TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2017

TRIMESTER 1

SWEN221

Software Development

Time Allowed: TWO HOURS

CLOSED BOOK

Permitted materials: No calculators permitted.
Non-electronic Foreign language to English dictionaries are allowed.

Instructions: Answer all questions
All questions are of equal value

Answer all questions in the boxes provided.
Every box requires an answer.
If additional space is required you may use a separate answer booklet.

Question	Topic	Marks
1.	Programming	30
2.	Inheritance and Polymorphism	30
3.	Testing	20
4.	Generics	30
5.	Java 8	10
	Total	120

1. Programming **(30 marks)**

(a) **(5 marks)** For each of the following groups of statements, clearly indicate the one statement that is true:

i. Generally, code conforming to professional style guidelines. . .

1. does not need comments.
2. needs some comments.
3. needs comments for every class and every method.
4. needs comments on every line of code.

ANSWER: 3

ii. Testing can . . .

1. demonstrate the absence of bugs.
2. prove that the code is robust.
3. show the presence of bugs.
4. prove that the code meets the specification.

ANSWER: 3

iii. Debugging is the process of . . .

1. finding and eliminating defects.
2. eliminating bugs.
3. finding and eliminating failures.
4. finding and eliminating infections.

ANSWER: 1

iv. Reproducing bugs . . .

1. is the job of the tester.
2. is a straightforward task during debugging.
3. can be difficult during debugging.
4. always results in new test cases.

ANSWER: 1 *or* 4

v. Which of the following is true?

1. Every defect causes a failure.
2. Every failure causes a defect.
3. Every failure is caused by a defect.
4. Every defect is caused by a failure.

ANSWER: 3

Consider the following classes and interfaces, which compile without error:

```
1  // A variable holding a boolean value
2  class ExprVar {
3    private boolean value;
4
5    public ExprVar(boolean value) { this.value = value; }
6
7    public boolean get() { return value; }
8
9    public void set(boolean value) { this.value = value; }
10 }
11
12 // An expression node has two operands and one output
13 abstract class ExpressionNode {
14   private ExprVar[] variables = new ExprVar[3];
15
16   public ExpressionNode(ExprVar op1, ExprVar op2, ExprVar out) {
17     variables[0] = op1;
18     variables[1] = op2;
19     variables[2] = out;
20   }
21   public void evaluate() {
22     boolean op1 = variables[0].get();
23     boolean op2 = variables[1].get();
24     variables[2].set(evaluate(op1, op2));
25   }
26   public abstract boolean evaluate(boolean op1, boolean op2);
27 }
28
29 // If both operands are true, output is true; otherwise, output
   is false.
30 class AndNode extends ExpressionNode {
31   public AndNode(ExprVar v1, ExprVar v2, ExprVar v3) {
32       super(v1,v2,v3);
33   }
34   public boolean evaluate(boolean op1, boolean op2) {
35       return op1 && op2;
36 } }
37
38 // If either input is true, output is true; otherwise, output is
   false.
39 class OrNode extends ExpressionNode {
40   public OrNode(ExprVar v1, ExprVar v2, ExprVar v3) {
41       super(v1,v2,v3);
42   }
43   public boolean evaluate(boolean op1, boolean op2) {
44       return op1 || op2;
45 } }
```

(b) Based on the code given on page 3, state the output you would expect for each of the following code snippets:

i. **(2 marks)**

```
1      ExprVar v1 = new ExprVar(true);
2      System.out.println(v1.get());
```

true

ii. **(2 marks)**

```
1      ExprVar v1 = new ExprVar(false);
2      ExprVar v2 = new ExprVar(true);
3      ExprVar v3 = new ExprVar(true);
4      ExpressionNode node = new AndNode(v1,v2,v3);
5      node.evaluate();
6      System.out.println(v1.get() + "␣" + v2.get() + "␣" + v3.get());
```

false true false

iii. **(2 marks)**

```
1      ExprVar v1 = new ExprVar(true);
2      ExprVar v2 = new ExprVar(false);
3      ExpressionNode node = new OrNode(v1,v2,v2);
4      node.evaluate();
5      System.out.println(v1.get() + "␣" + v2.get());
```

true true

iv. **(2 marks)**

```
1      ExprVar v1 = new ExprVar(true);
2      ExprVar v2 = new ExprVar(false);
3      ExprVar v3 = new ExprVar(false);
4      ExpressionNode node1 = new OrNode(v1, v2, v3);
5      ExpressionNode node2 = new AndNode(v3, v2, v1);
6      node1.evaluate();
7      node2.evaluate();
8      System.out.println(v1.get() + "␣" + v2.get() + "␣" + v3.get());
```

false false true

(c) **(7 marks)** Provide an implementation for a class `XorExpr`. This sets the `out` field to **`true`** if exactly one input is **`true`** (i.e. not both); otherwise, it sets it to **`false`**.

```
1 class XorExpr extends ExpressionNode {
2  public XorExpr(ExprVar op1, ExprVar op2, ExprVar out) {
3    super(op1, op2, out);
4  }
5
6  public boolean evaluate(boolean in1, boolean in2) {
7    return in1 != in2;
8  }
9 }
```

(d) **(10 marks)** Implement method `equals` for class `ExpressionNode`. Ensure that the standard requirements for `equals()` are met and that expressions of the same type that have the same variable values are considered equal.

```
public boolean equals(Object o) {









}
```

```
1  public boolean equals(Object o) {
2   if(o != null && this.getClass().equals(o.getClass())) {
3     ExpressionNode en = (ExpressionNode) o;
4     for(int i=0;i<variables.length;i++) {
5        if(variables[i].get() != en.variables[i].get()) {
6          return false;
7        }
8     }
9     return true;
10  }
11  return false;
12 }
```

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

2. Inheritance and Polymorphism **(30 marks)**

Consider the following Java classes and interfaces (which compile without error).

```
1  public abstract class Node {
2    public abstract int traverse(Traverser t);
3  }
4
5  public class NodeLeaf extends Node {
6     public int traverse(Traverser t) { return t.traverse(this); }
7  }
8
9  public class InnerNode extends Node {
10   public final Node next;
11
12   public InnerNode(Node next) {
13      this.next = next;
14    }
15
16   public int traverse(Traverser t) { return t.traverse(this); }
17 }
18
19 public interface Traverser {
20    public int traverse(NodeLeaf l);
21    public int traverse(InnerNode n);
22 }
```

(a) Given the above declarations, state whether the following classes compile without error. For any which do not compile, briefly state the problem.

i. **(2 marks)**

```
1  public class ATraverser implements Traverser {
2    public int traverse(LeafNode n) { return 1; }
3  }
```

Does not compile because `ATraverser` must implement `traverse(NodeLeaf)`

ii. **(2 marks)**

```
1 public class BTraverser implements Traverser {
2   public int traverse(Node n) { return 1; }
3 }
```

Does not compile, because `BTraverser` must implement `traverse(InnerNode)` (note: the given `traverse(Node)` is not sufficient because it overloads the required method)

iii. **(2 marks)**

```
1 public abstract class CTraverser implements Traverser {
2   public int traverse(NodeLeaf l) { return 1; }
3   public abstract int traverse(InnerNode n);
4 }
```

Compiles Ok, because `CTraverser` is declared **`abstract`**

(b) **(3 marks)** The `Node` class is declared **`abstract`**. Briefly, discuss what this means.

An **`abstract`** class cannot be instantiated and may contain **`abstract`** methods (i.e. methods without bodies which must be implemented by concrete subclasses). Abstract classes are similar to interfaces, but may contain method implementations and define fields.

(c) **(3 marks)** The `InnerNode.next` is declared **`final`**. Briefly, explain what this guarantees and what it does not guarantee.

This means that, once defined, those fields cannot have their value changed. Note, however, that the objects they refer to can still be changed.

Consider the following implementation of `Traverser` which compiles without error.

```
1 public class DTraverser implements Traverser {
2   public int traverse(NodeLeaf l) { return 0; }
3   public int traverse(InnerNode n) {
4     return 1+n.next.traverse(this);
5   }
6 }
```

(d) Give the output obtained from executing each code snippet belot.

i. **(3 marks)**

```
1 int c = new DTraverser().traverse(new NodeLeaf());
2 System.out.println(c);
```

```
0
```

ii. **(3 marks)**

```
1 int c = new NodeLeaf().traverse(new DTraverser());
2 System.out.println(c);
```

```
0
```

iii. **(3 marks)**

```
1 NodeLeaf t1 = new NodeLeaf();
2 InnerNode t2 = new InnerNode(t1);
3 int c = new DTraverser().traverse(t2);
4 System.out.println(c);
```

```
1
```

(e) **(4 marks)** In your own words, describe what the `DTraverser` does.

It counts the number of `InnerNodes` from a given `Node`.

(f) **(5 marks)** Give a `Traverser` implementation which returns 0 when the number of nodes is even and 1 when the number of nodes is odd.

```
1
2 public class EvenOddTraverser implements Traverser {
3  public int traverse(NodeLeaf l) { return 1; }
4  public int traverse(InnerNode n) {
5    int c = n.next.traverse(this);
6    return c == 0 ? 1 : 0;
7   }
8 }
```

3. Testing **(20 marks)**

(a) **(3 marks)** For each of the following characteristics, state whether it is the characteristic of *black-box testing* or *white-box testing*.

1. Test cases are generated directly from specification.

2. The aim is to reach a high-degree of code coverage.

3. Robust to implementation changes.

ANSWER(1): Blackbox
ANSWER(2): Whitebox
ANSWER(3): Blackbox

(b) **(2 marks)** List *two* potential causes of programmer bias.

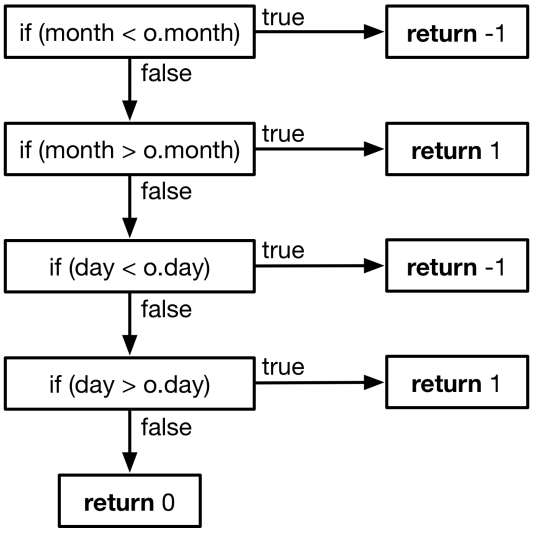
1. Misinterpretation of the specifications.
2. Wrong "belief" that a certain part is correct, and miss the test for it.
3. Programmer unlikely to represent target audience.

Consider the following classes, which compile without error:

```
1  public class Date {
2      private int month, day;
3
4      public Date(int m, int d) { this.month = m; this.day = d; }
5
6      public boolean equals(Object o) {
7          if(o instanceof Date) {
8              Date c = (Date) o;
9              if(c.month != month) { return false; }
10             if(c.day != day) { return false; }
11             return true;
12         }
13         return false;
14     }
15
16     public int compareTo(Date o) {
17         if(month < o.month) { return -1; }
18         if(month > o.month) { return 1; }
19         if(day < o.day) { return -1; }
20         if(day > o.day) { return 1; }
21         return 0;
22     }
23 }
```

```
1  public class DateTests {
2      @Test public void testEquals1() {
3          assertTrue(new Date(1,2).equals(new Date(1,2)));
4      }
5
6      @Test public void testEquals2() {
7          assertFalse(new Date(1,2).equals(new Date(3,4)));
8      }
9
10     @Test public void testCompare1() {
11         assertTrue(new Date(6,3).compareTo(new Date(2,1)) > 0);
12     }
13
14     @Test public void testCompare2() {
15         assertTrue(new Date(3,10).compareTo(new Date(5,3)) < 0);
16     }
17
18     @Test public void testCompare3() {
19         assertTrue(new Date(4,4).compareTo(new Date(4,4)) == 0);
20     }
21 }
```

(c) **(5 marks)** Draw the *control-flow graph* for the `Date.compareTo(Date)` method.



(d) There are different *coverage* criteria to measure the effectiveness of a test suite.

i. **(5 marks)** Calculate the total *statement coverage* of `Date` obtained by `DateTests`.

There are totally 19 statements in `Date`. 2 in the Constructor, 8 in `equals(Object)` and 9 in `compareTo(Date)`.
2 statements in the Constructor are covered.
In `equals(Object)`, the false side of the branch **`if`** `(o` **`instanceof`** `Date)` is not covered (1 statement). In the true side, the true sides of both if statements are covered. The false side of the second if statement is not covered (1 statement). In total there are 2 statements not covered in `equals(Object)`.
In `compareTo(Date)`, the false sides of all the if statements are covered. The true side of the last 2 if statements are not covered. There are 2 statements not covered in `compareTo(Date)`.
The total statement coverage is $(19-4)/19 = 0.79$.

ii. **(5 marks)** Calculate the total *branch coverage* of `Date` obtained by `DateTests`.

There are 7 branches in `Date`, 3 in `equals(Object)` and 4 in `compareTo(Date)`.
The branch **`if`**`(c.month != month)` in `equals(Object)` is completely covered.
The branches **`if`**`(month < o.month)` and **`if`**`(month > o.month)` are completely covered.
The total branch coverage is $3/7 = 0.43$.

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

4. Java Generics **(30 marks)**

Consider the following classes and interfaces, which compile without error:

```
1  public interface Fringe {
2      void add(int element);
3      int poll();
4  }
```

```
1  public class Stack implements Fringe {
2      private List<Integer> elements = new ArrayList<Integer>();
3
4      public void add(int element) { elements.add(element); }
5
6      public int poll() {
7          int e = elements.get(elements.size()-1);
8          elements.remove(elements.size()-1);
9          return e;
10     }
11 }
```

```
1  public class Queue implements Fringe {
2      private List<Integer> elements = new ArrayList<Integer>();
3
4      public void add(int element) { elements.add(element); }
5
6      public int poll() {
7          int e = elements.get(0);
8          elements.remove(0);
9          return e;
10     }
11 }
```

```
1  public class PriorityQueue implements Fringe {
2      private List<Integer> elements = new ArrayList<Integer>();
3
4      public void add(int element) { elements.add(element); }
5
6      public int poll() {
7          int pollIdx = 0;
8          for (int i = 1; i < elements.size(); i++) {
9              if (elements.get(i).compareTo(
10                         elements.get(pollIdx)) == -1) {
11                 pollIdx = i;
12             }
13         }
14         int e = elements.get(pollIdx);
15         elements.remove(pollIdx);
16         return e;
17     }
18 }
```

(a) **(3 marks)** The interface `Fringe` can only add and poll integer elements. By writing neatly in the box below, turn `Fringe` into a generic version `Fringe<T>` where `T` specifies the type of elements which can be added and polled.

```
1  public interface Fringe<T> {
2
3
4      void add(T element);
5
6
7      T poll();
8
9
10 }
```

(b) **(5 marks)** By writing neatly in the box below, turn `Stack` into a generic version `Stack<T>` where `T` specifies the type of elements which can be added and polled.

```
1  public class Stack<T> implements Fringe<T> {
2
3
4      private List<T> elements =
5          new ArrayList<T>();
6
7
8
9      public void add(T element) {
10
11
12         elements.add(element);
13
14
15     }
16
17     public T poll() {
18
19
20         T e = elements.get(elements.size()-1);
21
22
23         elements.remove(elements.size()-1);
24
25
26         return e;
27
28
29     }
30 }
```

(c) **(2 marks)** For turning `PriorityQueue` into a generic version, the following revision cannot compile. Explain the reason based on the code of `PriorityQueue` on page 17.

```
1 public class PriorityQueue<T> implements Fringe<T> {...}
```

There is a method `compareTo()` in `PriorityQueue`, which require `T` to implement `Comparable`. Without the upper bound of `Comparable` for `T`, the method `compareTo()` cannot be recognised.

(d) **(8 marks)** By writing neatly in the box below, turn `PriorityQueue` into a generic version.

```
1  public class PriorityQueue<T extends Comparable<T>>
2       implements Fringe<T> {
3
4      private List<T> elements =
5          new ArrayList<T>();
6
7
8      public void add(T element) {
9
10
11         elements.add(element);
12
13
14     }
15
16     public T poll() {
17
18
19         int pollIdx = 0;
20
21
22         for (int i = 1; i < elements.size(); i++) {
23
24
25             if (elements.get(i).compareTo(
26                     elements.get(pollIdx)) == -1) {
27
28
29                 pollIdx = i;
30
31
32         }}
33         T e = elements.get(pollIdx);
34
35
36         elements.remove(pollIdx);
37
38
39         return e;
40 }}
```

(e) **(3 marks)** State whether the following code compiles or not, and explain the reason(s).

```
1 List<Object> list = new ArrayList<Object>();
2 list.add("First");
3 List<String> strList = list;
```

It cannot compile. It will get the error of incompatible types between `List<Object>` and `List<String>`.

(f) Consider the following code:

```
1  interface Shape { void draw(); }
2  class Circle implements Shape { ... }
3  class Rectangle implements Shape { ... }
4  void drawAll(List<Shape> shapes) {
5      for (Shape shape : shapes) shape.draw();
6  }
7  public static void main (String[] args) {
8      List<Circle> circles = new ArrayList<Circle>();
9      List<Rectangle> recs = new ArrayList<Rectangle>();
10     drawAll(circles);
11     drawAll(recs);
12 }
```

i. **(2 marks)** The code cannot compile. Explain the reason.

`List<Shape>` is not applicable for arguments `List<Circle>` and `List<Rectangle>`.

ii. **(2 marks)** Rewrite the code for the `drawAll` method to make the code work.

```
void drawAll(List<? extends Shape> shapes) {
  for (Shape shape : shapes) shape.draw();
}
```

(g) **(5 marks)** Write a generic method `max(a,b)` that returns the maximal value between two values `a` and `b` of any *suitable* type.

```
1 <T extends Comparable<T>> T max(T a, T b) {
2     if (a.compareTo(b) == -1) return b;
3     else return a;
4 }
```

5. Java 8 **(10 marks)**

(a) **(2 marks)** List two properties of a *functional interface*.

- One and only one abstract method.
- Decorated with `@FunctionalInterface`.
- Can be represented as a lambda expression.

(b) **(2 marks)** State the difference between *default* methods and *static* methods in Java 8 interfaces.

We can override default methods, but cannot override static methods in the implementation classes.

(c) **(2 marks)** What is the output of the following code?

```
1 Optional<String> postcode1 = Optional.of("6001");
2 Optional<String> postcode2 = Optional.empty();
3
4 if (postcode1.isPresent()) System.out.println(postcode1.get());
5 if (postcode2.isPresent()) System.out.println(postcode2.get());
```

```
6001
```

(d) **(2 marks)** What is the output of the following code?

```
1  List<String> names =
2      Arrays.asList("allen", "clark", "john", "Alex", "Jack");
3
4  names = names.stream()
5          .map(s -> s.toLowerCase())
6          .filter(s -> s.startsWith("a"))
7          .collect(Collectors.toList());
8
9  for (String s : names) System.out.println(s);
```

```
allen
alex
```

(e) **(2 marks)** Consider the following code, write the stream version of lines 5–10.

```
1  class Person {
2    public String name;
3    public int age;
4  }
5  List<Person> persons = Arrays.asList(...);
6  List<String> selected = new ArrayList<String>();
7  for (Person p : persons) {
8    if (p.age < 18) selected.add(p.name.toLowerCase());
9  }
10 Collections.sort(selected);
```

```
List<String> selected = persons.stream()
    .filter(p -> p.age < 18)
    .map(p -> p.name.toLowerCase())
    .sorted()
    .collect(Collectors.toList());
```

* * * * * * * * * * * * * * *

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.