P2: Exercise Session 6

Claudio Corrodi

String concatenation

```
public String toString(int[] numbers) {
    String result = "";
    for (int i = 0; i < numbers.length-1; i++) {
        result += numbers[i];
        result += ", ";
    }
    result += numbers[numbers.length-1];
    return result;
}
```

What's wrong with this code?

String concatenation

```
public String toString(int[] numbers) {
    String result = "";
    for (int i = 0; i < numbers.length-1; i++) {
        result += numbers[i];
        result += ", ";
    }
    result += numbers[numbers.length-1];
    return result;
}
```

How many Strings are created?

String concatenation

```
public String toString(int[] numbers) {
    String result = "";
    for (int i = 0; i < numbers.length-1; i++) {
        result += numbers[i];
        result += ", ";
    }
    result += numbers[numbers.length-1];
    return result;
}
```

How many Strings are created?

One new String object for each assignment!

String concatenation

```
public String toString(int[] numbers) {
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < numbers.length-1; i++) {
        result.append(numbers[i]);
        result.append(", ");
    }
    result.append(numbers[numbers.length-1]);
    return result.toString();
}
```

Use StringBuilder/StringBuffer to build a String over time.

Floating point operations

What does this print?

```
System.out.println(new BigDecimal(0.1).toPlainString());
```

Floating point operations

What does this print?

```
System.out.println(new BigDecimal(0.1).toPlainString());
```

0.1000000000000000055511151231257827021181583404541015625

Floating point operations

What does this print?

```
System.out.println(new BigDecimal(0.1).toPlainString());
```

0.1000000000000000055511151231257827021181583404541015625

Floating point literals can not always be stored precisely!

Floating point operations

What does this print?

```
System.out.println(new BigDecimal(0.1).toPlainString());
```

0.1000000000000000055511151231257827021181583404541015625

Floating point literals can not always be stored precisely!

```
double f = 2.00;
double g = 1.10;
System.out.println(f - g == 0.90); // false
System.out.println(f - g == 0.8999999999999999); // true
```

Floating point operations

What if I want to write a test?

```
@Test
public void doubleAddition() {
    double r = 0.1 + 0.2;
    assertTrue(r == 0.3);
}
```

Floating point operations

What if I want to write a test?

```
@Test
public void doubleAddition() {
    double r = 0.1 + 0.2;
    assertTrue(r == 0.3);
}
```

```
junit.framework.AssertionFailedError:
Expected :0.3
Actual   :0.30000000000000004
  <Click to see difference>
```

Floating point operations

What if I want to write a test?

- Do not compare floats/doubles using ==
- Use an error tolerance

```
@Test
public void doubleAdditionFixed() {
    double epsilon = 0.0000001;
    double r = 0.1 + 0.2;
    assertTrue(Math.abs(0.3-r) < epsilon);
}
```

This test doesn't fail!

Try-catch-finally

```
class A {
    int m() {
        try { return 1; }
        catch (Exception err) { return 2; }
        finally { return 3; }
    }
}
```

```
A a = new A();
System.out.println(a.m());
```

What is being printed? 1, 2, or 3?

Try-catch-finally

```
class A {
    int m() {
        try { return 1; }
        catch (Exception err) { return 2; }
        finally { return 3; }
    }
}
```

```
A a = new A();
System.out.println(a.m());
```

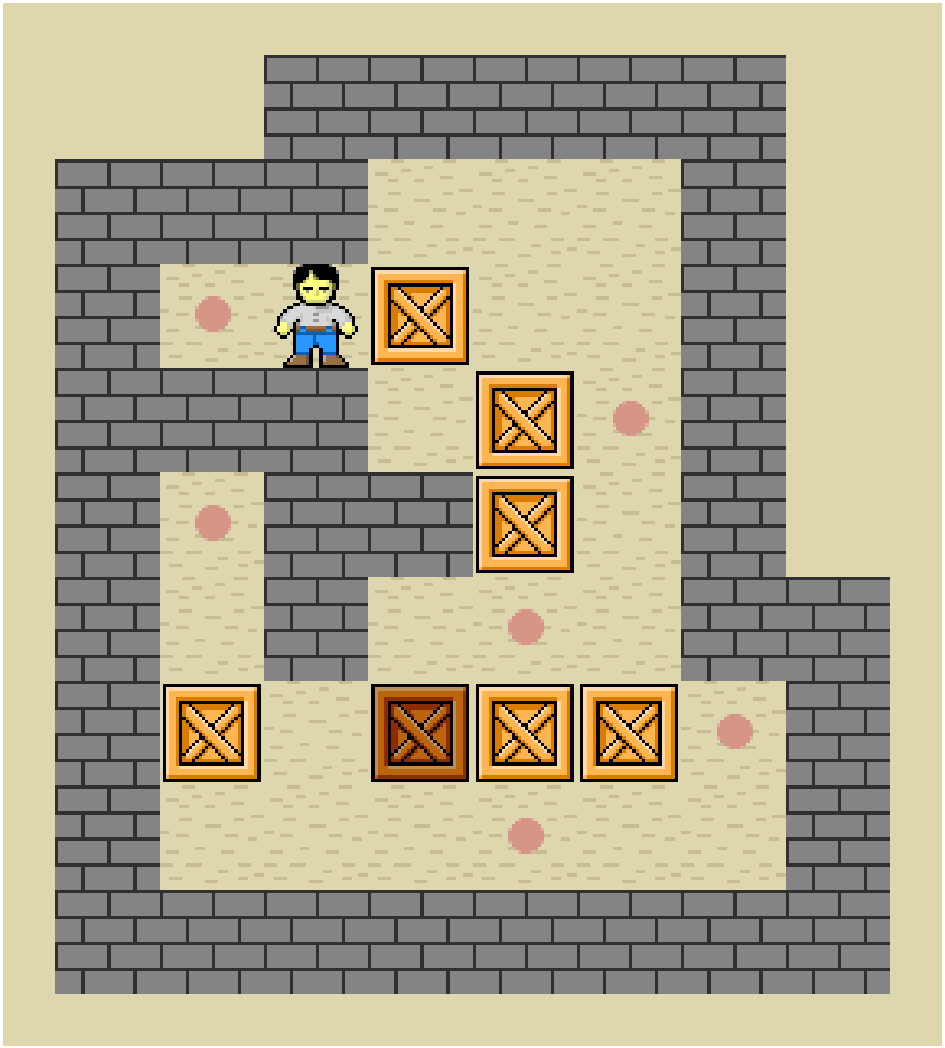
What is being printed? 1, 2, or 3?

3! “finally” blocks are **always** executed

P2: Exercise 6

Claudio Corrodi

Sokoban



https://en.wikipedia.org/wiki/Sokoban#/media/File:Sokoban_ani.gif

Sokoban: First Stage

You implement:

- Parse level files
- Basic game structure
- Text renderer
- Test parser

Sokoban: First Stage

You implement:

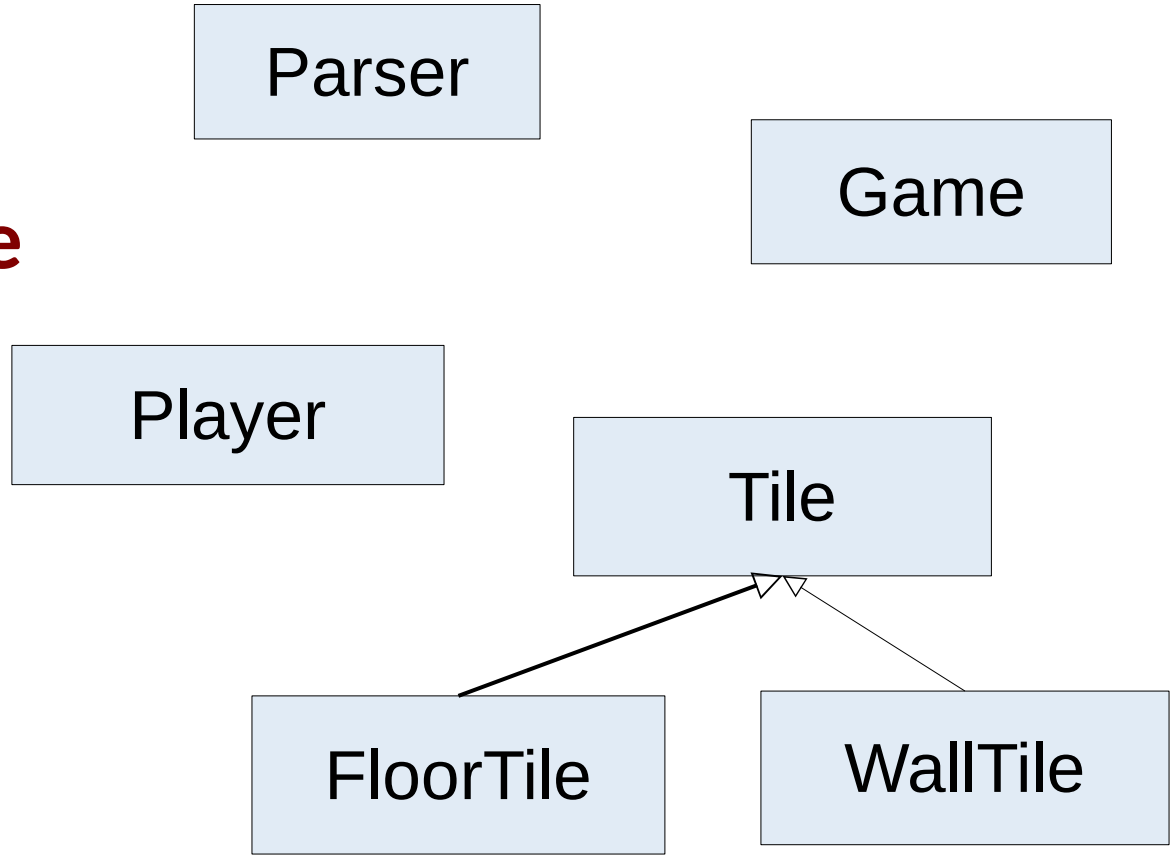
- **Parse level files**
- Basic game structure
- Text renderer
- Test parser

```
7  6
########
#      #
#P     #
###B   #
###   G#
########
```

Sokoban: First Stage

You implement:

- Parse level files
- **Basic game structure**
- Text renderer
- Test parser



Sokoban: First Stage

You implement:

- Parse level files
- Basic game structure
- **Text renderer**
- Test parser

- Pass through your game representation
- Print the **current state** of the game with the same characters as in the level files

Sokoban: First Stage

You implement:

- Parse level files
- Basic game structure
- Text renderer
- **Test parser**

Don’t forget to test other parts of the game as well!

Sokoban: Second Stage

You implement:

- Player movement
  - Handle moving towards walls, pushing boxes
- End the game when the puzzle is solved
- Test that solves the first level

Sokoban: Git tags

- Tags are “names for commits”
- Commonly used for specifying which commit is associated with a specific version
- You’ll need to tag both stages
  - We will look at these commits

Sokoban: Git tags

- Tag stages:

```
git tag -a sokoban1
git tag -a sokoban2
```

- **Don’t forget to push the tags to origin!**

```
git push --tags origin master
```

- Need help with git? The manpages are really helpful!

```
man git
man git-tag
...
```

Comments

- Apply what you learned so far
  - Unit tests, JavaDoc comments, design by contract, …
- Make sure that your code is readable and properly documented
  - If not, you might get a nef/revise or even a direct fail!
- Keep writing proper git commit messages
  - Yes, we read them.
  - Not doing this can also result in a fail!