Surname: . . . . . **First Name:** . . . . . **Student ID:** . . . . . .

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

TESTS – 2021

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	Examiners Use Only
1.	Code Comprehension	30	
2.	Testing & Object Contracts	30	
3.	Java Generics & Exceptions	30	
4.	Java Masterclass	30	
	Total	120	

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

Consider the following classes which compile without error:

```
1  interface Shape extends Cloneable {
2     /**
3      * Determine whether point within this shape.
4      */
5     public boolean contains(int x, int y);
6     /**
7      * Every shape can be cloned.
8      */
9     public Shape clone();
10 }
```

```
1  public class Rectangle implements Shape {
2     private final int x, y, width, height;
3
4     public Rectangle(int x, int y, int width, int height) {
5        this.x = x;
6        this.y = y;
7        this.width = width;
8        this.height = height;
9     }
10
11    public boolean contains(int x, int y) {
12       x = x - this.x;
13       y = y - this.y;
14       return 0 <= x && 0 <= y && x < width && y < height;
15    }
16
17    public Rectangle clone() {
18       return this;
19    }
20 }
```

```
1  public class Inverted implements Shape {
2     private final Shape shape;
3
4     public Inverted(Shape s) { shape = s; }
5
6     public boolean contains(int x, int y) {
7        return !shape.contains(x,y);
8     }
9
10    public Inverted clone() {
11       return new Inverted(shape.clone());
12    }
13 }
```

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

i. **(1 mark)**

```
1        Shape s1 = new Rectangle(10,10,20,20);
2        System.out.println(s1.contains(11,11));
```

true

ii. **(1 mark)**

```
1        Shape s2 = new Rectangle(20,20,10,10);
2        System.out.println(s2.contains(20,12));
```

false

iii. **(2 marks)**

```
1        Shape s3 = new Rectangle(10,10,20,20);
2        Shape s4 = new Inverted(s3);
3        System.out.println(s4.contains(11,11));
```

false

iv. **(2 marks)**

```
1        Shape s5 = new Rectangle(10,10,20,20);
2        Shape s6 = new Inverted(s5);
3        Shape s7 = new Inverted(s6);
4        System.out.println(s7.contains(11,11));
```

true

v. **(2 marks)**

```
1        Shape s8 = new Rectangle(20,20,10,10);
2        Shape s9 = s8.clone();
3        System.out.println(s8 == s9);
```

true

(b) **(4 marks)** In the box below, implement a class `Square` which is a `Rectangle` with the same *width* and *height*.

```
1 public class Square extends Rectangle {
2    public Square(int x, int y, int width) {
3       super(x,y,width,width);
4    }
5 }
```

(c) **(2 marks)** The field `Inverted.shape` is declared **`final`**. Briefly, discuss what this means.

It means that, once the field has been initilaised, it cannot be assigned again. Note that the object it refers to can still be modified.

(d) **(3 marks)** The method `Rectangle.clone()` simply returns **`this`**. Briefly, justify why this makes sense.

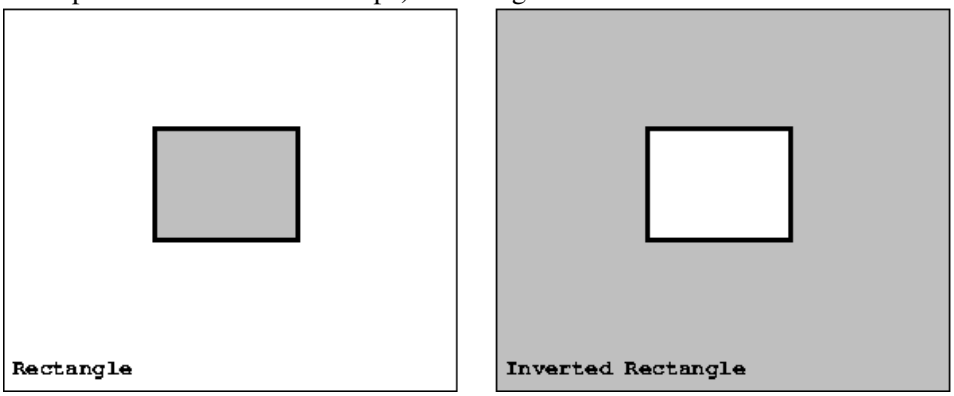
Since the `Rectangle` class cannot be modified (i.e. has only **`final`** primitive fields), there is no benefit to actually cloning the object.

(e) **(3 marks)** The method `Inverted.clone()` enables a *deep clone*, but does not guarantee it. Briefly, discuss what this means.

The `Inverted` class creates a new instance of itself in `clone()`, but relies on `shape.clone()` for its child. Therefore, it depends on whether `shape.clone()` implements a deep clone or not.

(f) **(4 marks)** In your own words, describe what the class `Inverted` represents.

It represents a shape which is the "flipped" version of another shape (i.e. which contains those points not in the other shape). See diagram.



(g) **(6 marks)** In the box below, implement a class `Intersection` which, for two shapes `s1` and `s2`, contains any point in both `s1` *and* `s2`.

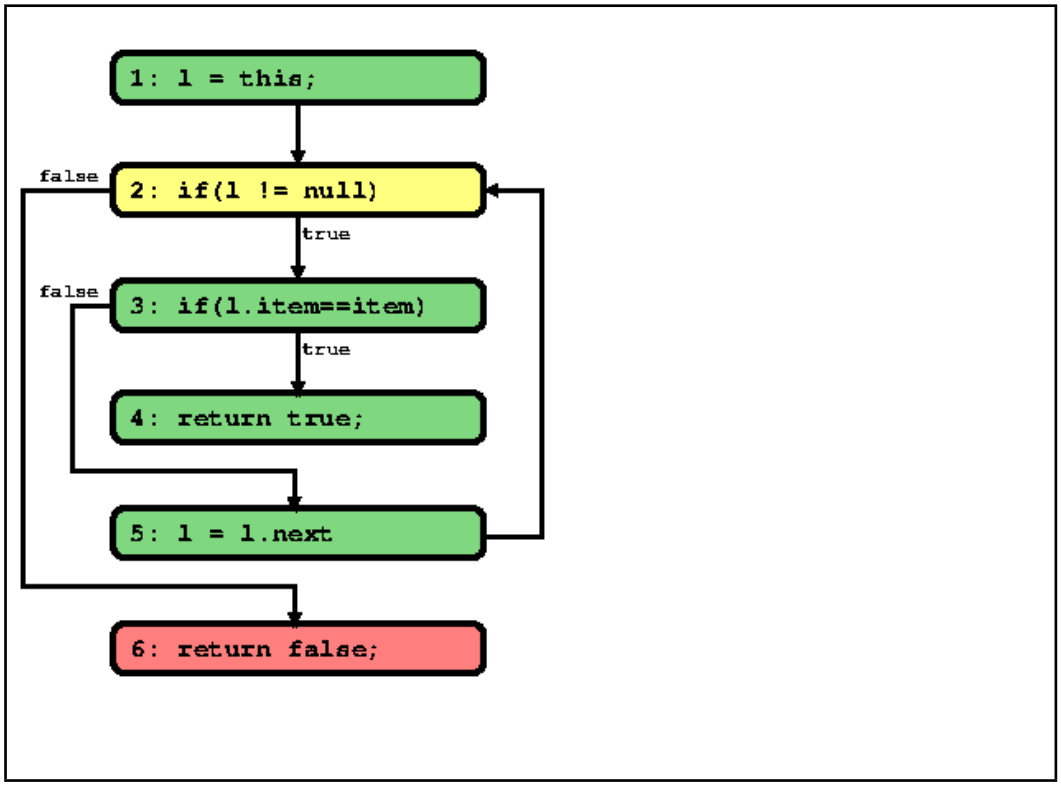
```
1  public class Intersection implements Shape {
2     private final Shape s1;
3     private final Shape s2;
4
5     public Intersection(Shape s1, Shape s2) {
6        this.s1 = s1;
7        this.s2 = s2;
8     }
9
10    public boolean contains(int x, int y) {
11       return s1.contains(x,y) && s2.contains(x,y);
12    }
13
14    public Shape clone() {
15       return new Intersection(s1.clone(),s2.clone());
16    }
17
18 }
```

2. Testing & Object Contracts **(30 marks)**

Consider the following class, which compiles without error:

```
1  public class IntList {
2     private final int item;
3     private final IntList next;
4
5     public IntList(int item, IntList next) {
6        this.item = item;
7        this.next = next;
8     }
9
10    public boolean contains(int item) {
11       IntList l = this;
12       while(l != null) {
13          if(l.item == item) {
14             return true;
15          }
16          l = l.next;
17       }
18       return false;
19    }
```

(a) **(6 marks)** Draw the *control-flow graph* for the `IntList.contains(int)` method.



(b) Consider the following test cases for the class `IntList`:

```
1     @Test void testFind_1() {
2        IntList l1 = new IntList(1,null);
3        assertTrue(l1.contains(1));
4     }
5
6     @Test void testFind_2() {
7        IntList l1 = new IntList(2,null);
8        IntList l2 = new IntList(1,l1);
9        assertTrue(l2.contains(2));
10    }
```

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

Both statements in constructor covered, plus five out of six in `constains()` method gives 7 / 8 = **87.5%**.

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

One branch out of two covered gives 1 / 2 = **50%**.

iii. **(2 marks)** Give one additional test case which increases the branch coverage obtained for class `IntList`.

```
1     @Test void testFind_3() {
2        IntList l = new IntList(1,null);
3        assertFalse(l.contains(2));
4     }
```

.

iv. **(4 marks)** Briefly, explain why it is impossible to obtain 100% *simple path coverage* for class `IntList`.

Because it is impossible to avoid going through outermost while loop at least once. Hence, the simple path $1 \rightarrow 2 \rightarrow 6$ is unrealisable.

(c) The `IntList` class does not implement `equals()` or `hashCode()`.

i. **(4 marks)** Give an appropriate implementation of the method `IntList.hashCode()`.

```
1     public int hashCode() {
2        if(next == null) {
3           return item;
4        } else {
5           return item ^ next.hashCode();
6        }
7     }
```

ii. **(6 marks)** Give an appropriate implementation of the method `IntList.equals()`.

```
1
2     public boolean equals(Object o) {
3        if(o != null && o.getClass() != IntList.class){
4           IntList l = (IntList) o;
5           if(item == l.item) {
6              if(next != null) {
7                 return next.equals(l.next);
8              } else {
9                 return next == l.next;
10             }
11          }
12       }
13       return false;
14    }
```

iii. **(4 marks)** The object contract for `equals()` states that it should be *consistent* across multiple invocations of `x.equals(y)`. Briefly, discuss what this means.

This means that multiple invocations of `x.equals(y)` should consistently return the same provided that `x` and `y` are not modified in some way that affects the `equals()` value.

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.

3. Java Generics & Exceptions **(30 marks)**

Consider the following code:

```
1  class Point { int x; int y; }
2  class ColPoint extends Point { String colour; }
3  class Aux1{
4    void print(List<Point> ps, Point p) {
5      for(Point pi : ps){
6        System.out.println(pi.x+","+pi.y);
7      }
8    }
9    void add(List<Point> ps, Point p){
10     ps.add(p);
11   }
12   void foo(){
13     Point p = new Point();
14     ArrayList<Point> ps = new ArrayList<Point>();
15     ArrayList<ColPoint> cps = new ArrayList<ColPoint>();
16    [???]
17   }
18 }
```

(a) **(2 marks)** If we replaced `[???]` with "`print(ps,p);`" would the code compile?

YES

(b) **(2 marks)** If we replaced `[???]` with "`print(cps,p);`" would the code compile?

NO

(c) **(6 marks)** Justify your answers for (a) and (b) above. Do both lines compile? If so, explain why. If not, is it possible to tweak the definition of `print` so that both lines compile?

Yes, it can be tweaked by changing the signature as follows:
`void print(List<? extends Point> ps, Point p)`

(d) **(2 marks)** If we replaced `[???]` with "`add(ps,p);`" would the code compile?

YES

(e) **(2 marks)**

If we replaced `[???]` with "`add(cps,p);`" would the code compile?

NO

(f) **(6 marks)** Justify your answers for (d) and (e) above. Do both lines compile? If so, explain why. If not, is it possible to tweak the definition of `add` so that both lines compile?

The code can not be tweaked to work in both cases: indeed the second case `add(cps,p);` is logically invalid: we can not add a `Point` to a list of `ColoredPoint`s.

(g) **(4 marks)** In Java, what is the difference between *checked* and *unchecked* exceptions?

Checked exceptions need to be explicitly mentioned in the method signature when they are leaked out. On the other hand, unchecked exceptions can always be leaked.

(h) **(2 marks)** Declare a class `SQLDisaster` as an unchecked exception.

```
class SQLDisaster extends Error {}
```

(i) **(2 marks)** Declare a class `SQLDisaster` as a checked exception.

```
class SQLDisaster extends Exception {}
```

(j) **(2 marks)** Is `Throwable` checked or unchecked? Briefly, justify your answer.

Throwable is checked. It needs to be checked because it is the supertype of all thowable values, so if it were unchecked, we could just cast any kind of exception to Throwable to break the type system.

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)**

In this exercise we will rely on two simple Java classes:

```
class Person{
  public String name;
  Person(String name){this.name=name;}
}
class Car{
  public Person driver;
  public Person passenger;
  Car(Person driver, Person passenger){
    this.driver=driver;
    this.passenger=passenger;
  }
}
```

(a) **(8 marks)** Consider the following code:

```
List<String> exercise1(List<Car> cars){
  return cars.stream()
    .filter(c->c.passenger!=null)
    .map(c->c.passenger.name)
    .collect(Collectors.toList());
  }
```

Rewrite the method `exercise1()`, keeping the exact same observable behaviour but using conventional **`if`**/**`while`**/**`for`** statements (i.e. without using streams).

```
List<String> exercise1(List<Car> cars) {
  List<String> res=new ArrayList<>();
  for(Car c:cars){
    if(c.passenger==null){continue;}
    res.add(c.passenger.name);
  }
  return Collections.unmodifiableList(res);
}
```

(b) **(8 marks)** Consider the following code:

```
1 List<Car> exercise2(List<String> names) {
2   List<Car> res=new ArrayList<>();
3   for(String n:names) {
4     Person p=new Person(n);
5     Car c=new Car(p,null);
6     res.add(c);
7   }
8   return Collections.unmodifiableList(res);
9 }
```

Rewrite the method `exercise2()`, keeping the exact same observable behaviour but without using **`if`**/**`while`**/**`for`** statements (i.e. using streams instead).

```
1 List<Car> exercise2(List<String> names){
2   return names.stream()
3     .map(n->new Person(n))
4     .map(p->new Car(p,null))
5     .collect(Collectors.toList());
6   }
```

(c) **(7 marks)** As for the web assessment tool, provide code to replace `[???]` such that `main(String[])` terminates normally. Note, we still rely on the `Person` and `Car` classes shown at the start of Question 4.

```
1  //The answer must have balanced parentheses
2  interface Unwrap {
3    Car car();
4    default String unwrap(){return car().driver.name;}
5  }
6
7  public class Question4 {
8    public static void main(String[] args) {
9      Unwrap a=()->([???]);
10     assert a.unwrap().equals("Bob");
11   }
12 }
```

```
new Car(new Person("Bob"),null)
```

(d) **(7 marks)** As for the web assessment tool, provide code to replace `[???]` such that `main(String[])` terminates normally. Note, we still rely on the `Person` and `Car` classes shown at the start of Question 4.

```
1  //The answer must have balanced parentheses
2  [???]
3  public class Question5 {
4    public static void main(String[] args){
5      LargeCar car=new LargeCar(new Person("Adam"),
6        new Person("David"),
7        new Person("Marco")
8      );
9      Car car0=car;
10     assert car.driver==car0.driver;
11     assert car.passenger==car0.passenger;
12     assert car.passenger.name.equals("David");
13     assert car.passenger2.name.equals("Marco");
14   }
15 }
```

```
1  class LargeCar extends Car{
2    Person passenger2;
3    LargeCar(Person driver, Person passenger1, Person passenger2) {
4      super(driver, passenger1);
5      this.passenger2=passenger2;
6    }
7  }
```

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

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.

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.