Software Development 3 (F27SG)

Lecture 1

# Introduction & (J)Unit testing

Rob Stewart

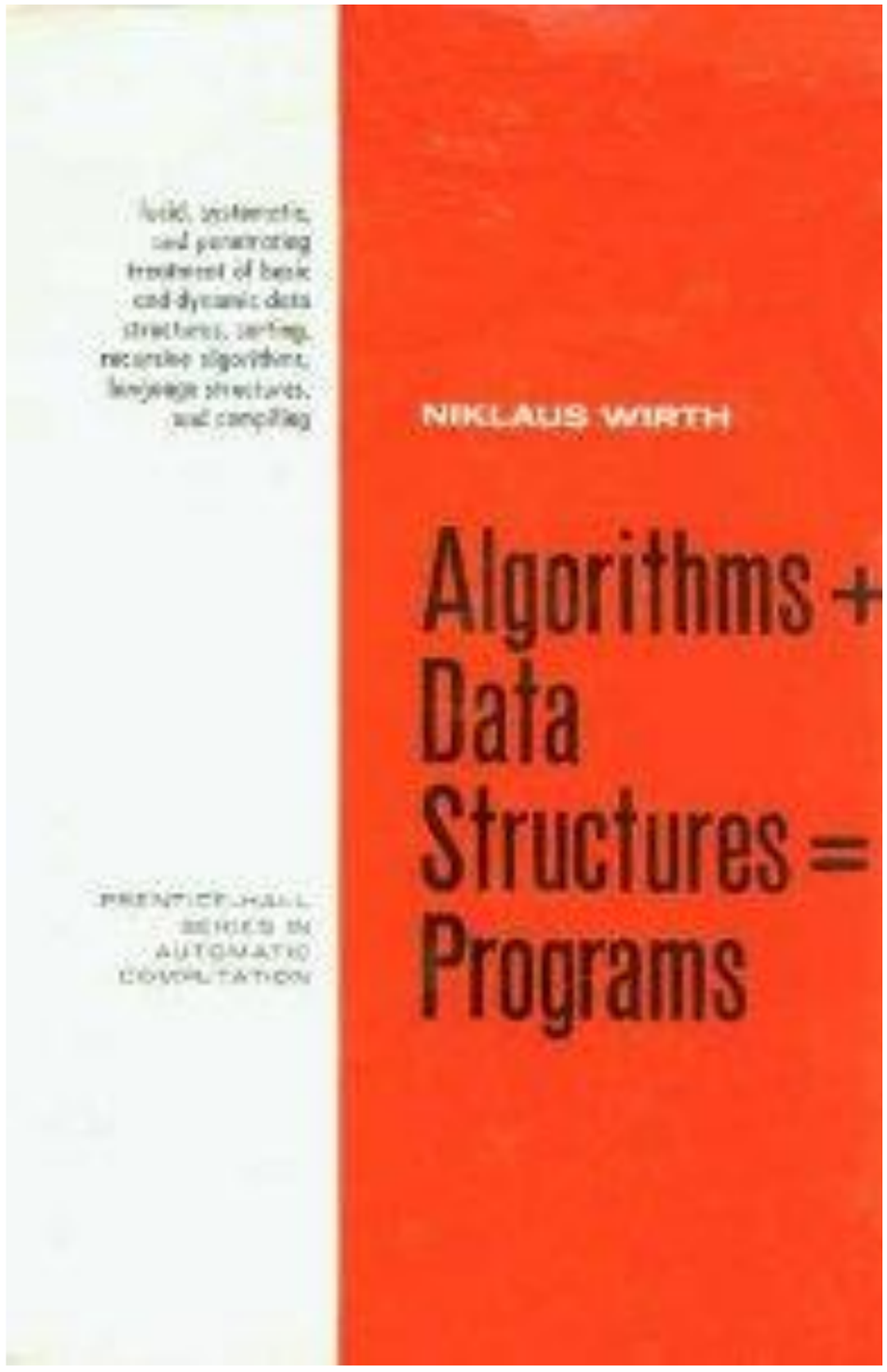
# About me…

- Rob Stewart, Assistant Professor
- Research interests
  - **Languages**: Parallel functional programming languages
  - **Architectures**: High Performance and embedded embedded systems
- Email: R.Stewart@hw.ac.uk
- Office: EM G.56
- Office hours: Tuesdays 14:15-15:15
- All materials will be on VISION
  - GitLab screencasts + PDFs
  - Lab sheets
  - Lecture slides
  - Lecture capture

# Topics

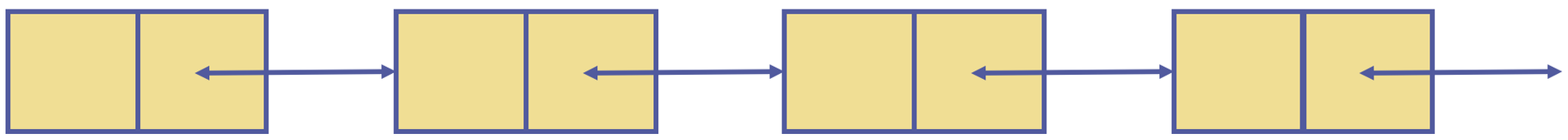
- You will learn how to write **efficient programs**
  - in particular when working with large data.
- This involves:
  - using the right **data structures** to represent data
  - implementation of **algorithms** over data structures
  - **analyse** and **compare** data structures and algorithms
  - use of different **programming techniques**
  - how to work with data in files

# This course is important!



# Data Structures

- In SD1 you used *objects* to store information
- In SD3: we will implement ***data structures*** to store **collections** of such objects, including
  - *static data structures*
    - can hold a fixed number of elements
  - *dynamic data structures*
    - can adjust the number of elements
  - *linked data structures* (dynamic)
    - special “wrapping objects” with references to others objects



# Algorithms

- We will implement algorithms **over** data structures
- An **algorithm** is a step-by-step procedure for performing some task
  - represented in a computer program or method
  - using underlying data structures
- You will learn about:
  - **linear techniques**
  - **recursive techniques**

# Importance of Algorithms

- Examples
  - **Google**'s **page rank** algorithm to search web pages
  - **Netflix** uses an algorithm called **pragmatic chaos** to decide which movies to recommend for you
- See **TED** talk by *Kevin Slavin* about algorithms

https://www.ted.com/talks/kevin_slavin_how_algorithms_shape_our_world?language=en

Ideas worth spreading

# Analysis

- It is important to be able to **analyse** and **compare** data structures and algorithms
  - the **time and space** usage required
  - Algorithm runtime: 1 minute? 1 hour?
  - Required memory: 1MB? 4GB?
- We will introduce **time** complexity
  - using a notation called **big-Oh**
  - Describe worst case increase of runtime
    - 100 data items: 1 minute
    - 500 data items: ?

# The use of Java

- We will use **Java** in this course ...
- ... most topics are **independent** of Java
- Many things you will learn to implement are already present in the large libraries available for Java
- .. **as a Computer Science it is important that you know how to implement these**

*[an exception is the last lectures which are Java-specific]*

# Resources

- We will **not use** a particular text book
- I would recommend:
  - Michael T. Goodrich & Roberto Tamassia. *Data Structures and Algorithms in Java (5th Edition)*
- Another option is:
  - Clifford A. Shaffer. *Data Structures and Algorithm Analysis (2013)*
  - Freely available online: http://people.cs.vt.edu/shaffer/Book/
- LinkedIn Learning: https://www.linkedin.com/learning
  - *Java Essential Training for Students*
  - Access is free using your Heriot-Watt University credentials

# Timetable

- **Lectures**
    - Thursday 09:15-10:15 (James Watt 2 Auditorium)
    - Friday 12:15-13:15 (James Watt 2 Auditorium)
- **Lab**
    - Group 1 (Last name starts with: A-H) *and GA students*
        - Friday 13:15-15:15 in EM 2.50/2.45 (2 hours)
    - Group 2 (Last name starts with: I-Z)
        - Monday 15:15 17:15 in EM 2.50/2.45 (2 hours)

# In the lectures we will…

- Cover all the material
  - Concepts
  - Code
- Work on exercises
  - We will not have separate tutorials
- Do lots of live coding

# In the lab you will…

- Apply what we have covered in the lectures

- It is crucial that you do the labs
  - This is the only way to learn it!
  - … and is the reason it is worth 50% of mark

**You need to go to lectures to do the labs!**

# Industry Readiness

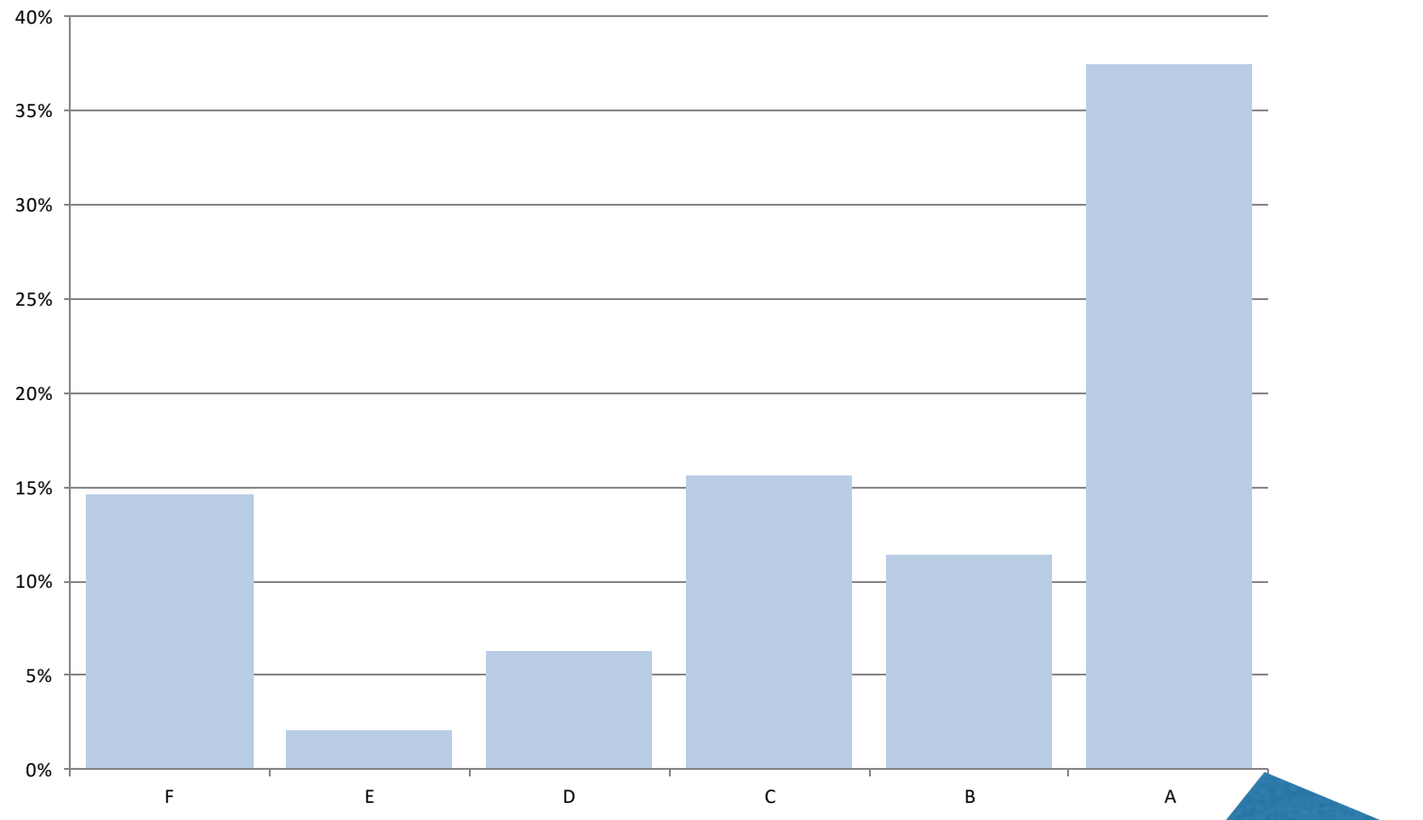
SD3 adopts widely used **industry best practise**

1. Test Driven Development
   - Tests first, implementation after
2. Automated software testing with GitLab Continuous Integration (CI) pipelines
   - only commits that pass all CI tests can be deployed, e.g. Amazon web services
3. Code reviews
   - Code almost never deployed after one iteration
   - Sometimes many reviewers per commit
   - To spot inefficient algorithms to satisfy requirements

https://gitlab.haskell.org/ghc/ghc

# It is important to attend

Results from 2017:




these attended!

# Lecture plan

| Week | Topics | |
|---|---|---|
| 1 | Unit testing and IO | |
| 2 | Complexity and stacks | |
| 3 | Recursion and linked lists | |
| 4 | Linked lists and dynamic stacks | |
| 5 | Queues and doubly linked lists | |
| 6 | **READING WEEK** | |
| 7 | Search and binary search trees | |
| 8 | Implementation & priority queues | |
| 9 | Tries & sorting | |
| 10 | Sorting & advanced Java | |
| 11 | Security | |
| 12 | Revision | |

# Lab deadlines

You will have one week to complete each lab

| Week | Group 1 (Friday) | Group 2 (Monday) |
|---|---|---|
| 1 | Unit testing/IO | |
| 2 | Stacks | Unit testing/IO |
| 3 | Recursion | Stacks |
| 4 | Linked lists / stacks | Recursion |
| 5 | Queues / doubly linked lists | Linked lists / stacks |
| 6 | | Queues / doubly linked lists |
| 7 | Search | |
| 8 | Trees | Search |
| 9 | Priority queues / tries | Trees |
| 10 | Sorting | Priority queues / tries |
| 11 | | Sorting |

# Feedback

- **You to me**
  - I will use the labs and exercises to adjust the plan
  - Let me know about things that you struggle with
  - Stop/start/continue half-way
  - Course evaluation form on Vision at the end
- **Me to you**
  - Formative feedback on code in the labs
  - Office hour
  - During lectures
- **You to you**
  - Peer feedback as code reviews on GitLab

# Handouts/resources

- All resources will be on **Vision**
- Lab sheets and lab code all available now
- Work through lab sheets in your own time
- Complete lab sheets in the lab and demonstrate
- GitLab always available

# This lecture

- SD3 takes the Test Driven Development (TDD) approach
- This will be achieved by using a framework for Java called *Junit*
- The rest of this lecture will give a brief introduction to Junit
- You will also use JUnit in SD2 in a few weeks

# Unit testing

- Unit testing is concerned with the low-level structure of program code.
- For Java a unit could refer to
  - A method
  - An object of a class
- Unit tests aim to determine whether the code achieves its associated specification
- Validating each unit reduces errors when integrating the pieces together later
  - Integration testing methods would then be used

# Test Driven Development

TDD in the labs

1. Specify the functional specification in terms of tests
2. Write the test before the code
3. Tests will fail
4. Implement the functionality
5. Keep implementing until a test passes
6. Create git commits and push to GitLab
7. Once all tests pass, clean up and document your code
8. Test again
9. If all tests still pass, create a final commit and push to GitLab

# JUnit

- Junit is a unit testing framework for Java
- Allows you to write unit tests in Java using a simple interface
- Automated testing enables running and rerunning tests very easily and quickly
- Easier that manually creating tests
  - E.g. use System.out.println()
  - Implement concepts of success/failure

# Junit Demo

**Double.java**

```
public class Double {
  public static int d1(int i){
      return 2*i;
   }
}
```

**DoubleTest.java**

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

public class DoubleTest {

   @Test
   public void testD1() {
    assertEquals("This is a message", 10, Double.d1(5));
   }
}
```

# Junit Demo

separate file for tests

DoubleTest.java

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

public class DoubleTest {

   @Test
   public void testD1() {
    assertEquals("This is a message", 10, Double.d1(5));
   }
}
```

# Junit Demo

import Junit library

**DoubleTest.java**

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

public class DoubleTest {

   @Test
   public void testD1() {
    assertEquals("This is a message", 10, Double.d1(5));
   }
}
```

# Junit Demo

**annotation** specifying that this is a test

**DoubleTest.java**

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

public class DoubleTest {

   @Test
   public void testD1() {
    assertEquals("This is a message", 10, Double.d1(5));
   }
}
```

# Junit Demo

test **assertion**

[*Double.d1(5) = 10*]

**DoubleTest.java**

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

public class DoubleTest {

   @Test
   public void testD1() {
    assertEquals("This is a message", 10, Double.d1(5));
   }
}
```

# Junit Eclipse demo

# Exercise

Abs.java

```
public class Abs {
  public static int abs(int i){
   if (i >= 0)
     return i;
   else
     return -i;
}}
```

Which of these jUnit tests will succeed?

- ***assertEquals(1, Abs.abs(1));***
- ***assertEquals(-1, Abs.abs(-1));***

If you have a internet access (phone/tablet/laptop)
Go to: www.socrative.com - `student login'
Room: SD32019
Select TRUE/FALSE for each assertion there

*[we will use Socrative throughout the course]*

# Exercise

Abs.java

```
public class Abs {
  public static int abs(int i){
   if (i >= 0)
     return i;
   else
     return -i;
}}
```

Which of these jUnit tests will succeed?

- ***assertEquals(1, Abs.abs(1));***
- ***assertEquals(-1, Abs.abs(-1));***
- ***assertEquals(-1, Abs.abs(1));***
- ***assertEquals(1, Abs.abs(-1));***

# Junit assertions

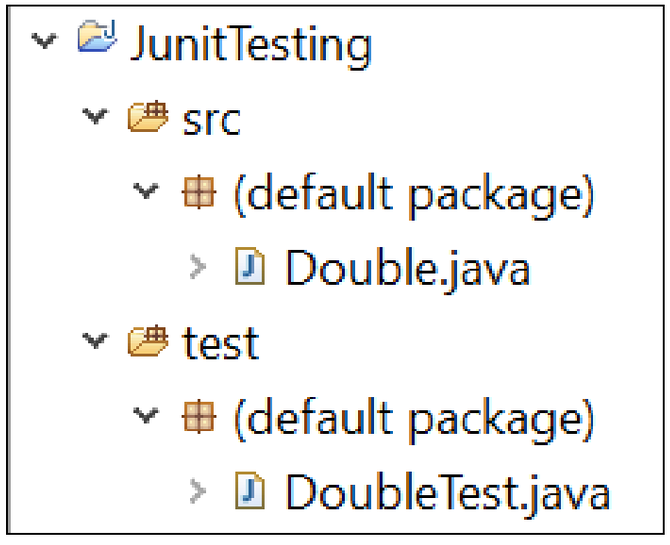
- We have seen
  **assertEquals(expected,actual)**
  - succeeds if expected and actual values are equal
  - optional string as a first argument
    - message that will be printed if the assertion fails
- There many other assertions
  - Some are illustrated next
  - For full details see
    - http://junit.sourceforge.net/javadoc/org/junit/Assert.html
  - and
    - http://junit.org/

# Junit assertions

| Statement | Description |
|---|---|
| fail(message) | Always fails. Often used to check that certain code is not reached |
| assertTrue(cond)<br>assertFalse(cond) | Checks if given condition is true/false |
| assertEquals(expected,actual) | Checks if expected and actual values are equals |
| assertArrayEquals(expected,actual) | Checks if expected and actual arrays are equals (their content) |
| assertNull (object) / assertNotNull(object) | Checks if an object reference is null or not null |
| assertSame(expected,actual)<br>assertNotSame(expected,actual) | Checks if expected and actual object references are the same (we will discuss object references later in course) |

# Writing Junit tests in Eclipse

- Separate test files from the actual program
  - Ideally develop separate directory in Eclipse project




- Each test method should
  - Return **void**
  - Have **no arguments**

```
@Test
public void testD1() { … }
```

- Name of class with test should have a “Test” suffix
  - E.g. test class for Double becomes **DoubleTest**
- Name of test method should start with test
  - … and include name of method you are testing
  - E.g. **testMethod1**, **testSortedMethod1**, …

# Junit @nnotations

- Java provides a set of *annotation*
  - Starts with @ (e.g. @Override)
  - Metadata about program
    - Not part of the program itself
- Junit uses a set of *annotations* to mark and configure test methods

# Junit @nnotations

- We have already seen **@Test**

```
@Test
public void testD1()
{ … }
```

  - Indicates that the method (**testD1**) is a test method
    - JUnit will run all methods annotated with @Test!
- Junit uses several other annotations e.g.
  - **@Before** : method is executed before each test
  - **@After** : method is executed after each test

# Writing Good Junit tests

- Good unit tests
  - document what the unit **should do**
  - is a specification of a unit
- This includes tests that should
  - succeed – positive tests
  - fail – negative tests
- Example of types of tests (incomplete)
  - If the output of a method is correct
  - Boundary conditions
    - e.g. empty/full collections, …
  - Edge cases
    - Operation on last element of an array
    - Very long strings, very large integers, unicode characters
  - Force error conditions
    - e.g. null pointer, reading from file that does not exists, …

# Time for a space race!

If you have a internet access (phone/tablet/laptop)
Go to: www.socrative.com - `student login'
Room: SD32019

# Summary

- Introduced unit testing and the Junit framework for Java
- Illustrated how to develop simple unit tests
- tips for how to write test cases

Next lecture: File Input/Output (IO)