

P2: Exercise 6

Pooja Rani

Square Objects

```
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

```
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

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; 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

```
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.

Overloading

```
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(...);
```

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

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

Overloading

```
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(...);
```

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

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

Overloading

```
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

Overloading

```
public class Renderer {  
    public String renderSquare(Square o) {  
        ...  
    }  
    public void renderSquare(Square o) {  
        ...  
    }  
}
```

Different return types, but same signature does not work!
(this can not be compiled)

Overriding

```
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) {checkOccupantType();
        }
    }
}
```

@Override indicates that we are redefining an inherited method.

Overriding

```
public abstract class Square{  
    public void landHereOrGoHome(Token token) {  
        print("Owner of the token "+token.player);  
    }  
}  
public class SimpleSquare extends Square {  
    @Override  
    public void landHereOrHome (Token token) {  
        Typo! Does not compile!  
    }  
}
```

@Override indicates that we are redefining an inherited method.

Overriding

```
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 (isOwner()) {
            landHereOrGoHome(token);
        }
    }
}
```

“super” can be used to call the overridden method.

Changing types when overriding

```
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

```
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

```
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

```
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.

Calling an inherited 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;  
    }  
}
```

Calling an inherited 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;  
    }  
}
```

Does not work: Square does not have a default constructor.

Calling an inherited 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);  
    }  
}
```

Calling an inherited 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);  
    }  
}
```

Still bad: call to super constructor must be first statement

Calling an inherited 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) {  
        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!

```
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!

```
public abstract class Square {  
    protected int xPositon, yPositon;  
  
    public Square(int x, int y) {  
        this.xPositon = x; this.yPositon = y;  
    }  
}  
  
public class HomeSquare extends Square {  
  
    public HomeSquare(Color color, int x, int y) {  
        super(x, y);  
        print(xPositon + "," + yPositon);  
    }  
}
```

Now we have access

Attributes and inheritance

- Private attributes: Inherited, but not accessible!

```
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

```
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

```
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

```
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());
```

→ null

→ null

“Overriding” attributes

```
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);  
System.out.println(obj.name);
```

→ null

→ “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

Comments

- 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!