P2 – Exercise Hour

Pascal André

23 April, 2021

Outline

- Inheritance
- Exercise Overview

Static and Dynamic Types

```
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;
```

Static and Dynamic Types

```
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: Tile

The **Static Type** of the variable…

- is declared in the program
- does never change

Static and Dynamic Types

```
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

Static and Dynamic Types

```
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

Overloading

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

Overloading

```
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

```
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(…);

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

Overloading

```
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(…);

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

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

Overloading

```
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

```
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);
```

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

Overloading

```
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.

Overriding

```
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**

Overriding

```
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.

Changing Types when Overriding

```
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) {…}
}
```

Changing Types when Overriding

```
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) {…}
}
```

Option 1:
Return types can be **more specific** when overriding methods.
Requirement: `Floor` must be subtype of `Tile`.

Changing Types when Overriding

```
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

```
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.

Calling an Inherited 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) {
        this.game = game;
    }
}
```

Calling an Inherited 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) {
        this.game = game;
    }
}
```

Does not work:
Tile does not have a **default constructor**.

Calling an Inherited 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;
    }
}
```

Call an inherited constructor with **`super(…)`**.
Note: Must be the first statement.

Attributes and Inheritance

```
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);
    }
}
```

Attributes and Inheritance

```
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);
    }
}
```

Does not compile:
xPosition and yPosition are **not accessible**.

Attributes and Inheritance

```
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);
    }
}
```

Now we have access

Attributes and Inheritance

```
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

```
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

```
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

```
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

```
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);
```

Shadowing Attributes

```
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);
```

→ null
→ "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 Overview

Current schedule for the exercises:

- **Exercise 7**
  - The last stage of the Checkers game is due today April 23rd.
- **For next week**
  - No new exercise – you get time to catch up on revisions and extended exercises. We try to give feedback for exercise 7 over the weekend so that you have time to revise any issues until next Friday.
- **Exercise 8**
  - Starts in a week on April 30th.

Questions?