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;
pubic 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
Static and Dynamic Types

```java
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

```java
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;
```

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

Overloading

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

Overloading

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);
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) {...}
}
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

```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(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(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);
    }
}
Attributes and Inheritance

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

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
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

For the first iteration, you should have:

• 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

• Game/Player
  • Implement player movement on the board by one step
  • Tests to check that player moves correctly
Exercise 6 - Outlook

Fully complete Exercise 5 (first iteration) and then tag your solution:

```bash
git tag -a v1 -m “Quoridor stage 1”
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
Exercise 6 - Outlook

For the second iteration, you should implement:

• Wall placement
  • Each turn, player can either move one step or place wall

• Validate wall placement
  • Check that all players are able to reach one goal tile at all times

Once you have finished, tag your solution:

```
git tag -a v2 -m "Quoridor stage 2"
git push origin --tags
```

Deadline: Friday, 12 April, 13:00
Questions?