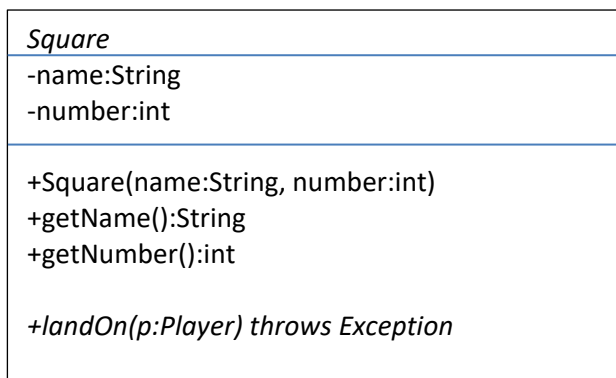
**Background**

We are going to build (part of) a Monopoly game. If your childhood has not included board games such as Monopoly, here's a couple of links.

- Images of the board: <https://goo.gl/images/DE5HjH>
- Rules of the game: <https://www.hasbro.com/common/instruct/monins.pdf> Please attention to the Go Square, Buying Property, Paying Rent and Jail.

### Notes About the UML

1. Pay attention to italics. On an exam, I may supplement with "" to ensure you see them
2. In Player, compareTo() is not shown in the UML yet you have to write it, because the <implements> relationship with Comparable demands it.
3. Likewise, in the subclass of Square, the landOn() method is not shown yet must be written, because concrete classes MUST implement any & all abstract methods from its superclass. It is often shown but I have not done so here, because I want you to learn what it means to inherit from an abstract class



All parts must be completed to be considered for a Satisfactory mark for the lab

### **Part 1 : The Player Class**

1. You are given most of the implementation of the Player class (as well as its JUnit test). You must simply implement the Comparable Interface
  - In the class declaration, add “ implements Comparable”
  - Within the class implementation, add the skeleton code for the compareTo() method
  - Look at the notes and/or do a web-search for the API of the Comparable interface. Study the return values that are expected from this method
  - Use the player’s netWorth as the measure of comparison. **One player is greater than the other if its networth is greater than the other’s netWorth.**
2. Run your code against the provided JUnit class

### **Part 2 : The Square Abstract Class**

1. Using the UML design, implement the very small Square class.
2. It is an abstract class; consequently, we cannot instantiate it. There is no JUnit test to run.<sup>1</sup>

### **Part 3 : The Easy Square Concrete SubClasses**

1. Implement the GoSquare **sub**class. (Don’t forget the extends)
  - The landOn() method should set the location of the player to **this** square, and increase the player’s cash by 200 dollars
  - **Special Instruction (with a special learning objective):** Even though, in Part 2, you (should have) advertised an Exception for Square’s abstract method landOn(), do **NOT advertise an Exception in the GoSquare’s landOn() method.**
    - Refer to the lecture notes on Exception, to the slides titled “Inheritance & Throwing”: A subclass may throw the same – or less – exceptions than its parent class.
  - Run the GoSquareTest
2. Implement the IncomeTaxSquare subclass.
  - Its constructor should set the maximumTax to 200 dollars
  - The landOn() method should set the location of the player to **this** square, and decrease the player’s cash by 10% of the player’s netWorth (up to maximumTax).
    - **As in the GoSquare class, do not advertise an Exception on landOn().**
  - Run the IncomeTaxSquareTest
3. Implement the GoToJailSquare subclass
  - This square is a little different: When a player lands on GoToJail, they are immediately (re-)directed to the Jail square.

---

<sup>1</sup> Actually, there is one. Look at it and find out how you can test an Abstract class.

- As in the GoSquare class, do not advertise an Exception on landOn().
- Run the GoToJailSquareTest

#### **Part 4 : The Property Concrete SubClasses**

The Property subclass is a wee bit more involved because a property may have an owner to which a player must pay rent if they land on the property.

1. Create an initial version of the PropertySquare subclass.
  - Its constructor should initialise all instance variables, including the owner which should be set to null (because by default a square is not owned)
  - In this initial version, the landOn() method should initially simply set the location of the player to **this** square
  - Run the PropertySquareTest . It should fail, but at least you are on your way.
2. Go back and fully implement the PropertySquare::landOn( ) method
  - **Tip:** If you don't know Monopoly, you can reverse-engineer the rules by reading the JUnit test code.
  - If a square has no owner, the player must buy the square, if they have enough cash.
    - In the PropertySquare itself, the owner must be updated.
  - If the square already has an owner, the given player must pay rent for landing on this square. Rent is paid **from** the player **to** the owner of **this** square. If a player has insufficient cash to pay the required rent, all possible rent is paid to the owner **and an exception is thrown** to announce that the given player is now out of the game.
  - Run the PropertySquareTest until all tests pass