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

Student ID: .....

EXAMINATIONS — 2013

Trimester 1

SWEN221

Software Development

Time Allowed: THREE HOURS

Instructions: Closed Book.
There are 180 possible marks on the exam.

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

No calculators permitted.
Non-electronic Foreign language dictionaries are allowed.

No reference material is allowed.

Question	Topic	Marks
1.	Debugging and Code Comprehension	30
2.	Inheritance and Exception Handling	30
3.	Testing	30
4.	Java Generics	30
5.	Threading and Garbage Collection	30
6.	Inheritance and Polymorphism	30
	Total	180

Question 1. Debugging and Code Comprehension [30 marks]

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

```
1  public interface SimpleSet {
2
3      /**
4       * Add a new item into this SimpleSet.  If item is already
5       * stored in this SimpleSet, then this method does nothing.
6       * Otherwise, it stores item in this SimpleSet.
7       */
8      public void add(Object item);
9
10     /**
11      * Check whether an object is currently stored in this SimpleSet
12      * which equals() the given item.
13      */
14     public boolean contains(Object item);
15 }
16
17 public class Point {
18   private int x;
19   private int y;
20
21   public Point(int x, int y) {
22      this.x = x;
23      this.y = y;
24   }
25 }
```

(a) (i) [2 marks] Based on the documentation provided for `SimpleSet`, state the output you would expect from the following code snippet:

```
1 SimpleSet s = ...;
2 Point p = new Point(1,1);
3 s.add(p);
4 if(s.contains(new Point(1,1))) {
5   System.out.println("MATCH");
6 } else {
7   System.out.println("NO MATCH");
8 }
```

(ii) [5 marks] In the box below, provide an appropriate `equals(Object)` method for class `Point`.

(iii) [2 marks] In the box below, provide an appropriate `hashCode()` method for class `Point`.

Consider the following implementation of an `ArraySet`, which compiles without error:

```
1  public class ArraySet implements SimpleSet {
2      private Object[] items;
3      private int count; // counts number of elements currently used.
4
5      public ArraySet() {
6          this.items = new Object[2];
7          this.count = 0;
8      }
9
10     public void add(Object item) {
11         if(item == null) {
12           throw new IllegalArgumentException("Cannot add null!");
13         }
14         items[count] = item;
15         count = count + 1;
16     }
17
18     public boolean contains(Object o) {
19         for(int i=0;i!=items.length;++i) {
20             if(items[i].equals(o)) {
21                 return true;
22             }
23         }
24         return false;
25     }
26 }
```

(b) There is a bug in method `ArraySet.contains(Object)`.

(i) [3 marks] Give example code which could have caused the following exception:

```
1 Exception in "main" java.lang.NullPointerException
2     at ArraySet.contains(ArraySet.java:20)
3         ...
```

(ii) [3 marks] Briefly, outline how you would fix this bug.

(c) [3 marks] Give example code which could have caused the following exception:

```
1 Exception in "main" java.lang.ArrayIndexOutOfBoundsException: 2
2     at ArraySet.add(ArraySet.java:14)
3     ...
```

(d) [8 marks] In the box below, provide an updated version of `ArraySet.add(Object)` which allows an `ArraySet` to hold an unlimited number of objects.

(e) [4 marks] Currently, `ArraySet` **`implements`** `SimpleSet` (defined on page 2). Briefly, discuss whether or not this seems appropriate.

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.

Question 2. Inheritance and Exception Handling [30 marks]

Consider the following classes which compile without error:

```
1  public class Parent {
2    protected String[] items;
3
4    public Parent(String[] items) { this.items = items; }
5
6    public String get(int index) { return items[index]; }
7  }
8
9  public class Child extends Parent {
10   public Child(String[] items) { super(items); }
11
12   public String get(int index) {
13      try { return super.get(index); }
14      catch(ArrayIndexOutOfBoundsException e) { return null; }
15 } }
```

(a) Briefly describe what happens when each of the following code snippets is executed:

(i) [2 marks]

```
1  Parent parent = new Parent(new String[0]);
2  String str = parent.get(0);
```

(ii) [2 marks]

```
1  Child child = new Child(new String[0]);
2  String str = child.get(0);
```

(iii) [2 marks]

```
1  Parent parent = new Parent(null);
2  String str = parent.get(0);
```

(b) The method `Child.get(int)` *overrides* the method `Parent.get(int)`.

(i) [2 marks] When does a method *override* another?

(ii) [2 marks] When does a method *overload* another?

(c) The `Parent.items` field is declared as **`protected`**.

(i) [2 marks] Briefly, state what *protected* means in this case.

Now, suppose `items` was changed to be **`private`**.

(ii) [2 marks] How would this affect class `Child`?

(iii) [2 marks] How would this affect external classes which use `Parent` or `Child`?

(d) Consider the following classes which also compile without error:

```
1  public class AltParent {
2    protected String[] items;
3
4    public AltParent(String[] items) { this.items = items; }
5
6    public String get(int index) throws BadIndexException {
7      try
8      {
9         if(index < 0 || index >= items.length) {
10          throw new BadIndexException("bad index");
11        }
12        return items[index];
13     } finally {
14        items = null;
15 } } }
16
17 public class BadIndexException extends Exception {
18   public BadIndexException(String msg) { super(msg); }
19 }
```

(e) The class `BadIndexException` defines a *checked exception.*

(i) [4 marks] Briefly, state the difference between *checked* and *unchecked* exceptions.

(ii) [2 marks] Based on this, modify the following code snippet so it now compiles:

```
1  public String lookup(String[] items, int index) {
2    return new AltParent(items).get(index);
3  }
```

(f) The method `AltParent.get(int)` uses a **`finally`** block.

(i) [4 marks] Briefly, explain what the **`finally`** block in `AltParent.get(int)` does.

(ii) [4 marks] Briefly, discuss the situations in which **`finally`** blocks are commonly used.

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.

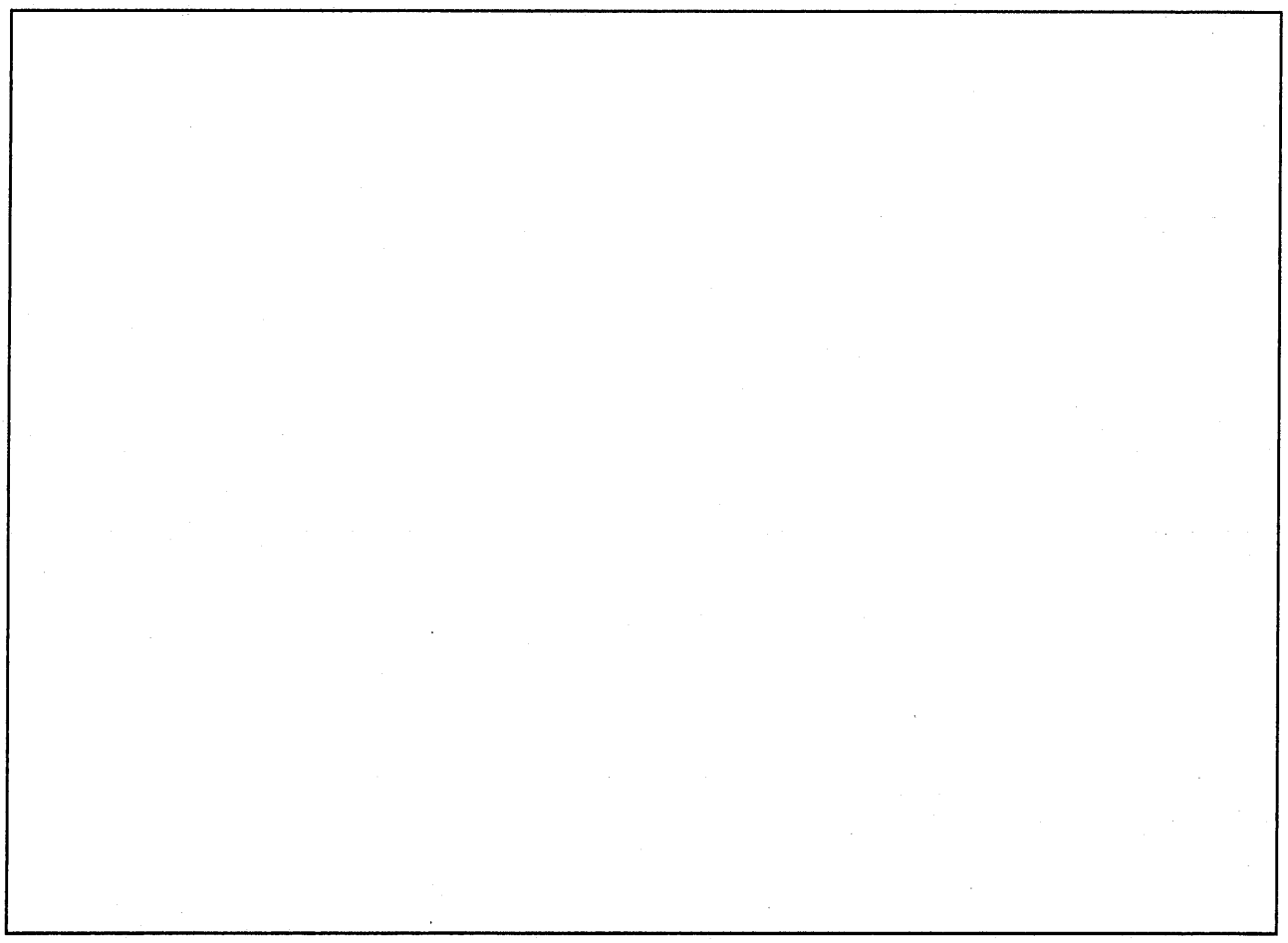
Question 3. Testing

[30 marks]

(a) Consider the following classes which compile without error:

```
1  public class IntArray {
2    private int[] items;
3
4    public IntArray(int[] items) {
5       this.items = items;
6    }
7
8    public int find(int item) {
9      int i = 0;
10     while(i < items.length) {
11        if(items[i] == item) { return i; } // found
12        i = i + 1;
13     }
14     return -1; // not found
15   }
16 }
```

(i) [6 marks] Draw the *control-flow graph* for the `IntArray.find(int)` method:



(ii) [2 marks] What is *statement coverage*?

(iii) [2 marks] What is *branch coverage*?

Consider the following test suite provided for `IntArray`:

```
1  public class IntArrayTests {
2    @Test void testFind_1() {
3      int[] items = {1,2,3};
4      assertTrue(new IntArray(items).find(1) == 0);
5    }
6
7    @Test void testFind_2() {
8      int[] items = {1,2,3};
9      assertTrue(new IntArray(items).find(2) == 1);
10   }
11 }
```

(iv) [2 marks] Give the total *statement coverage* of `IntArray` obtained with `IntArrayTests`.

(v) [2 marks] What problem is there with the test cases found in `IntArrayTests`?

(vi) [2 marks] Give one additional test case which increases the statement coverage obtained for `IntArray`.

(b) Consider the following class which compiles without error:

```
1 public class Util {
2   public static int m(int x, int y) {
3       if(x > y) { return x; }
4       else { return y; }
5   }
6 }
```

(i) [4 marks] Why is an *exhaustive* test of all inputs for `Util.m(int,int)` impractical?

(ii) [4 marks] What is *white-box* testing?

(c) Consider the following class which compiles without error:

```
1 public class Util {
2   public static int f(int x, int y) {
3       int z = 0;
4       if(x >= y) { z = z + 1; }
5       if(x > y) { z = z + 2; }
6       return z;
7   }
8 }
```

(i) [2 marks] Briefly, discuss whether the value returned from method `Util.f(int,int)` can ever be 2.

(ii) [2 marks] Based on your answer above, state what an *infeasible* path is.

(iii) [2 marks] What effect do infeasible paths have on code coverage?

Question 4. Java Generics

[30 marks]

(a) The `Pair` class, shown below, implements a generic container for holding pairs of objects.

(i) [6 marks] By writing neatly on the box below turn `Pair` into a generic version, `Pair<T1,T2>`, where `T1` and `T2` specify the type of items held in the pair.

```
1  public class Pair {
2
3    private Object first;
4
5    private Object second;
6
7    public Pair(Object first, Object second) {
8      this.first = first; this.second = second;
9    }
10
11   public Object getFirst() { return this.first; }
12
13   public Object getSecond() { return this.second; }
14 }
```

(ii) [2 marks] In the box below, provide code which creates an instance of a generic `Pair` which holds a `String` and an `Integer`.

(b) [6 marks] The following code does not compile because `List<String>` is not a *subtype* of `List<Object>`. Explain the problem.

```
1 List<String> strings = new ArrayList<String>();
2 List<Object> objects = strings;
3 objects.add(new Integer(1));
```

(c) [6 marks] The following code does not compile because `List<String>` is not a *supertype* of `List<Object>`. Explain the problem.

```
1 List<Object> objects = new ArrayList<Object>();
2 List<String> strings = objects;
3 objects.add(new Integer(1));
4 String str = strings.get(0);
```

(d) This question concerns Java's *wildcard types*.

(i) [2 marks] Give an example of a wildcard type.

(ii) [4 marks] Briefly, explain what wildcard types are. You may use examples to illustrate.

(e) [4 marks] By writing neatly on the box below, turn `select(Pair,int)` into a generic function which accepts a generic pair, `Pair<T1,T2>`, and returns the best possible generic type `T`.

```
1
2   public Object select(Pair p, int i) {
3     if(i == 0) { return p.getFirst(); }
4     else { return p.getSecond(); }
5   }
```

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.

Question 5. Threading and Garbage Collection [30 marks]

(a) Java supports the notion of *threads*.

(i) [2 marks] Briefly, describe what a thread is.

(ii) [3 marks] Briefly, describe why threads are useful.

(b) Consider the following class which compiles without error:

```
1 public class Counter {
2   private int count;
3   public void inc() { count = count + 1; }
4   public int get() { return count; }
5 }
```

(i) [6 marks] Suppose two threads share one instance of `Counter`. If both call `inc()` exactly once, what value can we expect for field `count` afterwards? Explain your answer.

(ii) [3 marks] In the box below, rewrite `Counter` using *synchronisation* so that `count` correctly matches the total number of calls to `inc()` made by both threads.

(iii) [7 marks] In the box below, provide code to start two threads that share an instance of `Counter`. When each thread starts, it should call `inc()` ten times.

(c) This question concerns *garbage collection.*

(i) [2 marks] Briefly, state what is meant by the term *reachable.*

(ii) [2 marks] Briefly, describe what *garbage collection* is.

(iii) [5 marks] Briefly, describe a simple garbage collection algorithm.

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.

Question 6. Inheritance and Polymorphism [30 marks]

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

```
1  public abstract class Tree {
2    public abstract int walk(Walker w);
3  }
4
5  public class TreeLeaf extends Tree {
6     public int walk(Walker w) { return w.walk(this); }
7  }
8
9  public class TreeNode extends Tree {
10   public final Tree left;
11   public final Tree right;
12
13   public TreeNode(Tree left, Tree right) {
14      this.left = left;
15      this.right = right;
16   }
17
18   public int walk(Walker w) { return w.walk(this); }
19 }
20
21 public interface Walker {
22    public int walk(TreeLeaf l);
23    public int walk(TreeNode n);
24 }
```

(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 OneWalker implements Walker {
2    public int walk(TreeNode n) { return 1; }
3  }
```

(ii) [2 marks]

```
1 public abstract class AbstractWalker implements Walker {
2   public int walk(TreeLeaf l) { return 0; }
3   public abstract int walk(TreeNode n);
4 }
```

(iii) [2 marks]

```
1 public class ConcreteWalker implements Walker {
2   public int walk(TreeLeaf l) { return 0; }
3   public int walk(Tree n) { return 1; }
4 }
```

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

(c) [3 marks] The `TreeNode.left` and `TreeNode.right` fields are declared **`final`**. Briefly, discuss what this means.

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

```
1 public class CountWalker implements Walker {
2   public int walk(TreeLeaf l) { return 1; }
3   public int walk(TreeNode n) {
4     return n.left.walk(this) + n.right.walk(this);
5   }
6 }
```

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

(i) [3 marks]

```
1 int c = new CountWalker().walk(new TreeLeaf());
2 System.out.println(c);
```

(ii) [3 marks]

```
1 int c = new TreeLeaf().walk(new CountWalker());
2 System.out.println(c);
```

(iii) [3 marks]

```
1 TreeLeaf t1 = new TreeLeaf();
2 TreeLeaf t2 = new TreeLeaf();
3 int c = new CountWalker().walk(new TreeNode(t1,t2));
4 System.out.println(c);
```

(e) [4 marks] In your own words, describe what the `CountWalker` class does.

(f) [5 marks] Give a `Walker` implementation which determines the maximum *depth* of a `Tree`. That is, the longest path from the tree's root to any leaf.

****

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.

Student ID: .....

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.