P2: Exercise 6

Pooja Rani
public abstract class Square {
    public void landHereOrGoHome(Token token) {
        print("Owner of the token "+token.player);
    }
}

public class StarSquare extends Square {
    ...
}
public class HomeSquare extends Square {
    ...
}
Static and dynamic types

```java
public abstract class Square {
    public void landHereOrGoHome(Token token) {
        print("Owner of the token "+token.player);
    }
}

public class StarSquare extends Square {
    ...
}
public class HomeSquare extends Square {
    ...
}

StarSquare star = new StarSquare(...);
HomeSquare home = new HomeSquare(...);
Square o = star;
```

**Static type** of a variable: Type declared in the program, never changes

- star: StarSquare
- home: HomeSquare
- o: Square
Static and dynamic types

public abstract class Square {
    public void landHereOrGoHome(Token token) {
        print("Owner of the token "+token.player);
    }
}

public class StarSquare extends Square { ... }
public class HomeSquare extends Square { ... }

StarSquare star = new StarSquare(...);
HomeSquare home = new HomeSquare(...);
Square o = star;

**Dynamic type** of a variable: Type of the object bound to the variable at runtime (may change during runtime)
star: StarSquare
home: HomeSquare
o: StarSquare
public abstract class Square {
    public void landHereOrGoHome(Token token) {
        print("Owner of the token "+token.player);
    }
}

public class StarSquare extends Square { ... }
public class HomeSquare extends Square { ... }

StarSquare star = new StarSquare(...);
HomeSquare home = new HomeSquare(...);
Square o = star; o = home;

**Dynamic type** of a variable: Type of the object bound to the variable at runtime (may change during runtime)

star: StarSquare
home: HomeSquare
**o: HomeSquare**
Overloading

```java
class Renderer {
    void renderSquare(StarSquare starSquare) {
        print(starSquare);
    }
    void renderSquare(HomeSquare homeSquare) {
        print(homeSquare);
    }
}
```

Methods within a class **can have the same name** if they have different parameter lists.
Overloading

Methods within a class can have the same name if they have different parameter lists.

```java
public class Renderer {
    public void renderSquare(StarSquare starSquare) {
        print(starSquare);
    }
    public void renderSquare(HomeSquare homeSquare) {
        print(homeSquare);
    }
}

Renderer renderer = new Renderer();

StarSquare star = new StarSquare(...);
HomeSquare home = new HomeSquare(...);

renderer.renderSquare(star);
renderer.renderSquare(home);
```
Overloading

Methods within a class **can have the same name** if they have different parameter lists.

```java
public class Renderer {
    public void renderSquare(StarSquare starSquare) {
        print(starSquare);
    }
    public void renderSquare(HomeSquare homeSquare) {
        print(homeSquare);
    }
}
```

Renderer renderer = **new** Renderer();

StarSquare star = **new** StarSquare(...);
HomeSquare home = **new** HomeSquare(...);

renderer.renderSquare(star);
renderer.renderSquare(home);

Method is selected based on the **static type** of the arguments.
Overloading

```java
public class Renderer {
    public void renderSquare(StarSquare starSquare) {
        print(starSquare);
    }
    public void renderSquare(HomeSquare homeSquare) {
        print(homeSquare);
    }
}
```

Methods within a class **can have the same name** if they have different parameter lists.

Renderer renderer = new Renderer();
StarSquare star = new StarSquare(...);
HomeSquare home = new HomeSquare(...);
Square o = home;
renderer.renderSquare(o);

Does not compile: Static type of o is Square and there is no method named `renderSquare` that takes such an argument.
Different return types, but same signature does not work!
(this can not be compiled)
public abstract class Square{
    public void landHereOrGoHome(Token token) {
        print("Owner of the token "+token.player);
    }
}

public class SimpleSquare extends Square {
    @Override
    public void landHereOrGoHome (Token token) {
        super. landHereOrGoHome(token);
        if (isOccupied) {checkOccuapantType();
    }
}

@Override indicates that we are redefining an inherited method.
@Override indicates that we are redefining an inherited method.
public abstract class Square{
    public void landHereOrGoHome(Token token) {
        print("Owner of the token "+token.player);
    }
}

public class SimpleSquare extends Square {
    @Override
    public void landHereOrGoHome (Token token) {
        super. landHereOrGoHome(token);
        if (isOccupied) {checkSimpleSquareType();}
    }
}
Changing types when overriding

```java
public abstract class Square {
    public abstract Square interact(Token token);
}

public class SimpleSquare extends Square {
    @Override
    public Square interact(Token token) {
        return null;
    }
}
```
Changing types when overriding

```java
public abstract class Square {
    public abstract Square interact(Token token);
}

public class SimpleSquare extends Square {
    @Override
    public SimpleSquare interact(Token token) {
        return null;
    }
}
```

Return types can be more specific when overriding methods (SimpleSquare must be a subtype of Square).
Changing types when overriding

```java
public abstract class Square {
    public abstract Square interact(Token token);
}

public class SimpleSquare extends Square {
    @Override
    public SimpleSquare interact(Token token) {
        return null;
    }
}
```
Changing types when overriding

```java
public abstract class Square {
    public abstract Square interact(Token token);
}

public class SimpleSquare extends Square {
    @Override
    public SimpleSquare interact(Object object) {
        return null;
    }
}
```

Accept at least what the inherited method accepts.
public abstract class Square {
    protected int xPosition, yPosition;

    public Square(int x, int y) {
        this.xPosition = x;
        this.yPosition = y;
    }
}

public class HomeSquare extends Square {
    private Color color;

    public HomeSquare(Color color, int x, int y) {
        this.color = color;
    }
}
public abstract class Square {
    protected int xPos, yPos;

    public Square(int x, int y) {
        this.xPos = x;
        this.yPos = y;
    }
}

public class HomeSquare extends Square {
    private Color color;

    public HomeSquare(Color color, int x, int y) {
        this.color = color;
    }

    Does not work: Square does not have a default constructor.
public abstract class Square {
    protected int xPosition, yPosition;

    public Square(int x, int y) {
        this.xPosition = x;
        this.yPosition = y;
    }
}

public class HomeSquare extends Square {
    private Color color;
    public HomeSquare(Color color, int x, int y) {
        this.color = color;
        super(x, y);
    }
}
public abstract class Square {
    protected int xPosition, yPosition;

    public Square(int x, int y) {
        this.xPosition = x;
        this.yPosition = y;
    }
}

public class HomeSquare extends Square {
    private Color color;

    public HomeSquare(Color color, int x, int y) {
        this.color = color;
        super(x, y);
    }
}

Still bad: call to super constructor must be first statement
public abstract class Square {
    protected int xPosition, yPosition;

    public Square(int x, int y) {
        this.xPosition = x;
        this.yPosition = y;
    }
}

public class HomeSquare extends Square {
    private Color color;

    public HomeSquare(Color color, int x, int y) {
        super(x, y);
        this.color = color;
    }
}

This is how it’s done
Attributes and inheritance

- Be careful when working with inherited attributes
- Private attributes: Inherited, but not accessible!
Attributes and inheritance

- Private attributes: Inherited, but not accessible!
Attributes and inheritance

- Private attributes: Inherited, but not accessible!

```java
public abstract class Square {
    private int xPosition, yPosition;

    public Square(int x, int y) {
        this.xPosition = x; this.yPosition = y;
    }
}

public class HomeSquare extends Square {
    public HomeSquare(Color color, int x, int y) {
        super(x, y);
        print(xPosition +","+yPosition);
    }
}
```

Does not compile!

x and y are not accessible
Attributes and inheritance

- Private attributes: Inherited, but not accessible!

```java
public abstract class Square {
    protected int xPosition, yPosition;

    public Square(int x, int y) {
        this.xPosition = x;
        this.yPosition = y;
    }
}

public class HomeSquare extends Square {

    public HomeSquare(Color color, int x, int y) {
        super(x, y);
        print(xPosition +","+yPosition);
    }
}
```

Now we have access
Attributes and inheritance

- Private attributes: Inherited, but not accessible!

```java
public abstract class Square {
    private int xPosition, yPosition;
    public Square(int x, int y) {
        this.xPosition = x; this.yPosition = y;
    }

    protected int getX() {return xPosition;}
    protected int getY() {return yPosition;}
}

public class HomeSquare extends Square {
    public HomeSquare(Color color, int x, int y) {
        super(x, y);
        print(getX() + "", " + getY());
    }
}
```

This works too
“Overriding” attributes

```java
public class Square {
    public String name;
    public String getName() { return this.name; }
}

public class HomeSquare extends Square {
    public String name;
    public String getName() { return this.name; }
}
```
“Overriding” attributes

```java
public class Square {
    public String name;
    public String getName() { return this.name; }
}

public class HomeSquare extends Square {
    public String name;
    public String getName() { return this.name; }
}

HomeSquare home = new HomeSquare();
Square obj = home;
obj.name = "home"

System.out.println(home.getName());
System.out.println(obj.getName());
```
“Overriding” attributes

```java
public class Square {
    public String name;
    public String getName() { return this.name; }
}

public class HomeSquare extends Square {
    public String name;
    public String getName() { return this.name; }
}
```

```java
HomeSquare home = new HomeSquare();
Square obj = home;
obj.name = "home";

System.out.println(home.getName()); // null
System.out.println(obj.getName());   // null
```
“Overriding” attributes

```java
public class Square {
    public String name;
    public String getName() { return this.name; }
}

public class HomeSquare extends Square {
    public String name;
    public String getName() { return this.name; }
}

HomeSquare home = new HomeSquare();
Square obj = home;
obj.name = "home";

System.out.println(home.name);  \rightarrow null
System.out.println(obj.name);  \rightarrow "home"
```
Overloading & Overriding

- **Overloading**
  - Same method name, different signatures
  - Return types must match

- **Overriding**
  - Redefine inherited methods
  - Use “super.methodname(...)” (or “super(...)” in constructors)
  - Must call a super constructor if there’s no argumentless constructor available in the superclass
  - Accept more, return less
Exercise 6: More Ludo!

- Third stage: game rules
  - Implement the remaining action of player
  - Keep track of the game state

- Fourth stage: Random game runner
  - Initialize a new game with 2-4 players
  - Print the game state (player name, rolled number, board state)
  - Play until a player wins
Better commit quality! :-)  
  - (with exceptions)

Good code quality
  - JavaDoc, contracts, tests, ...
  - Keep it up!

Exercise 6 is not that big... use it to catch up!