

# Assess Yourself

Write the Character `createCharacter(String name)` method, that throws an `InvalidNameException` if the name provided is *not* a Game of Thrones character name.

```
private static final String[] VALID_NAMES =
    {"Jon Snow", "Arya Stark", "Danaerys Targaryen", ...}

public Actor createActor(String name)
    throws InvalidNameException {

    for (String s : VALID_NAMES) {
        if (s.equals(name)) {
            return new Character(name);
        }
    }

    throw new InvalidNameException(name);
}
```

SWEN20003  
Object Oriented Software Development

Software Testing and Design

Semester 1, 2018

# The Road So Far

- Java Foundations
- Classes and Objects
- Abstraction
- Advanced Java
  - ▶ Generic Classes
  - ▶ Generic Programming
  - ▶ Exception Handling

# Lecture Objectives

After this lecture you will be able to:

- **Write better** code
- **Design better** software
- **Test** your software for bugs

# Documentation

# Boring Stuff (Code Formatting)

While writing code is largely subjective, there are plenty of **conventions** that most programmers share:

- Use consistent layout (indentation, white space)
- Avoid long lines (80 characters is “historic”)
- Beware of tabs
- Lay out comments and code neatly
- Sensible naming of variables, method and classes
- Divide long files into sections with clear purposes
- Avoid copy and pasting/duplicating code
- Use a comment to explain each section

# Boring Stuff (Comment Style)

While writing comments is largely subjective, there are plenty of **conventions** that most programmers share:

- Intended primarily for **yourself**, and developers writing code **with** you
- Code should be written to be **self-documenting**; readable without extra documentation
- Comments “tell the story” of the code
- If your code were removed, comments should be sufficient to “piece together” the algorithm
- Comments should be attached to *blocks* of code, which loosely correspond to *steps* in completing your algorithm

# Boring Stuff (Comment Placement)

## Bad Comment Placement

```
<line of code>  
// This is a comment below my code
```

## Terrible Comment Placement

```
<line of code> // This is an inline comment
```

## Great Comment Placement

```
// This is a comment above my code  
<line of code>
```

Comments appearing **before** code are like a “prologue” for your code; they introduce the *idea* of the code before you actually try to digest it.



# Boring Stuff (Javadoc)

```
javadoc.equals(comments); // false
```

- Javadoc is a special kind of comment that can be **compiled to HTML**
- Intended primarily for developers **using** your program (exactly like Slick documentation)
- Used to document packages, classes, methods, and attributes (among others)
- Should document how to **use** and **interact** with your classes and their methods
- Various @ tags (like @param and @return) for generating specific documentation

# Software Design

# Poor Design Symptoms

When your design sucks...

**Rigidity** Hard to modify the system because changes in one class/method cascade to many others

**Fragility** Changing one part of the system causes **unrelated parts** to break

**Immobility** Cannot decompose the system into reusable modules

**Viscosity** Writing “hacks” when adding code in order to preserve the design

**Complexity** Lots of clever code that isn't necessary right now; premature optimisation is bad

**Repetition** Code looks like it was written by Cut and Paste

**Opacity** Lots of convoluted logic, design is hard to follow

# GRASP

In SWEN30006 you'll learn about

- G General
- R Responsibility
- A Assignment
- S Software
- P Patterns/Principles

## Keyword

*GRASP*: A series of guidelines for assigning responsibility to classes in an object-oriented design; how to break a problem down into modules with clear responsibility.

# GRASP Basics

## Keyword

*Cohesion*: Classes are designed to **solve clear, focused problems**. The class' methods/attributes are related to, and work towards, this objective. Designs should have **maximum** cohesion.

## Keyword

*Coupling*: The degree of interaction between classes; invoking another class' methods or accessing/modifying its variables. Designs should have **minimum** coupling.

# GRASP Basics

## Keyword

*Open-Closed Principle:* Classes should be **open** to extension, but **closed** to modification.

In practice, this means if we need to *change* or *add* functionality to a class, we should not modify the original, but instead use **inheritance**.

# GRASP Basics

## Keyword

*Abstraction*: Solving problems by creating *abstract data types* to represent problem components; achieved in Java through *classes*, which represent data and actions.

## Keyword

*Encapsulation*: The details of a class should be kept *hidden* or *private*, and the user's ability to access the hidden details is *restricted* or *controlled*. Also known as **data** or **information hiding**.

# GRASP Basics

## Keyword

*Polymorphism*: The ability to use an object or method in many different ways; achieved in Java through *ad hoc* (overloading), *subtype* (overriding, substitution), and *parametric* (generics) polymorphism.

## Keyword

*Delegation*: Keeping classes *focused* by passing work to other classes.

Computations should be performed in the class with the *greatest amount of relevant information*.



# Software Testing

# Bug Fixing

How do you normally find/fix a bug?

- Print statements

```
System.out.println("Why does my code not reach here?");
```

- Google

How to fix my Java code

- Forums (Stackoverflow, etc.)

Someone please help my code is broken

## Bug Fixing

Java offers a structured method for **testing**:

### Keyword

*Unit*: A small, well-defined component of a software system with one, or a small number, of responsibilities.

### Keyword

*Unit Test*: Verifying the operation of a *unit* by testing a single *use case* (input/output), intending for it to **fail**.

### Keyword

*Unit Testing*: Identifying bugs in software by subjecting every *unit* to a suite of *tests*.

# Assess Yourself

What **use cases** can you think of for the following code?

```
public boolean makeMove(Player player, Move move) {  
  
    int row = move.row;  
    int col = move.col;  
  
    if (row < 0 || row >= SIZE || col < 0 || col >= SIZE ||  
        !board[row][col].equals(EMPTY)) {  
        return false;  
    }  
  
    board[row][col] = player.getCharacter();  
  
    return true;  
}
```

# Assess Yourself

## Use Cases:

- Valid input
- Invalid input

Great... But that's not helpful

# Assess Yourself

## Use Cases:

- Valid input
  - ▶ Does a move with row and column on the board...
    - ★ Mutate the board if it is empty?
    - ★ Mutate the right position on the board?
    - ★ Does the right character get used?
    - ★ Does the method return true in this case?
- Invalid input
  - ▶ Does a move that is not on the board do nothing?
  - ▶ Does a move do nothing if the position is full?
  - ▶ Does the method return false in these cases?

## Another Look

```
public boolean makeMove(Player player, Move move) {  
  
    int row = move.row;  
    int col = move.col;  
  
    if (row < 0 || row >= SIZE || col < 0 || col >= SIZE ||  
        !board[row][col].equals(EMPTY)) {  
        return false;  
    }  
  
    board[row][col] = player.getCharacter();  
  
    return true;  
}
```

How could we *better abstract* this code to make testing easier?

# Creating Units

```
public boolean makeMove(Player player, Move move) {  
  
    int row = move.row;  
    int col = move.col;  
  
    if (row < 0 || row >= SIZE || col < 0 || col >= SIZE ||  
        !board[row][col].equals(EMPTY)) {  
        return false;  
    }  
  
    board[row][col] = player.getCharacter();  
  
    return true;  
}
```



# Creating Units

```
public boolean cellIsEmpty(Move move) {  
    return board[move.row][move.col].equals(EMPTY);  
}
```

```
public boolean onBoard(Move move) {  
    return move.row >= 0 && move.row < SIZE &&  
        move.col >= 0 && move.col < SIZE;  
}
```

```
public boolean isValidMove(Move move) {  
    if (onBoard(move) && cellIsEmpty(move)) {  
        return true;  
    }  
    return false;  
}
```

```
public boolean makeMove(Player player, Move move) {  
    board[move.row][move.col] = player.getCharacter();  
}
```

# Unit Testing With Java

Much better! What now?

## Keyword

*Manual Testing:* Testing code manually, in an ad-hoc manner. Generally difficult to reach all edge cases, and not scalable for large projects.

## Keyword

*Automated Testing:* Testing code with automated, purpose built software. Generally faster, more reliable, and less reliant on humans.

# JUnit Automated Testing

## Keyword

*assert*: A true or false statement that indicates the success or failure of a test case.

## Keyword

*TestCase class*: A class dedicated to testing a single unit.

## Keyword

*TestRunner class*: A class dedicated to *executing* the tests on a unit.

# TestCase Class

```
import static org.junit.Assert.*;
import org.junit.Test;

public class BoardTest {
    @Test
    public void testBoard() {
        Board board = new Board();
        assertEquals(board.cellIsEmpty(0, 0), true);
    }

    @Test
    public void testValidMove() {
        Board board = new Board();
        Move move = new Move(0, 0);
        assertEquals(board.isValidMove(move), true);
    }
}
```

# TestCase Class

```
@Test
public void testMakeMove() {
    Board board = new Board();
    Player player = new HumanPlayer("R");
    Move move = new Move(0, 0);
    board.makeMove(player, move);
    assertEquals(board.getBoard()[move.row][move.col], "r");
}
```

# TestRunner Class

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;



public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(BoardTest.class);

        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }





        System.out.println(result.wasSuccessful());
    }
}
```

# TestRunner Class

Finished after 0.075 seconds

Runs: 3/3    Errors: 0    Failures: 1



 BoardTest [Runner: JUnit 4] (0.054 s)  
     testBoard (0.000 s)  
     testValidMove (0.000 s)  
     testMakeMove (0.054 s)

```
testMakeMove(BoardTest): expected:<[r]> but was:<[R]>  
false
```

# JUnit Automated Testing

Woops, there was a bug *in my test*

```
@Test
public void testMakeMove() {
    Board board = new Board();
    Player player = new HumanPlayer("R");
    Move move = new Move(0, 0);
    board.makeMove(player, move);
    assertEquals("R", board.getBoard()[move.row][move.col]);
}
```

Automated testing is as useful for testing your *test suite* as it is for testing your *program*.



# Assess Yourself

Write a unit test to verify that when a move is made **off the board**, the `isValidMove` method returns false.

There are actually (at least) **four** test cases for this, but here's one:

```
@Test
public void testValidMove2() {
    Board board = new Board();
    Move move = new Move(-1, 0);
    assertEquals(false, board.isValidMove(move));
}
```

# JUnit Advantages

Large teams and open source development **(should) always** use automated testing:

- Easy to set up
- Scalable
- Repeatable
- Not human intensive
- Incredibly powerful
- **Finds bugs**

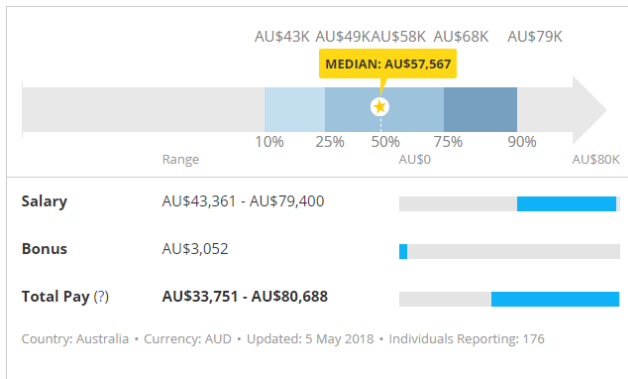
We don't expect you to use it, but getting used to automated testing makes you more useful in a team.

Here's [an example](#).

# Assess Yourself

What units, use cases, and unit tests can you write for Project 2B?

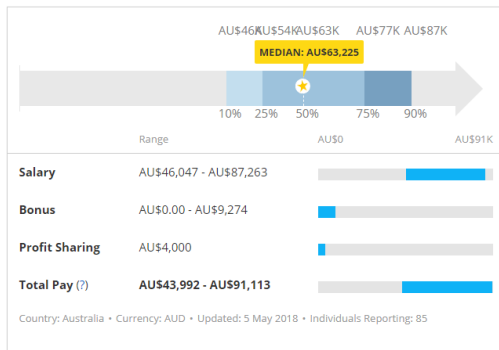
# Software Testing and QA Jobs



## Keyword

**Software Tester:** Conducts tests on software, primarily to find and eliminate bugs.

# Software Testing and QA Jobs



## Keyword

*Software Quality Assurance:* Actively works to improve the development process/lifecycle. Directs software testers to conduct tests, primarily to prevent bugs.

# Metrics

## Documentation

This will be assessed in the project, but not the exam.

## Software Design

You will need to be able to define the *keywords* defined in this lecture. You will need to know *definitions* for the exam, but you will not be assessed on software design principles by writing code.

## Software Testing

You will need to be able to define the keywords as well as implement a *unit test* for the exam. You will **not** be asked to write a `TestRunner` class, only one or two standalone test cases.