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 }
```

NO MATCH

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

```
boolean equals(Object o) {
  if(o != null && this.class == .getClass()) {
   Point p = (Point) o;
   return this.x == p.x && this.y == p.y;
  }
  return false;
}
```

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

```
int hashCode() {
   return this.x + this.y;
}
```

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         ...
```

```
ArraySet arr = new ArraySet();
arr.contains(null);
```

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

This bug can be fixed by adjusting the loop on line 19 to iterate upto `count` items, rather than `items.length`.

(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     ...
```

```
ArraySet arr = new ArraySet();
arr.add("x");
arr.add("y");
arr.add("z"); // out-of-bounds exception!
```

(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.

```
public void add(Object item) {
  if(item == null) {
    throw new IllegalArgumentException("Cannot add null!");
  }
  if(count == items.length)) {
    Object[] newItems = new Object[count*2];
    System.arrayCopy(items,0,newItems,0,count);
    items = newItems;
  }
  items[count] = item;
  count = count + 1;
}
```

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

No, it is not appropriate. This is because the specification for `SimpleSet` given in its JavaDoc states that `add()` should not add duplicate values into the set. However, the method `ArraySet.add()` does not check for this. Instead, `ArraySet.add()` should call `contains()` before attempting to add an element.

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);
```

An `ArrayIndexOutOfBoundsException` is thrown on line 6 because we're attempting to read `items[0]`, but `items` is an empty array (i.e. has size 0).

(ii) [2 marks]

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

An `ArrayIndexOutOfBoundsException` is thrown on line 6 (as before), but is caught on line 14 and **`null`** is returned instead.

(iii) [2 marks]

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

A `NullPointerException` is thrown on line 6, because we're attempting to read `items[0]`, but `items == null`.

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

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

A method overrides another when it has the same signature (i.e. name and parameter types) as another method defined in a superclass

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

A method overloads another when it's defined in the same class (or possibly a superclass) as another method with the same name, but which has different parameter types.

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

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

Here, **`protected`** means that only methods in the same class or its subclasses can access the field.

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

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

The class `Child` would no longer be able to access `items` directly; however, in fact, `Child` only access `items` indirectly via the `get()` method and so would be unaffected.

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

Most external classes would be unaffected as they would not have had access before anyway. However, any external classes which extended classes `Parent` or `Child` and which accessed `items` directly would no longer compile.

(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.

Checked exceptions represent recoverable errors and extend `Exception`, but not `RuntimeException`. Such exceptions must either be caught using a **`try-catch`** statement or declared in a **`throws`** clause. Unchecked exceptions represented unexpected errors which indicate programmer error; these extend `RuntimeException`, and do not need to be caught or declared in a **`throws`** clause.

(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  }
```

```
public String lookup(String[] items, int index) throws BadIndexException
  return new AltParent(items).get(index); }
```

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

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

A **`finally`** block is always executed after its corresponding **`try-catch`** block, regardless of whether the block exited normally or whether an exception was thrown from within the block and caught (or not). Therefore, in this case, the **`finally`** block sets `items` to **`null`** whenever method `get()` is called.

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

Finally blocks are commonly used to ensure that resources are properly cleaned up. For example, to ensure that a `File` or `Socket` is closed after a method finishes (regardless of how it exits).

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.

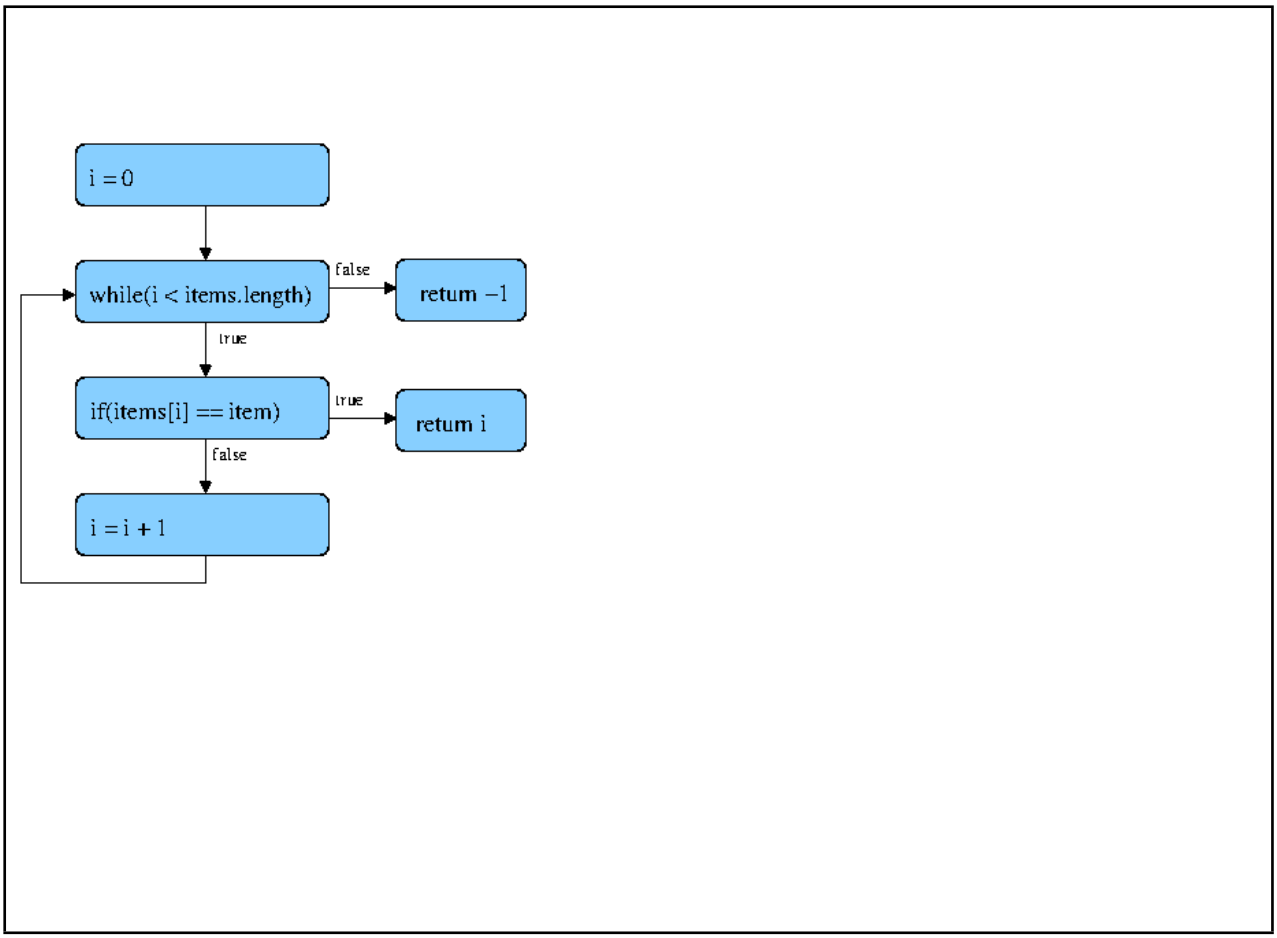
Question 3. Testing

[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*?

The proportion of statements covered by the tests

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

The proportion of branching statements where both sides of the branch are tested

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`.

= 6 / 7

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

They do not test the case where an item is not found

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

```
@Test void testFind_2() {
  int[] items = 1,2,3;
  assertTrue(new IntArray(items).find(4) == -1);
}
```

(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?

An exhaustive test requires testing every possible combination of inputs. Given that an **`int`** variable has 2^{32} possible values, and there are two of them, this gives a total of 2^{64} possible input values to test. This would take an infeasible amount of time to run through each test!

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

White box testing is whether testing is conducted with full access to the source code for the program being tested. This contrasts with black-box testing, where only the specification is available. One advantage of white-box testing is that you can obtain better code coverage; however, one disadvantage is that it leads to testing bias.

(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`.

For the statement `z=z+2` to be executed, we need that `x > y`; however, in such case, the statement `z=z+1` would have already been executed! Thus, `z=2` is not possible

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

An infeasible path is a path through the control-flow graph which appears possible but, in fact, can never happen. For example, the path where **`if`**`(x >= y)` is false but **`if`**`(x > y)` is true is infeasible

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

Infeasible paths do not affect statement or branch coverage. However, they make it impossible to obtain 100% path 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 }
```

(see page 21 for answer)

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

```
new Pair<String,Integer>("h",1);
```

(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));
```

The above code should fail to compile on Line 2. The problem is that, although `String` is a subtype of `Object`, it is not the case that `List<String>` is a subtype of `List<Object>`. In fact, for a generic list `List<T2>` to be a subtype of another list `List<T1>` we require that `T1` is `T2`.

The problem above is that by allowing line 2 to compile, variable `objects` references the same list object as `strings`. Furthermore, adding `Integer` object to `objects` means it is also added to `strings`, which breaks the invariant that `strings` only ever holds `String` objects.

(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);
```

This case is the symmetric opposite of the previous example. This time, however, it is less intuitive because `Object` is not a subtype of `String`! As before, line 2 should not be allowed to compile.

Again, the problem above is that by allowing line 2 to compile, variable `objects` references the same list object as `strings`. Furthermore, adding `Integer` object to `objects` means it is also added to `strings`, which breaks the invariant that `strings` only ever holds `String` objects. Thus, when we attempt to retrieve a `String` object on line 4, we will in fact be given an `Integer` object.

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

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

```
List<?>
```

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

A wildcard represents a type which exists, but is unknown. This severely limits what can be done with such a type (for example, we cannot `add()` an element into a `List<?>`, although we can `get()` an element); however, on the otherhand, wildcards permit subtyping between generic types (e.g. `List<?>` $\geq$ `List<T>`). Finally, it possible to employ *bounds* with wildcard types. For example, `List<? extends Number>` represents a list of unknown types which are all at least a `Number`.

(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    }
```

Answer:

```
public <T> T select(Pair<? extends T, ? extends T> p, int i) { ... }
```

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.

(answer to Question 4a)

```
1  public class Pair<T1,T2> {
2
3    private T1 first;
4
5    private T2 second;
6
7    public Pair(T1 first, T2 second) {
8      this.first = first; this.second = second;
9    }
10
11   public T1 getFirst() { return this.first; }
12
13   public T2 getSecond() { return this.second; }
14 }
```

Question 5. Threading and Garbage Collection [30 marks]

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

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

A thread is a single flow of control (or execution) through a program. Multiple threads of control may be executing the same program at the same time.

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

Threads enable multiple things to happen simultaneously within a program (or, at least, the appearance of such). On machines with multiple cores this can genuinely lead to better utilisation of cores and, ultimately, better performance. However, it requires the problem being solved to be divided upto in chunks which can be executed separately in different threads. For example, a individual requests for a web server can be handled by different threads; likewise, summing the elements of a large list can be divided up into a number of smaller chunks for parallel execution.

(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.

In this case, we can certainly expect the following possible values:

- `2` — This is the simplest case, where both threads call `inc()` at separate times, leading to the expected outcome.
- `1` — In this case, both threads call `inc()` at roughly the same time. Both threads read `count` before either has had the chance to write the increment value back. This means that they will then both write the same value of `1`.
- `0` — Since `counter` is not marked **`volatile`** (and access to it is not synchronized), its entirely possible that any modifications to field `count` are not made *visible* to other threads.

(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.

```
1
2  public class Counter {
3    private int count;
4    public synchronized void inc() { count = count + 1; }
5    public synchronized int get() { return count; }
6  }
```

(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.

```
1
2  public class CounterJob extends Thread {
3    private Counter counter;
4
5    public CounterJob(Counter c) { counter = c; }
6
7    public void run() {
8       for(int i=0;i!=10;++i) { c.inc(); }
9    }
10
11   public void static main(String[] args) {
12     Counter counter = new Counter();
13     CounterJob job1 = new CounterJob(counter);
14     CounterJob job2 = new CounterJob(counter);
15     job1.start();
16     job2.start();
17   }
18 }
```

(c) This question concerns *garbage collection*.

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

An object `o1` is reachable from another object `o2` if it — or an object reachable from it — contains a field which refers to `o2`.

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

Garbage collection is JVM background process which periodically attempts to free unreachable objects in order to increase available memory.

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

Mark and sweep is the most common (and simple) garbage collection algorithm. This starts by traversing and marking those objects reachable from the nominated "roots" (references held on an active threads stack, or in global variables, etc). The marked objects are then swept down to lower memory locations which compacts them together and leaves those which are unreachable above. The unreachable objects can then be released *en masse*.

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  }
```

Does not compile because `OneWalker` must implement `walk(TreeLeaf)`

(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 }
```

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

(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 }
```

Does not compile, because `ConcreteWalker` must implement `walk(TreeNode)` (note: the given `walk(Tree)` is not sufficient because it overloads the required method)

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

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 `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);
```

1

(ii) [3 marks]

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

1

(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);
```

2

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

It counts the number of `TreeLeaf`s in a given `Tree`.

(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.

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