P2 – Exercise Hour

Pascal André
3 April, 2020
Outline

• Inheritance

• Exercise 5: Recap

• Exercise 6: Outlook
public abstract class Tile {
    public void enter(Player player) {
        System.out.println(player + " enters " + this);
    }
}

public class Floor extends Tile {…}
public class Wall extends Tile {…}

Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = wall;
public abstract class Tile {
    public void enter(Player player) {
        System.out.println(player + " enters " + this);
    }
}

public class Floor extends Tile {...}
public class Wall extends Tile {...}

Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = wall;

The Static Type of the variable...
• is declared in the program
• does never change

wall: Wall
floor: Floor
tile: Tile
public abstract class Tile {
    public void enter(Player player) {
        System.out.println(player + " enters " + this);
    }
}

public class Floor extends Tile {...}
public class Wall extends Tile {...}

Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = wall;

wall: Wall
floor: Floor
tile: Wall

The Dynamic Type of the variable...
• is bound to the object at runtime
• may change during execution of program
public abstract class Tile {
    public void enter(Player player) {
        System.out.println(player + " enters " + this);
    }
}

public class Floor extends Tile {...}
public class Wall extends Tile {...}

Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = wall; tile = floor;

wall: Wall
floor: Floor
**tile: Floor**

The Dynamic Type of the variable...
- is bound to the object at runtime
- may change during execution of program
public class Renderer {
    public void renderTile(Wall wall) {
        print(wall);
    }
    public void renderTile(Floor floor) {
        print(floor);
    }
}
Overloading

```java
public class Renderer {
    public void renderTile(Wall wall) {
        print(wall);
    }
    public void renderTile(Floor floor) {
        print(floor);
    }
}
```

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

```java
public class Renderer {
    public void renderTile(Wall wall) {
        print(wall);
    }

    public void renderTile(Floor floor) {
        print(floor);
    }
}
```

Renderer renderer = new Renderer();
Wall wall = new Wall(...);
Floor floor = new Floor(...);

renderer.renderTile(wall);
renderer.renderTile(floor);

Methods within a class can have the same name if they have different parameter lists.
public class Renderer {
    public void renderTile(Wall wall) {
        print(wall);
    }
    public void renderTile(Floor floor) {
        print(floor);
    }
}

Renderer renderer = new Renderer();

Wall wall = new Wall(…);
Floor floor = new Floor(…);

renderer.renderTile(wall);
renderer.renderTile(floor);

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

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

```java
public class Renderer {
    public void renderTile(Wall wall) {
        print(wall);
    }
    public void renderTile(Floor floor) {
        print(floor);
    }
}
```

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

Renderer renderer = new Renderer();

Wall wall = new Wall(…);
Floor floor = new Floor(…);
Tile tile = floor;

renderer.renderTile(tile);
Overloading

```java
public class Renderer {
    public void renderTile(Wall wall) {
        print(wall);
    }
    public void renderTile(Floor floor) {
        print(floor);
    }
}
```

Renderer renderer = new Renderer();

Wall wall = new Wall(...);
Floor floor = new Floor(...);
Tile tile = floor;

renderer.renderTile(tile);

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

Does not compile: Static type of tile is Tile. There is no method renderTile(Tile tile) that takes such an argument.
Overloading

```java
public class Renderer {
    public String renderTile(Wall wall) {
        return "Wall";
    }
    public void renderTile(Wall wall) {
        print(floor);
    }
}
```

Different return types but same signature does not work!
This can not be compiled.
public abstract class Tile {
    public void landHere(Player player) {
        // define basic landing of player on tile
    }
}

public class Floor extends Tile {
    @Override
    public void landHere(Player player) {
        super.landHere(player);
        // define additional floor-related details when landing here
    }
}

@Override indicates that we are redefining an inherited method
public abstract class Tile {
    public void landHere(Player player) {
        // define basic landing of player on tile
    }
}

public class Floor extends Tile {
    @Override
    public void landHere(Player player) {
        super.landHere(player);
        // define additional floor-related details when landing here
    }
}

"super" can be used to call the overridden method.
public abstract class Tile {
    /**
     * Return yourself if argument is same tile, null otherwise
     */
    public abstract Tile matches(Tile tile) {...}
}

public class Floor extends Tile {
    @Override
    public Tile matches(Tile tile) {...}
}
public abstract class Tile {
   /**
    * Return yourself if argument is same tile, null otherwise
    */
   public abstract Tile matches(Tile tile) {...}
}

public class Floor extends Tile {
   @Override
   public Floor matches(Tile tile) {...}
}
public abstract class Tile {

    /**
     * Return yourself if argument is same tile, null otherwise
     */
    public abstract Tile matches(Tile tile) {...}

}

public class Floor extends Tile {

    @Override
    public Floor matches(Tile tile) {...}

}
Changing Types when Overriding

```java
public abstract class Tile {
    /**
     * Return yourself if argument is same tile, null otherwise
     */
    public abstract Tile matches(Tile tile) {...}
}

public class Floor extends Tile {
    @Override
    public Floor matches(Object object) {...}
}
```

Option 2:
Accept at least what the inherited method accepts.
public abstract class Tile {
    protected int xPosition, yPosition;

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

public class Floor extends Tile {
    private Game game;

    public Floor (Game game, int x, int y) {
        this.game = game;
    }
}
public abstract class Tile {
    protected int xPosition, yPosition;

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

public class Floor extends Tile {
    private Game game;

    public Floor(Game game, int x, int y) {
        this.game = game;
    }
}

Calling an Inherited Constructor

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

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

public class Floor extends Tile {
    private Game game;

    public Floor(Game game, int x, int y) {
        super(x, y);
        this.game = game;
    }
}
public abstract class Tile {
    private int xPosition, yPosition;

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

public class Floor extends Tile {
    public Floor (int a, int b) {
        super (a, b);
        System.out.println(xPosition + "", " + yPosition);
    }
}
public abstract class Tile {
    private int xPosition, yPosition;
    public Tile(int x, int y) {
        this.xPosition = x;
        this.yPosition = y;
    }
}

public class Floor extends Tile {
    public Floor (int a, int b) {
        super (a, b);
        System.out.println(xPosition + "", " + yPosition);
    }
}
public abstract class Tile {
    protected int xPosition, yPosition;

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

public class Floor extends Tile {
    public Floor(int a, int b) {
        super(a, b);
        System.out.println(xPosition + "", " + yPosition);
    }
}
### Attributes and Inheritance

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

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

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

public class Floor extends Tile {
    public Floor(int a, int b) {
        super(a, b);
        System.out.println(getX() + "", " + getY());
    }
}
```

Using inherited getter-methods works too.
Shadowing Attributes

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

public class Floor extends Tile {
    public String name;
    public String getName() {return this.name}
}
```
Shadowing Attributes

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

public class Floor extends Tile {
    public String name;
    public String getName() { return this.name; }
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = “floor”;

System.out.println(floor.getName());
System.out.println(tile.getName());
```
Shadowing Attributes

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

public class Floor extends Tile {
    public String name;
    public String getName() {return this.name}
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = “floor”;

System.out.println(floor.getName()); // → null
System.out.println(tile.getName());   // → null
```
Shadowing Attributes

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

public class Floor extends Tile {
    public String name;
    public String getName() { return this.name; }
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = "floor";

System.out.println(floor.name);
System.out.println(tile.name);
```
public abstract class Tile {
    public String name;
    public String getName() {return this.name}
}

public class Floor extends Tile {
    public String name;
    public String getName() {return this.name}
}

Floor floor = new Floor();
Tile tile = floor;
tile.name = “floor”;

System.out.println(floor.name);  // null
System.out.println(tile.name);    // “floor”
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 5 – Recap Stage 1

For the first iteration of the Sokoban game, you should have added:

• Initial game setup
  • Prepare your game’s representation by setting up required classes
  • e.g. create classes like `Game`, `Player`, `Tile` etc.

• Parser
  • Reads game specification files and creates game instance
  • Tests to check that parser creates game correctly

• Renderer
  • Prints a game state to standard output
  • Tests to check that renderer prints game state correctly

```
git tag -a v1 -m “sokoban1”
git push origin --tags
```
Exercise 5 – Recap Stage 2

For the second iteration of the Sokoban game, you should have added:

• Player Movement
  • Allow player to move around on the board (not required to be interactive)
  • Tests to show that player movement is working

• Game Winning Scenario
  • Game should terminate when all boxes are on a goal tile

• Tests
  • Add a JUnit test that solves the level `levels/basic1.sok`
  • Use parser to create new game; instruct player to move on board to solve puzzle; use renderer to print each game state incl. game winning message

• Debugger
  • In a markdown file describe 3+ cases where you have used the debugger
Exercise 6 – Outlook

Fully complete Exercise 5 (1st + 2nd stage) and then tag your final solution:

```bash
git tag -a v2 -m “sokoban2”
git push origin --tags
```

• Apply the concepts we have covered so far:
  • Object-Oriented Design Principles
  • Responsibility Driven Design
  • Design by Contract
  • Unit Testing
  • JavaDoc for class and method comments
For the third iteration of the Sokoban game, you should implement:

- **Validation of Player Movement**
  - Only allow valid moves (do not allow moving through walls)
- **Box Movement**
  - Player can move boxes (if possible in current game state)
- **New `C` Tile**
  - Add new “Completed Tile” that represents goal tile with a box on it
  - Update classes: parser can read new tile and renderer can visualize it
- **Tests**
  - Add unit tests to check your implementation of the above three tasks
furthermore…

- **Interactivity**
  - Make game interactive by adding main routine to run the program
  - Take user input to move the player
  - Re-render board after each step so player sees current game representation

- **UML: Sequence Diagram**
  - User writes input command that pushes box onto goal tile

```bash
git tag -a v3 -m "sokoban3"
git push origin --tags
```
Exercise 6 – Outlook Stage 4

For the fourth iteration of the Sokoban game, you should add:

• Refactoring
  • Write markdown file documenting refactoring process of any class

• Packages
  • Create different packages for your classes

• Override `toString()` Methods
  • Provide reasonable `toString()` method for all objects (except test classes)

• Minimize Mutability
  • Declare instance variables which are unmodified after initialization as `final`
Exercise 6 – Outlook Stage 4

furthermore...

• Encapsulation and Information Hiding
  • Use appropriate access modifiers for all methods and instance variables

• Check Parameters for Validity
  • Write `assert` statements to check method parameters for their validity
  • Write JavaDoc comments for all public methods incl. parameter restrictions

Once you have finished, tag your solution:

```
git tag -a v4 -m “sokoban4”
git push origin --tags
```

Deadline: Friday, 24 April, 13:00