



Victoria University  
of Wellington, New Zealand  
*Te Whare Wananga o te  
Upoko o te Ika a Maui  
Aotearoa*



# SWEN221: Software Development 24: Review

David J. Pearce & Marco Servetto & Julian Mackay & Nicholas  
Cameron & James Noble

<sup>1</sup> Engineering and Computer Science, Victoria University

# EXAM: Mon 24<sup>th</sup> June @14:30

- Exam Rooms
  - HMLT103, KKLT301, COLT122, HULT323, KKLT303
- The questions
  - Exam is **TWO HOURS** long
  - 4 compulsory questions worth **30 marks** each
  - Should spend at most **30 minutes** on each question
  - Generally, each has **easy** bits, **medium hard** bits and **hard** bits
    - If you cannot do a part, do not waste time stressing about it
    - Go and find something easier!
- As usual, you should make sure you can do **past papers!**
  - Look under "Past Exams" on course homepage + library

**CALCULATORS ARE NOT ALLOWED!!!**

- Simple stuff you need to know:
  - What **variables** and **parameters** are and do
  - What **static** means
  - What an **iterator** is
  - What **for(Collection l : ls) { ... }** does, and for other looping constructs
  - What **++i** and **i++** mean and how they differ
  - What **new** does
  - The difference between **references** and **objects** and **primitives**
  - How to write **recursive** methods
  - How to write a **class**, **interface**, **abstract class**
  - How to use **extends** and **implements**
  - How to generate **control flow graph** from method
  - ...
- Interesting stuff you need to know:
  - How **Polymorphism** really works in Java
  - How **inheritance** and **subtyping** work together
  - How to use **Java Generics**, including type bounds
  - What **public/private/protected** do and don't do
  - How to write **JUnit tests**, and what **test coverage** is
  - How to use **Exceptions**
  - How to write **equals**, **hashCode** and **compareTo** methods
  - How to calculate **test coverage** (for given criteria)
  - What **inner** and **anonymous** classes are
  - What **reflection**, **serialisation** + **cloning** are
  - What **Lambdas**, **Streams** and **Optional** are
  - What **Garbage Collection** is
  - ...

## #2 - The **JU**nit 5 API

A range of assertion methods:

- `assertTrue(boolean)`
- `assertTrue(String message, boolean)`

And a whole lot more:

- `assertEquals(Object expect, Object actual)`
- `assertEquals(float expected, float actual, float delta)`
- `assertNull`, `assertNotNull`
- `assertTrue`, `assertFalse`
- `assertSame`, `assertNotSame`
- `fail()`, `fail(String message)`

# #3 - Debugging

- **Defect:** Error in code created by programmer
- **Infection:** Error in program state
- **Propagation:** - Bad program state leads to more bad states
- **Failure:** Program finally does something wrong

## #4 – Code Style

```
class Date {  
    int day;    // day field  
    int month; // month field  
    int year;   // year field  
  
    int nextDay() {    // next day method  
        int r = day + 1; // r is day + 1  
        return r;      // return r  
    }  
}
```

- What's wrong with this?

# #5 - Exceptions

- Unchecked Exceptions
  - Subclasses of `RuntimeException`
  - E.g. `NullPointerException` and `IndexOutOfBoundsException`
- Checked Exceptions
  - Subclasses of `Exception`, but not `RuntimeException`
  - e.g. `IOException`
  - Must be declared in a method's *throws clause*:
    - If it throws one, or doesn't catch one thrown by called method
    - Otherwise compile-time error

# #6 - Assertions

```
assert x!=null;
```

if x is null and assertions are enabled, then the semantic of the assertion is equivalent to simply

```
throw new AssertionError( );  
//Unchecked Exception
```

```
assert x!=null : "msg";
```

if x is null and assertions are enabled, then the semantic of the assertion is equivalent to simply

```
throw new AssertionError( "msg" );  
//Unchecked Exception
```



# #7 - Inheritance I

- For two classes/interfaces A and B:
  - if A **extends** B, or A **implements** B, then  $A <: B$

```
class Point { int xpos; int ypos; ... }  
class ColouredPoint extends Point { int colour; }  
  
void move(Point p, int dx, int dy) {  
    p.xpos += dx;  
    p.ypos += dy;  
}  
ColouredPoint cp = new ColouredPoint(...);  
move(cp,1,1);  
System.out.println("cp.xpos = " + cp.xpos);
```

Through p  
we cannot  
see "colour"  
but it is  
there!

- Therefore, in this case,  $\text{ColouredPoint} <: \text{Point}$
- Meaning we can use a `ColouredPoint` instead of a `Point`!

# #8 - Inheritance II

```
class A {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    ... // other operations  
}  
class B {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
    ... // other operations  
}
```



```
class C {  
    private int value;  
    public int add(int x) {  
        return value+x;  
    }  
}  
class A extends C {  
    ... // other operations  
}  
class B extends C {  
    ... // other operations  
}
```

# #9 – Polymorphism I

```
class Cat {  
    String whatAmI() {  
        return "I'm a Cat!";  
    }  
}  
  
class Kitten extends Cat {  
    String whatAmI() {  
        return "I'm a Kitten!";  
    }  
}
```

```
class NinjaKitten extends Kitten {  
    String isKickedBy(Kitten k) { return "Ouch!"; }  
}
```

```
Cat bob = new NinjaKitten();  
System.out.println("Bob: " + bob.whatAmI());
```

A) Bob: "I'm a Kitten!"

B) Bob: "Ouch!"

C) error

# #10 – Polymorphism II

- Allows to create a hierarchy of classes/interfaces, and to model our problem domain.
- Dynamic dispatch (overriding) ensures subclass can change behaviour as needed
- For example, method toString()
  - allows any possible object,
  - of any possible class,
  - included the one that still does not exist, to **decide** how to convert into a String

```

class Card {
    private int number, suit;

    public Card(int n, int s) { number = n; suit = s; }

    public boolean equals(Object o) {
        if(o instanceof Card) {
            Card c = (Card) o;
            return c.number == number && c.suit == suit;
        }
        return false;
    }

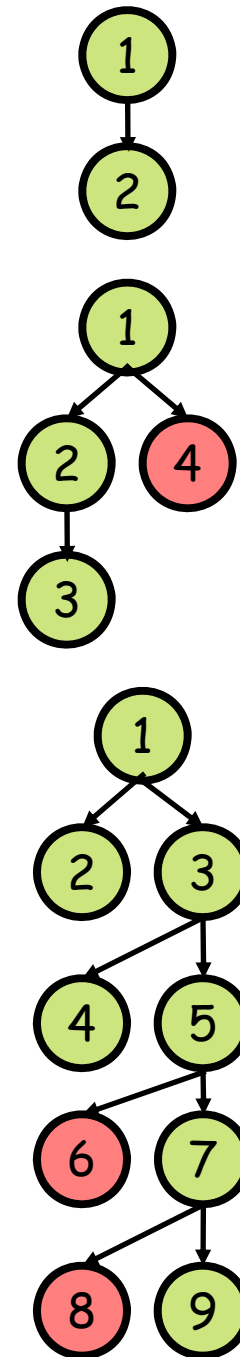
    public int compareTo(Card c) {
        if(suit > c.suit) { return -1; }
        else if(suit < c.suit) { return 1; }
        else if(number < c.number) { return -1; }
        else if(number > c.number) { return 1; }
        else { return 0; }
    }
}

```

Method Coverage = 3 / 3 = 100%

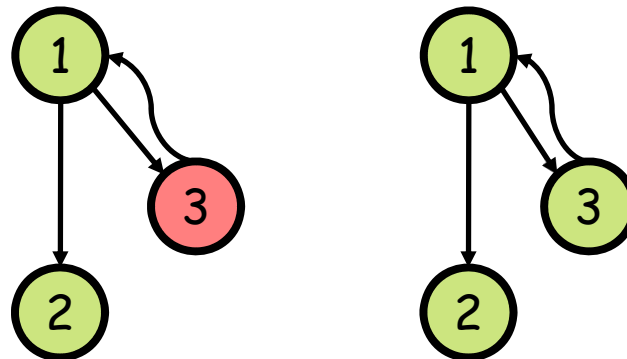
Statement Coverage = 12 / 15 = 80%

Branch Coverage = 2 / 5 = 40%



# #12 – Testing III

**Definition:** A **simple execution path** is a path through the method which iterates each loop at most once.



- Simple Path Coverage Criteria:
  - Aim to test all simple paths through a method
  - Helps keep the number of tests manageable
  - Two paths in above loop example

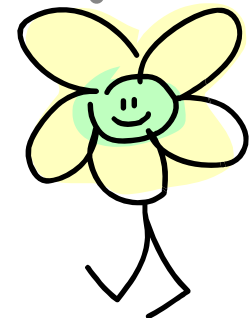
# #13 - Encapsulation

```
class Money {  
public int dollars;  
    public int cents; // cents < 100 must always hold  
    ...  
}
```

---

```
class Account {  
    int balance; // in cents  
    ...  
    void deposit(Money m) {  
        balance += (m.dollars*100) + m.cents;  
    }  
    Money getBalance() {  
        Money r = new Money();  
        r.dollars = 0;  
        r.cents = balance;  
        return r;  
    }  
}
```

Doesn't  
work now



# #14 – Generics I

```
class Vec<T> {  
    private Object[] elems = new Object[16];  
    private int end = 0;  
    public void add(T e) {  
        if(end == elems.length) { ... }  
        elems[end] = e; end=end+1;  
    }  
    public T get(int index) {  
        if(index >= end) { throw ... }  
        else return (T) elems[index];  
    }  
}
```

“T” is a  
generic  
parameter

“T” represents  
the type of  
object held in  
Vec

This says v is a  
Vec of Cats

Can only put  
Cats into v

```
Vec<Cat> v = new Vec<Cat>();  
v.add(new Cat());  
Cat c = v.get(0); // don't have to cast :-)
```

Can only get  
Cats out of v



# #15 - Generics II

```
class Cup<T> {  
    T f;  
    Cup(T f) {  
        this.f = f;  
    }  
}
```



**Cup<Tea>**

```
Cup<Tea> myCup = new Cup<Tea>(new Tea());
```

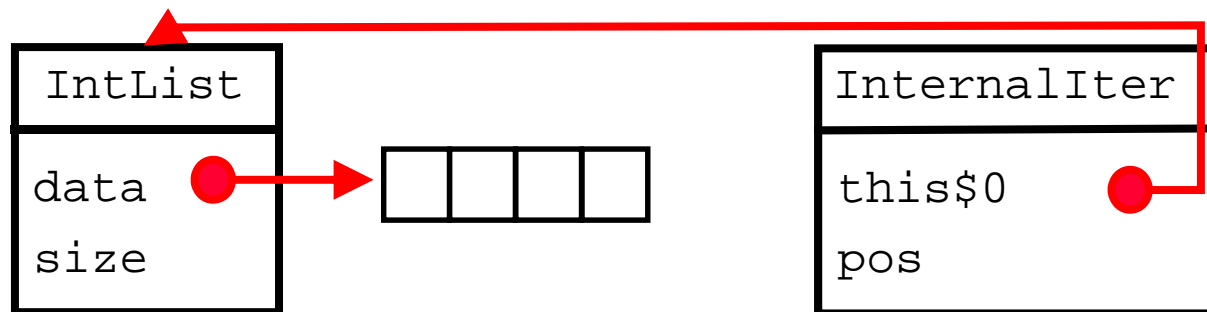
```
Cup<?> myCup = new Cup<Tea>(new Tea());
```

# #16 - Reflection

- Reflection gives access to *metadata*
  - That is, data about data
  - In this case, the data describes our classes
  - We can find out:
    - What an object's class is
    - What methods that class has (inc. private + protected)
    - What their parameter / return types are
    - What fields that class has (inc. private + protected)
    - What their types are
    - What interfaces the class implements
    - What class it extends from

# #17 - Inner Classes

- Inner classes have *parent pointer*
  - For accessing fields/methods of enclosing class (parent)
  - Parent pointer automatically supplied for new inner class



```
public class IntList implements Iterable<Integer> {  
    private int[] data;  
    private int size;  
  
    private class InternalIter implements Iterator<Integer> {  
        private int pos;  
    }  
}
```

# #18 – Object Contracts

- Need to override `Object.equals`
- Trickier than it sounds:
  - "It is **reflexive**: for any non-null reference value `x`, `x.equals(x)` should return `true`."
  - "It is **symmetric**: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`."
  - "It is **transitive**: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`."
  - "It is **consistent**: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified."
  - "For any non-null reference value `x`, `x.equals(null)` should return `false`."

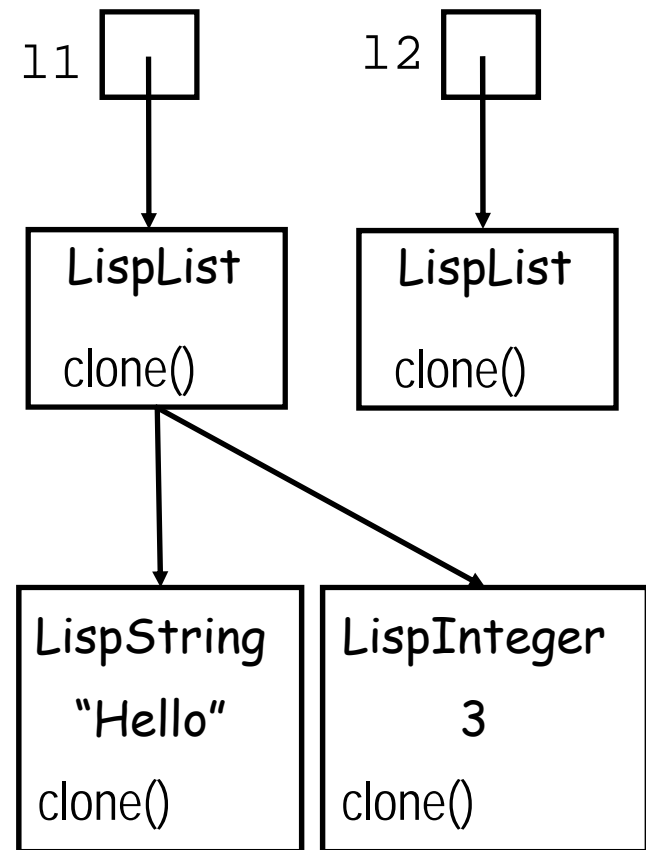
# #19 – Serialisation + Cloning

```
class LispList implements cloneable {
    private List<ListExpr> elements =
        new ArrayList<ListExpr>();

    ...
    public Object clone() {
        try { return super.clone(); }
        catch(CloneNotSupportedException e) {
            // cannot get here
        }
    }
}

LispInteger i = new LispInteger(3);
LispString s = new LispString("Hello");
LispList l1 = new LispList();
l1.add(i);
l1.add(s);

LispExpr l2 = (LispExpr) l1.clone();
```



- What does this actually do?

## #20 - Puzzlers

- How to check an integer is odd?

```
boolean isOdd(int x) {  
    return (x%2) == 1;  
}
```

- Does this method work?

A) Yes

B) No

C) Don't know

# #21 – Lambdas

## Comparators using long syntax for anonymous classes

```
Collections.sort(ls, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.compareToIgnoreCase(s2);  
    }  
});
```

## Comparators using short syntax for anonymous classes

```
Collections.sort(ls, (s1, s2) -> s1.compareToIgnoreCase(s2));
```

- Convenient syntax for anonymous nested classes (later)

# #22 – Optional & Streams

You can easily do that using streams:

```
List<Aeroplane> attempts=...
```

```
attempts.stream()//first, cache the fitness  
    .forEach(a->a.computeAverageFlightTime());
```

```
List<Aeroplane> best20 = attempts.stream()  
    .sorted((a1,a2)->a1.getFlightTime()-a2.getFlightTime())  
    .limit(20)//take the first 20  
    .collect(Collectors.toList());
```

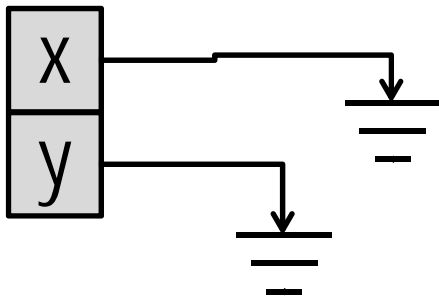
computeAverageFlightTime can be slow, you could need to do it for all your attempts!

- **Modern** hardware have multiple processors.

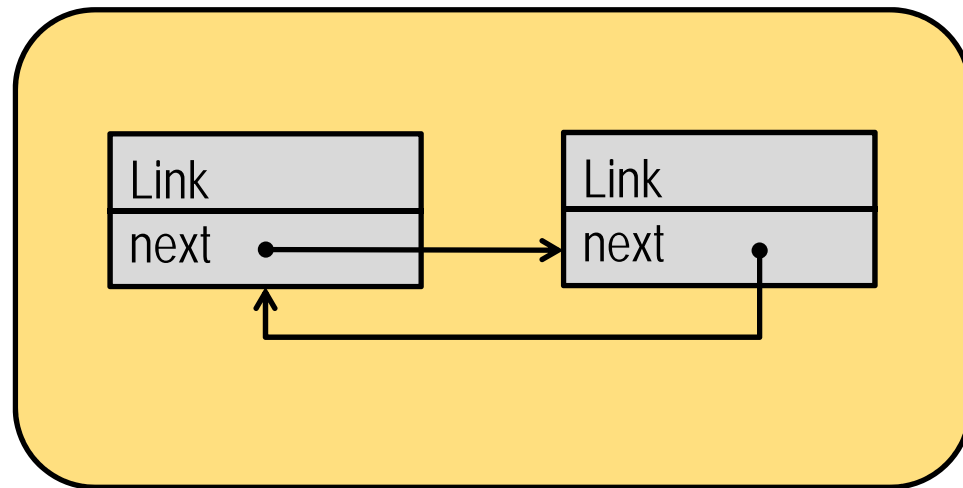


# #23 Garbage Collection

```
class Link {  
    private Link next;  
    public Link(Link next) { this.next = next; }  
    public static void main(String[] args) {  
        Link x = new Link(null);  
        Link y = new Link(x);  
        x.next = y;  
        x = null;  
        y = null;  
    }  
}
```



Call Stack



Heap



That's all folks ... Good Luck!!