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

EXAMINATIONS – 2018

TRIMESTER 1

SWEN 221

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

You may answer the questions in any order. Make sure you clearly identify the question you are answering.

Question	Topic	Marks
1.	Code Comprehension	30
2.	Testing	30
3.	Lambdas and Streams	30
4.	Java Masterclass	30
	Total	120

1. Code Comprehension **(30 marks)**

Consider the following classes which compile without error:

```
1  public abstract class Square {
2    public enum Color { RED, GREEN, BLUE };
3
4    public abstract void flip();
5    public abstract Color get();
6  }
```

```
1
2  public class Blank extends Square {
3    public void flip() {}
4    public Color get() { return null; }
5  }
```

```
1
2  public class Door extends Square {
3    private final Color color;
4    private boolean open;
5
6    public Door(Color color) { this.color = color; }
7
8    public void flip() {
9      if (open) { open = false; }
10     else { open = true; }
11   }
12   public Color get() {
13     if (open) { return color; }
14     else { return null; }
15   }
16 }
```

```
1  public class Board {
2    private Square[] squares;
3
4    public Board(Square[] squares) { this.squares = squares; }
5
6    public void flip(int x) { squares[x].flip(); }
7
8    public int count(Square.Color color) {
9      int count = 0;
10     for (int x = 0; x != squares.length; ++x) {
11       Square.Color found = squares[x].get();
12       if (color == found) {
13         count = count + 1;
14       }
15     }
16     return count;
17   }
18   }
```

(a) Based on the code given on page 2, state the output you would expect for each of the following code snippets:

i. **(2 marks)**

```
1    Square[] squares = new Square[] { new Blank() };
2    Board board = new Board(squares);
3    System.out.println(board.count(Square.Color.BLUE));
```

0

ii. **(2 marks)**

```
1    Square[] squares = new Square[] {
2       new Blank(), new Door(Square.Color.BLUE)
3    };
4    Board board = new Board(squares);
5    System.out.println(board.count(Square.Color.BLUE));
```

0

iii. **(2 marks)**

```
1    Square[] squares = new Square[] {
2       new Blank(), new Door(Square.Color.BLUE)
3    };
4    Board board = new Board(squares);
5    board.flip(0);
6    System.out.println(board.count(Square.Color.BLUE));
```

0

iv. **(2 marks)**

```
1    Square[] squares = new Square[] {
2       new Blank(), new Door(Square.Color.BLUE)
3    };
4    Board board = new Board(squares);
5    board.flip(1);
6    System.out.println(board.count(Square.Color.BLUE));
```

1

v. **(2 marks)**

```
1     Square[] squares = new Square[] {
2        new Blank(), new Door(Square.Color.BLUE), new Blank()
3     };
4     Board board = new Board(squares);
5     board.flip(1);
6     System.out.println(board.count(null));
```

2

(b) **(3 marks)** Consider the method `Door.flip()`. Does it *overload* or *override* the method `Square.flip()`? Justify your answer.

`Door.flip()` overrides `Square.flip()`. This is because they have the same name, and the same parameter types. For a method to overload another, they would need the same name but *different* parameter types.

(c) **(4 marks)** Provide a subclass of `Door` where `flip()` only has an effect the first time it is called:

```
1  class SingleDoor extends Door {
2    private boolean first = true;
3
4    public SingleDoor(Color color) { super(color); }
5
6    public void flip() {
7      if(first) { super.flip(); first=false; }
8  } }
```

(d) **(5 marks)** In your own words, describe what the `Board.count()` function does:

It counts the number of squares in the board with a given colour. Blank squares can also be counted using **`null`** as the colour.

(e) Consider the following snippet of code:

```
1 Square square = new Door(Square.Color.BLUE);
```

i. **(4 marks)** The *static type* of variable `square` is `Square`. Briefly, discuss what this means.

The static type of a variable or field is its declared type. This limits the possible values for the variable `square` to the subtypes of `Square`. Only methods declared in `Square` (or its superclasses) can be called on `square`, even for subclasses with additional methods.

ii. **(4 marks)** The *dynamic type* of variable `square` is `Door`. Briefly, discuss what this means and how it affects the execution of method `square.flip()`.

The dynamic of a variable or field is its actual type at runtime. Since method `Square.flip()` is **`abstract`** and has no implementation, , the actual method which is executed is determined by the dynamic type. Thus, different subclasses can provide different behaviours for the `flip()` method.

SPARE PAGE FOR EXTRA ANSWERS

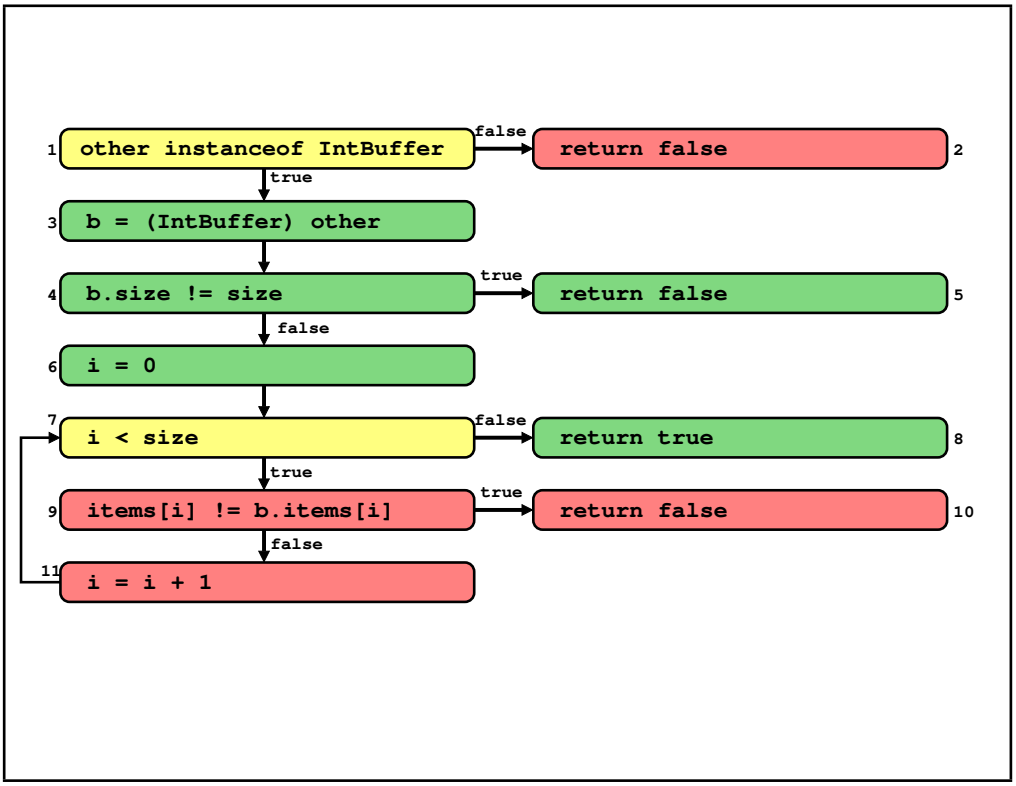
Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

2. Testing **(30 marks)**

(a) Consider the following class, which compiles without error:

```
1  public class IntBuffer {
2    private int[] items;
3    private int size;
4
5    public IntBuffer(int max) {
6      this.items = new int[max]; this.size = 0;
7    }
8
9    public void write(int item) { items[size++] = item; }
10
11   public int read() { return items[--size]; }
12
13   public boolean equals(Object other) {
14     if (other instanceof IntBuffer) {
15       IntBuffer b = (IntBuffer) other;
16       if (b.size != size) { return false; }
17       for (int i = 0; i < size; i = i + 1) {
18         if (items[i] != b.items[i]) { return false; }
19       }
20       return true;
21     } else {
22       return false;
23  } } }
```

i. **(8 marks)** Draw the *control-flow graph* for the `IntBuffer.equals(object)` method.



Consider the following test cases for the class `IntBuffer`:

```
1  public class IntBufferTests {
2    @Test public void testReadWrite() {
3      IntBuffer b = new IntBuffer(1);
4      b.write(0);
5      assertTrue(b.read() == 0);
6    }
7
8    @Test public void testEquals_1() {
9      IntBuffer b1 = new IntBuffer(1);
10     IntBuffer b2 = new IntBuffer(1);
11     assertTrue(b1.equals(b2));
12   }
13
14   @Test public void testEquals_2() {
15     IntBuffer b1 = new IntBuffer(1);
16     IntBuffer b2 = new IntBuffer(1);
17     b1.write(0);
18     assertFalse(b1.equals(b2));
19   }
```

ii. **(2 marks)** Give the total *statement coverage* obtained for class `IntBuffer` from the tests provided above.

4 statements from `IntBuffer(int)`, `write(int)`, `read()` all covered, and 7 statements from `equals(Object)` covered (see CFG). This gives 11 / 15 = **73.3%**.

iii. **(3 marks)** Give the total *branch coverage* obtained for class `IntBuffer` from the tests provided above.

Only 1 branch covered (see CFG) out of 4. This gives 1/4 = **25%**.

iv. **(3 marks)** Give the total *simple path coverage* obtained for class `IntBuffer` from the tests provided above.

All Simple Paths: $\{1,2\}$, $\{1,3,4,5\}$, $\{1,3,4,6,7,8\}$, $\{1,3,4,6,7,9,10\}$, $\{1,3,4,6,7,9,11,7,8\}$, $\{1,3,4,6,7,9,11,7,9,10\}$

Covered Simple Paths: $\{1,3,4,5\}$, $\{1,3,4,6,7,8\}$

That gives $2/6 = 33\%$ covered simple paths.

v. **(6 marks)** Give three additional test cases which increase the branch coverage obtained for `IntBuffer` to 100%.

```
1   @Test public void testEquals_3() {
2     IntBuffer b1 = new IntBuffer(1);
3     IntBuffer b2 = new IntBuffer(1);
4     b1.write(0); b2.write(1);
5     assertFalse(b1.equals(b2));
6   }
7
8   @Test public void testEquals_4() {
9     IntBuffer b1 = new IntBuffer(1);
10    IntBuffer b2 = new IntBuffer(1);
11    b1.write(0); b2.write(0);
12    assertTrue(b1.equals(b2));
13  }
14
15  @Test public void testEquals_5() {
16    assertFalse(new IntBuffer(1).equals("Dave"));
17  }
```

vi. **(4 marks)** Give additional test cases as necessary to increase the simple path coverage obtained for `IntBuffer` to 100%.

```
1   @Test public void testEquals_6() {
2     IntBuffer b1 = new IntBuffer(1);
3     IntBuffer b2 = new IntBuffer(1);
4     b1.write(0); b2.write(0);
5     b1.write(1); b2.write(0);
6     assertFalse(b1.equals(b2));
7   }
```

vii. **(4 marks)** The `IntBuffer` class does not contain *infeasible* paths. Briefly, explain what this means using an example to illustrate.

An infeasible path is an execution path through a function which cannot, in fact, be taken. This is because the logic of the function prevents this particular path from being possible. A simple example would be:

```
if(x > 0) { ... }
if(x > 0) { ... }
```

Here, only paths taking either *both* false or *both* true branches can be executed. The other two paths are, hence, infeasible.

3. Lambdas and Streams **(30 marks)**

Consider the following code which compiles without error:

```
1  class Item{
2    /**center can not be null*/
3    Point center;
4    Point center() {return center;}
5    Item(Point center){this.center=center;}
6  }
7
8  class Point{
9    public int x; public int y;
10   public Point(int x,int y) {this.x=x;this.y=y;}
11   public String toString() {return "("+x+","+y+")";}
12 }
13
14 public class Main {
15   public static void main(String[] arg) {
16     List<Item> items=new ArrayList<>();
17     addSomeItems(items);
18     System.out.println(centerOfMass(items));
19   }
20
21   public static void addSomeItems(List<Item> items) {
22     items.add(new Item(new Point(20,5)));
23     items.add(new Item(new Point(10,5)));
24     //items.add(new Item(null));
25   }
26
27   public static Point centerOfMass(List<Item> items) {
28     List<Point> centers=items.stream()
29       .map(i->i.center()).collect(Collectors.toList());
30     int xTot=centers.stream().mapToInt(p->p.x).sum();
31     int yTot=centers.stream().mapToInt(p->p.y).sum();
32     xTot/=centers.size();
33     yTot/=centers.size();
34     return new Point(xTot,yTot);
35   }
36 }
```

(a) The following questions are concerned with lines 28–29:

i. **(2 marks)** What is the role of `i->i.center()`?

The list is turned into a stream; for each (`Item`) element we extract the (`Point`) center field, The lambda `i->i.center()` is used to extract the center field.

ii. **(2 marks)** What is the role of `Collectors.toList()`? Why is it needed?

We then collect the result into a new list. The collector `Collectors.toList()` turn a `Stream<Point>` into a `List<Point>`.

iii. **(2 marks)** The method `.map(..)` is generic. What is the type inferred for the invocation in this line?

```
Stream<Point> map(Function<Item,Point>)
```

(b) The following questions are concerned with lines 30–31:

i. **(1 mark)** What is the type of `p` in `p.x`?

```
Point
```

ii. **(3 marks)** Lines 30–31 use `.sum()` where as lines 28–29 use `.collect(..)`. What is the difference?

Those lines extract the x (y) coordinate from the point and sum them all together. We use `mapToInt` in order to avoid boxed integers, and also because `IntStream` supports `.sum()` while `Stream<Integer>` does not. Here by using `.sum()` we get a single value out. Before we used `.collect(..)` in order to produce a collection.

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.

(c) **(4 marks)** Terminology:

Stream operations are in 2 different groups: one group contains `.collect(..)` and `.sum()`, while the other group contains `.map()` and `.mapToInt()`

Name the group containing `.collect(..)` and `.sum()`:

Terminal operations

Name another operation in the group above:

`.forEach()`

Name the group containing `.map(..)` and `.mapToInt()`:

Intermediate operations

Name another operation in the group above:

`.filter()`

(d) Debugging:

The last line of `addSomeItems` is commented out (i.e. Line 24). Assuming we uncomment this line and run our program, answer the following:

i. **(4 marks)** Observed effect: What is the observable result? If there is a printed message, what is is? If there is an exception, what is it, and **what is the line** it originates inside `centerOfMass`?

The program fail with a `NullPointerException` while executing line `30`

ii. **(3 marks)** Defect: Is there a defect in the code? If so, where is it?

The line 5 is defective, since it allows to creates an invalid `Item`.

iii. **(3 marks)** Infection: Is there infection in this execution? If so, where is it?

This execution creates an `Item` object with an invalid **`null`** center. Such object is stored in the `items` list in the main. Note that infection is a property of run time datastructures, and not of the code.

iv. **(3 marks)** Propagation: Is there propagation in this execution? If so, where is it?

The local variable `centers` inside of method `centerOfMass` contains a **`null`** reference instead of an actual `Point` object. This would not happen if all `Items` had a (non-null) `center`. That is, the infected data is propagating and invalidating more data-structures on its way.

v. **(3 marks)** Failure: Do we observe a failure in the code? If so, where do we observe it?

The failure is observed when in the lambda `p->p.x` `p.x` is executed with `p==`**`null`**.

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 Masterclass **(30 marks)**

As for the web assessment tool, for each of the following questions, provide in the answer box the code that should replace `[???]`.

(a) **(6 marks)**

```
1 //The answer must have balanced parentesis
2 [???]
3 public class Exercise1 {
4   public static void main(String[]arg) {
5     A a=new A();
6     assert a.m()==3;
7   }
8 }
```

```
class A{int m(){return 3;}}
```

(b) **(6 marks)**

```
1 //The answer must have balanced parentesis
2 [???]
3 public class Exercise2 {
4   public static void main(String[]arg) {
5     A a1=new A(1);
6     A a2=new B(2);
7     assert a1.f==1;
8     assert a2.f==2;
9   }
10 }
```

```
class A{
int f;
A(int f){this.f=f;}
}
class B extends A{
B(int f){super(f);}
}
```

(c) **(6 marks)**

```
1  //The answer must have balanced parentesis
2  interface I{
3    int a();
4    int b();
5    }
6  public class Exercise3 {
7    public static void main(String [] arg){
8      I i=([???]);
9      assert i.a()==1 && i.b()==2;
10   }
11 }
```

```
new I(){
public int a(){return 1;}
public int b(){return 2;}
}
```

(d) **(6 marks)**

```
1  //The answer must have balanced parentesis
2  public class Exercise4{
3
4  public static int m1(){
5    try{
6      try{if(true){[???]}}
7      catch(Error e){return 2;}
8      finally{return 2;}
9      }
10   catch(Throwable t){return 10;}
11  }
12 public static int m2(){
13   try{
14     try{if(true){[???]}}
15     catch(Error e){return 2;}
16     return 2;
17     }
18   catch(Throwable t){return 10;}
19  }
20 public static void main(String [] arg){
21   assert (m1()!=m2()):"assertion:"+m1()+"!="+m2();
22 }
23 }
```

```
throw new NullPointerException();
```

(e) **(6 marks)**

```
1  //The answer must have balanced parentesis
2  interface A{ int m();}
3  interface B extends A{
4    [???]
5    }
6  public class Exercise5 {
7    public static void main(String[]arg) {
8          A a=new B(){};
9          assert a.m()==3;
10   }
11 }
```

```
default int m() {return 3;}
```

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

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.