TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VUW VICTORIA UNIVERSITY OF WELLINGTON

Student ID: .....

EXAMINATIONS — 2011

MID-YEAR

SWEN221

Software Development

Time Allowed: 2 Hours

Instructions: There are 120 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.
Non-electronic Foreign language dictionaries are allowed.
Calculators ARE NOT ALLOWED.
No reference material is allowed.

Question	Topic	Marks
1.	Style, Debugging and Exceptions	20
2.	Inheritance and Polymorphism	20
3.	Encapsulation and Object Contracts	20
4.	Java Generics	20
5.	Testing	20
6.	Threading, Garbage Collection and Reflection	20
	Total	120

Question 1. Style, Debugging and Exceptions [20 marks]

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

(i) Generally, good code . . .

1. does not need any comments.
2. will need some comments.
3. needs comments on every statement.

Ans: 2

(ii) Following Java conventions as discussed in this course, constructors should be named . . .

1. CapsWithWholeWordsCaps.
2. firstLowercaseThenCaps.
3. UPPERCASE_WITH_UNDERSCORE.

Ans: 1

(iii) Debugging is the process of . . .

1. reporting bugs.
2. checking there are no bugs.
3. finding and eliminating bugs.

Ans: 3

(iv) Reproducing bugs . . .

1. should be avoided during the debugging process.
2. is an important and straightforward step in the debugging process.
3. is an important and sometimes difficult step in the debugging process.

Ans: 3

(v)

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

Ans: 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.

```
1  public class PalindromeCheck {
2
3    public static void main(String[] args) {
4      try {
5        String word = null;
6        boolean caseSensitive = true;
7        for (int i = 0; i != args.length; i++) {
8          word = args[i];
9          if (word.startsWith("-")) {
10           String arg = args[i];
11           if (arg.equals("ci")) {
12             caseSensitive = false;
13           }
14           if (arg.equals("-help")) {
15             System.out.println("Palindrome␣check");
16             return;
17           } else {
18             throw new RuntimeException("Unknown␣option");
19           }
20         }
21       }
22       if (! caseSensitive) {
23         word = word.toUpperCase();
24       }
25       System.out.println("Result:␣" + palindromeCheck(word));
26     }
27     catch (Exception e) {
28       System.out.println("Sorry");
29     }
30     finally {
31       System.out.println("Bye");
32     }
33   }
34
35   /**
36    * Returns true if and only if the given word is a palindrome,
37    * that is, is the same when read forwards as when read backwards.
38    * For example, the word "otto" is a palindrome
39    * but the word "palindrome" is not.
40    */
41   public static boolean palindromeCheck(String word) {
42     ...
43   }
44 }
```

(b) For each of the following program calls, state what the program prints. You may assume that method `palindromeCheck(String word)` behaves as described.

(i) [2 marks] `java PalindromeCheck -help`
The input to `main` will be the array `{ "-help" }`

Palindrome check
Bye

(ii) [2 marks] `java PalindromeCheck -verbose otto`
The input to `main` will be the array `{ "-verbose", "otto" }`

Sorry
Bye

(iii) [2 marks] `java PalindromeCheck otto palindrome`
The input to `main` will be the array `{ "otto", "palindrome" }`

Result: false
Bye

(iv) [4 marks] The program has a bug that prevents the case-insensitive case from working properly. Provide an input that exposes the bug, explain what the problem is, and how you would fix it.

The case-insensitive case fails because of the check `arg.equals("ci")` on line 11. It should be `arg.equals("-ci")`.

A input which would expose this error is `java PalindromeCheck -ci otTo`. This will current prints *`"Sorry"`*, but should print *`"Result:␣true"`*.

(c) [5 marks] Using a *recursive* method, implement `palindromeCheck(String word)` so that it behaves as described.

```
public static boolean palindromeCheck(String word) {
 if (word.length() < 2) return true;
 if (word.charAt(0) == word.charAt(word.length()-1)) {
  return palindromeCheck(word.substring(1, word.length()-1));
 }
 return false;
}
```

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 Polymorphism [20 marks]

Consider the following Java classes and interfaces.

```
1  interface Endangered {}
2
3  interface Named { String getName(); }
4
5  class Animal { public boolean chases(Animal a) { return false; } }
6
7  class Kiwi extends Animal implements Endangered {}
8
9  class Cat extends Animal implements Named {
10     public boolean chases(Animal a) { return true; }
11     public String getName() { return "Kitty"; }
12 }
13
14 class Dog extends Animal implements Named {
15     public boolean chases(Cat c) { return true; }
16     public String getName() { return "Fluffy"; }
17 }
```

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

(i) [2 marks]

```
1  class Sheep extends Animal implements Named { }
```

Sheep is not abstract and does not override abstract method getName() in Named

(ii) [2 marks]

```
1  abstract class Tuatara extends Animal implements Endangered, Named {
2     public abstract String getName();
3  }
```

correct

(iii) [2 marks]

```
class SuperKiwi extends Kiwi {
    public boolean chases(Animal a) {
        if (a instanceof Cat) {
            Named kitty = (Cat) a;
            return kitty.getName() == "Kitty";
        }
        return false;
    }
}
```

correct

(b) Given the above declarations, state either the output of the following code snippets or explain why the code does not compile.

(i) [2 marks]

```
    Animal puss = new Cat();
    puss.getName();
```

Doesn't compile because `Animal` doesn't have method `getName()`

(ii) [2 marks]

```
    Cat puss = new Cat();
    Kiwi kiwi = new Kiwi();
    System.out.println(puss.chases(kiwi));
```

Output: true

(iii) [2 marks]

```
1     Animal puss = new Cat();
2     Kiwi kiwi = new Kiwi();
3     System.out.println(puss.chases(kiwi));
```

Output: true

(iv) [2 marks]

```
1     Animal puss = new Cat();
2     Dog fluffy = new Dog();
3     System.out.println(fluffy.chases(puss));
```

Output: false

(c) [6 marks] Discuss the similarities and differences between an abstract class and an interface.

1. Neither abstract classes nor interfaces can be instantiated.
2. Interfaces are limited to public methods and constants with no implementation. Abstract classes can have partial implementations, protected parts, etc. Thus, abstract classes can improve code reuse.
3. A Class may implement several interfaces but can only extend one abstract class.

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. Encapsulation and Object Contracts [20 marks]

(a) Consider the following Java code. It compiles without error, but is of poor quality.

```
1  class IntArray  implements Cloneable {
2      public int[] data;
3      public int size;
4
5      public boolean equals(IntArray other) {
6          if (this == other) return true;
7          if (size != other.size) return false;
8          for (int i = 0; i < size; i++) {
9              if (this.data[i] != other.data[i]) {
10                 return false;
11             }
12         }
13         return true;
14     }
15
16     public int hashCode() {
17         // Returns a hash code based on the contents of the array data
18         return java.util.Arrays.hashCode(data);
19     }
20
21     public Object clone() {
22         IntArray cloned = new IntArray();
23         cloned.data = this.data;
24         cloned.size = this.size;
25         return cloned;
26     }
27 }
```

(i) [2 marks] Is `IntArray` properly encapsulated? Justify your answer.

No, it is not properly encapsulated. This is because the fields `Data` and `size` are **`public`** and, hence, the implementation of `IntArray` is exposed for all to see.

(ii) [2 marks] Briefly, discuss one advantage of proper encapsulation.

Changes to the internal implementation of a class does not effect client code which uses that class.

(iii) [6 marks] Identify and explain 3 problems with the `equals` and `hashCode` methods given above.

1) wrong argument type — should be `equals(Object other)`.

2) in `equals()`, variable `other` should be checked for **`null`**

3) `equals()` method is not compatible with `hashCode()`. This is because two instances of `IntArray` may be `equals()`, but have different `hashCode()` values. This can happen when the `size` variable is less than the length of `data`, and we have different "junk" values in the unused portion.

(iv) [6 marks] There are two standard ways of implementing the `clone` method. Discuss these two approaches, and identify which is used by `IntArray`.

- **Shallow Clone.** Only items at the first level of the object are cloned. Items at deeper levels remain the same, and so aliasing between the cloned and original object may occur.
- **Deep Clone.** Objects at all levels are cloned. Therefore, the cloned and original objects are entirely separate.

`IntArray` implements a shallow clone.

(v) [4 marks] Rewrite the `clone` method given for `IntArray` so that it uses the other type of clone.

```
public Object clone() {
  IntArray cloned = new IntArray();
  cloned.data = new int[size];
  cloned.size = size;
  for(int i=0;i!=size;++i) {
   cloned.data[i] = data[i];
  }
  return cloned;
}
```

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 4. Java Generics [20 marks]

(a) The `Tree` class, shown below, implements a *binary tree*.

(i) [6 marks] By writing neatly on the box below turn `Tree` into a generic version, `Tree<T>`, where `T` specifies the type of data held in the tree.

```
1  public class Tree {
2      private Tree left;
3      private Tree right;
4      private Object data;
5
6      public Tree(Tree left, Tree right, Object data) {
7          this.left = left;
8          this.right = right;
9          this.data = data;
10     }
11
12     public Tree left() {
13         return left;
14     }
15
16     public Tree right() {
17         return right;
18     }
19
20     public Object data() {
21         return data;
22     }
23
24     public static void flatten(Tree tree, List list) {
25         if(tree.left != null) { flatten(tree.left,list); }
26         list.add(tree.data);
27         if(tree.right != null) { flatten(tree.right,list); }
28 }   }
```

(see page 17 for answer)

(ii) [4 marks] In the box below, provide code which creates an instance of a generic `Tree` which holds `Strings`. Your tree should contain at least three nodes.

```
Tree<String> t1 = new Tree<String>(null,null,"Hello");
Tree<String> t2 = new Tree<String>(null,null,"World");
Tree<String> t3 = new Tree<String>(t1,t2,"␣");
```

(iii) [2 marks] Briefly, discuss why the generic `Tree<T>` is preferable to the non-generic version.

In the non-generic version of `Tree`, one could not be sure exactly what objects were stored in the three. Thus, one would need to cast objects as they came out of the tree, leading to potential `ClassCastExceptions` at runtime. With the generic version, one knows that every object in an instance of `Tree<T>` is at least an instance of `T` — no casting is required.

(iv) [2 marks] Suppose you wanted a generic version of `Tree` which ensured every data object had a `compareTo()` method. Briefly, discuss how you would do this.

The `Comparable<T>` interface includes the `compareTo()` method. Therefore, you could force every element of the tree to implement this interface by using a lower bound, as follows:

```
public class Tree<T extends Comparable<T> > { ... }
```

(b) [6 marks] In Java, `List<String>` is **not** a subtype of `List<Object>`. Discuss why this is not permitted, using example code to illustrate.

In Java, generic parameters must be the same between subtypes. For example, `ArrayList<String>` is a subtype of `List<String>` because `ArrayList` is a subtype of `List`, and the generic parameters are identical.

The reason for this is that, otherwise, one could break the invariants maintained by the generic type. The following illustrates:

```
void add(List<Object> list, Object item) { list.add(item); }
...
List<String> ls = new ArrayList<String>();
Integer num = 1; // auto-boxing applied here
add(ls,num);
```

If `List<String>` were a subtype of `List<Object>`, this code would compile — but, there would be a serious problem. This is because the code allows an `Integer` object to be added into a `List<String>` — thereby breaking the invariant that the list held only `Strings`.

(answer to Question 4a)

```
1  public class Tree<T> {
2    private Tree<T> left;
3    private Tree<T> right;
4    private T data;
5
6    public Tree(Tree<T> left, Tree<T> right, T data) {
7        this.left = left;
8        this.right = right;
9        this.data = data;
10   }
11
12   public Tree<T> left() {
13       return left;
14   }
15
16   public Tree<T> right() {
17       return right;
18   }
19
20   public T data() {
21       return data;
22   }
23
24   public static <S> void flatten(Tree<S> tree, List<? super S> list) {
25       if(tree.left != null) { flatten(tree.left,list); }
26       list.add(tree.data);
27       if(tree.right != null) { flatten(tree.right,list); }
28 } }
```

Question 5. Testing [20 marks]

(a) [3 marks]

Discuss the similarities and differences between white-box and black-box testing.

In white-box testing, the tester has access to the full source code of the program being tested, as well as the specification. In contrast, black-box testing the tester has access only to the specification, and the idea is to eliminate any inherent bias which arises from looking directly at the code.

(b) [3 marks]

What benefit does simple path coverage provide in managing test coverage?

Simple path coverage provides a better indicator of test coverage than either statement or branch coverage do. This is because statement and/or branch coverage can report 100% coverage, even though certain execution paths which can arise in practice have not been tested.

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) On the following page there is code for the `InsuranceCalculator` class.

```
1  public class InsuranceCalculator {
2
3          public InsuranceCalculator() {}
4
5          // Returns -1 if age is either less than 18 or greater than 64,
6          // or if the number of previous claims is either negative or greater
7          // than 5.  Otherwise returns a positive value representing the
8          // insurance premium for insuring this person.
9          public int calculatePremium(int age, int numPrevClaims) {
10                 if ((age < 18) || (age > 64)) {
11                    return -1;
12                 }
13                 if ((numPrevClaims < 0) || (numPrevClaims > 5)) {
14                    return -1;
15                 }
16                 int premium = 100;
17                 if ((age / 10) > numPrevClaims) {
18                    for (int loop = 0; loop < numPrevClaims; loop++) {
19                        premium = premium * 1.1;
20                    }
21                    if (premium > (age * 5)) {
22                       premium = age * 5;
23                    }
24                 } else {
25                    premium += 100;
26                    for (int loop = 0; loop < numPrevClaims - 1; loop++) {
27                        premium = premium * 1.1;
28                    }
29                    if (premium < (age * 10))
30                       premium = age * 10;
31                 }
32                 return premium;
33         }
34
35 }
```

(i) [8 marks]

Write five JUnit test cases for the `calculatePremium` method. Your test cases should cover the boundary conditions identified in the method's comment.

```
@Test void test1() {
  InsuranceCalculator ic = new InsuranceCalculator();}
  assertTrue(300 == ic.calculatePremium(30,0));
}

@Test void test2() {
  InsuranceCalculator ic = new InsuranceCalculator();}
  assertTrue(-1 == ic.calculatePremium(17,0));
}

@Test void test3() {
  InsuranceCalculator ic = new InsuranceCalculator();}
  assertTrue(-1 == ic.calculatePremium(65,0));
}

@Test void test4() {
  InsuranceCalculator ic = new InsuranceCalculator();}
  assertTrue(-1 == ic.calculatePremium(30,-1));
}

@Test void test5() {
  InsuranceCalculator ic = new InsuranceCalculator();}
  assertTrue(-1 == ic.calculatePremium(30,6));
}
```

(ii) [6 marks]

Will the five black box test cases you have written also give you 100% statement coverage? Justify your answer by identifying (for each test case) what statements are covered.

No, it won't. Lines covered by tests are:

- `test1()` — covers lines 10, 13, 16, 17, 25, 26, 29, 30, 31 and 32.
- `test2()` — covers lines 10-12.
- `test3()` — covers lines 10-12.
- `test4()` — covers lines 10, 13-15.
- `test5()` — covers lines 10, 13-15.

As we can see from this, lines 18–23 are not covered by any test. Likewise, line 27 is not 27 covered either. Therefore, we clearly have not achieved 100% coverage.

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. Threading, Garbage Collection & Reflection [20 marks]

Consider the `SimplePrint` class below:

```
1  public class SimplePrint extends Thread {
2
3    public SimplePrint() {}
4
5    public void run() { System.out.println("run called"); }
6
7    public static void main(String[] args) {
8      new SimplePrint().run();
9      new SimplePrint().start();
10   }
11 }
```

(a) [2 marks]

There are two statements in the `main` method. What are the consequences of each of these statements?

The first statement on line 8 simply calls the `run()` method as per normal. The second statement on line 9 starts up a separate thread of execution, which will run concurrently with the thread executing the statement on line 9.

(b)

The `SimpleCounter` class on the following page describes a very simple multi-threaded program.

(i) [4 marks]

Assume the program runs to completion and the `System.out.println()` method is properly synchronized. Will the output always be that shown below? If so, why? If not, why not?

count=100000
count=100000

No, it will not always return those values. The reason for this is that the two `Thread`s may not be at the same position through the loop when `swap` is called. This means the process of incrementing `count` may not go in a strictly sequential order.

(ii) [3 marks]

Sometimes the program doesn't complete, and seemingly freezes forever. Why?

This is because a dead-lock can occur if (by chance) both `Threads` enter their `swap()` method at the same time. Then, each will be waiting for the other to release its lock and, hence, they will wait forever.

```
1  public class SimpleCounter extends Thread {
2
3    private int count;
4    public SimpleCounter otherCounter;
5
6    SimpleCounter() { count = 0; }
7
8    public synchronized void setCount(int val) { count = val; }
9
10   public synchronized int getCount() { return count; }
11
12   public synchronized void swap() {
13     int value = count;
14     count = otherCounter.getCount();
15     otherCounter.setCount(value);
16     return;
17   }
18
19   public void run() {
20     for (int loop = 0; loop < 100000; loop++) {
21       setCount(getCount() + 1);
22       swap();
23     }
24     System.out.println("count=" + getCount());
25   }
26
27   public static void main(String[] args) {
28     SimpleCounter c1 = new SimpleCounter();
29     SimpleCounter c2 = new SimpleCounter();
30     c1.otherCounter = c2;
31     c2.otherCounter = c1;
32     c1.start();
33     c2.start();
34   }
35
36 }
```

(c) [3 marks]

Clearly indicate which one (or more) of the following three statements are true.

1. Non-static inner classes can access fields/methods of the enclosing class.
2. Inner classes automatically extend the enclosing class.
3. An instance of a private inner class cannot be returned from a method of the enclosing class to a method of an outside class.

ANS: (1) only

(d) In the context of garbage collection, an object can be destroyed when it is no longer *reachable*.

(i) [2 marks]

Describe what is meant by the term *reachable*.

An object `o1` is *reachable* from another object `o2` if `o2` holds a reference to `o1`.

(ii) [3 marks]

Discuss the validity of the following statement: "An object is destroyed immediately when it becomes unreachable."

This statement is not always correct. This is because an unreachable object can only be destroyed by the garbage collector (and even then it may not always be). As the garbage collector is only run periodically, there maybe a significant delay between an object becoming unreachable and it being actually destroyed.

(e) [3 marks] In Java, you cannot normally access **`private`** fields outside of their declared class. Briefly, discuss how *reflection* can be used to do this.

Using reflection you can examine the fields of a class using the `getDeclaredFields()` method. For example, using code such as:

```
for(Field f : o.getClass().getDeclaredFields()) {
  System.out.println(f.getName());
}
```

This piece of code will actually list **`private`**, as well as **`public`** fields. Furthermore, one can read the value of a `Field f` using the `get()` method and/or set its value using the `set()` 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.

****