TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI

VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2016

TRIMESTER 1

SWEN221

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
All questions are of equal value

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

Question	Topic	Marks
1.	Code Comprehension	30
2.	Java Generics	30
3.	Java Masterclass	30
4.	Java 8	30
	Total	120

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

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

```
1  interface Pipe {
2      void write(int item);
3  }
```

```
1  class Buffer implements Pipe {
2      private int[] items;
3      private int writePos;
4      private int readPos;
5
6      public Buffer(int len) { items = new int[len]; }
7
8      public void write(int item) {
9          items[writePos] = item;
10         writePos = writePos + 1;
11     }
12     public int read() {
13         int item = items[readPos];
14         readPos = readPos + 1;
15         return item;
16    }
17 }
```

```
1  class NegativeFilter implements Pipe {
2      private Pipe next;
3
4      public NegativeFilter(Pipe n) { next = n; }
5
6      public void write(int item) {
7          if (item >= 0) { next.write(item); }
8      }
9  }
```

```
1  class Accumulator implements Pipe {
2      private Pipe next;
3      private int sum;
4
5      public Accumulator(Pipe n) { next = n; }
6
7      public void write(int item) {
8          sum = sum + item;
9          next.write(sum);
10     }
11 }
```

(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        Buffer b = new Buffer(2);
2        b.write(1);
3        System.out.println(b.read());
```

```
1
```

(ii) **(2 marks)**

```
1        Buffer b = new Buffer(2);
2        b.write(20);
3        b.write(30);
4        System.out.println(b.read());
```

```
20
```

(iii) **(2 marks)**

```
1        Buffer b = new Buffer(2);
2        Pipe p = new NegativeFilter(b);
3        p.write(-99);
4        p.write(100);
5        System.out.println(b.read());
```

```
100
```

(iv) **(2 marks)**

```
1    public static void question1d() {
2        Buffer b = new Buffer(10);
3        Pipe nf = new NegativeFilter(b);
4        Pipe acc = new Accumulator(nf);
5        acc.write(100);
6        acc.write(-99);
7        acc.write(10);
8        System.out.println(b.read() + "," + b.read());
```

```
100,1
```

(b) Consider the following error message:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at Buffer.write(Buffer.java:9)
```

(i) **(3 marks)** Briefly, describe how this error could have arisen.

This error can arise if an attempt is made to write too many items into a `Buffer`. For example, if the `Buffer` can hold two items and we try to write three.

(ii) **(3 marks)** In the box below, provide code which will cause this error.

```
Buffer b = new Buffer(2); b.write(1);
b.write(1);
b.write(1);
```

(c) **(6 marks)** `Fork` is an implementation of `Pipe` which connects to *two* `Pipe` instances. When an item is written to a `Fork`, it is then written to both connections. In the box below, provide an implementation of `Fork`.

```
1  class Fork implements Pipe {
2    private final Pipe left;
3    private final Pipe right;
4
5    public Fork(Pipe left, Pipe right) {
6      this.left = left;
7      this.right = right;
8    }
9
10   public void write(int item) {
11     left.write(item);
12     right.write(item);
13   }
14 }
```

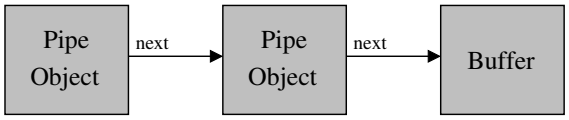
SPARE PAGE FOR EXTRA ANSWERS

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

(d) **(5 marks)** In the box below, explain the meaning of the following statement in terms of *objects* and *references*. Your discussion should include a diagram to illustrate.

"A pipeline consists of one or more pipes connected together, and ending with a buffer"

A pipeline corresponds to a sequence of `Pipe` objects connected together by their respective `next` references (or equivalent). Each branch in the pipeline is eventually terminated by a `Buffer`. Items move down the pipeline by being passed from one `write(int)` method to another.



Here we see a pipeline made up of three *objects*. Each `Pipe` object is connected to the next one in the pipeline via its next *reference*.

(e) **(5 marks)** In the box below, explain the meaning of the following statement. Your discussion should indicate how you would modify the code given on page 2.

"Abstract classes can be used to eliminate duplicate code"

An abstract class can contain abstract methods which have no body (similar to how methods are declared in interfaces). Concrete subclasses must implement any abstract methods. Abstract classes may also contain methods with bodies which can be reused amongst subclasses.
For example, in the `Pipe` example we could create an abstract class `AbstractPipe` which contains a field `Pipe next` and a corresponding constructor. The classes `NegativeFilter` and `Accumulator` can then `extend` this abstract class to provide a concrete implementation of the `write(int)` 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. Java Generics **(30 marks)**

(a) Recall the `Pipe` interface from page 2, and its associated implementations.

(i) **(2 marks)** By writing neatly in the box below, turn `Pipe` into a generic version `Pipe<T>` where `T` specifies the type of items which can be written.

```
1 interface Pipe {
2     void write(int item);
3 }
```

See page 11 for answer.

(ii) **(6 marks)** By writing neatly in the box below, turn `Buffer` into a generic version `Buffer<T>`.

```
1  class Buffer implements Pipe {
2
3      private int[] items;
4
5      private int writePos;
6
7      private int readPos;
8
9      public Buffer(int len) {
10
11         items = new int[len];
12     }
13
14     public void write(int x) {
15
16         items[writePos] = x;
17
18         writePos = writePos + 1;
19     }
20
21     public int read() {
22
23         int x = items[readPos];
24
25         readPos = readPos + 1;
26
27         return x;
28     }
29 }
```

See page **??** for answer.

(b) The following interface illustrates a "colour pipeline". That is, a kind of `Pipe` to which we can only write instances of the class `Colour`.

```
1 interface ColourPipeline<T extends Colour> extends Pipe<T> { }
```

(i) **(4 marks)** In the above, "`<T extends Colour>`" indicates that `Colour` is an *upper bound* on `T`. In your own words, briefly explain what this means.

An upperbound `B` for a generic type `T` indicates that any type instantiated for `T` must be a subtype of `B`. That is, `T` can be the upperbound itself, or a subclass of the upperbound or a class which implements the upper bound (if it is an interface).

(ii) **(5 marks)** Provide a *generic method* `writeAll(T[],ColourPipe<T>)` in the box below which writes every array element into the `ColourPipe<T>` parameter.

```
1 <T extends Colour> void writeAll(T[] in, ColourPipe<T> out) {
2   for(T item : in) {
3     out.write(item);
4   }
5 }
```

(c) The following code does not compile because `Pipe<Colour>` is not a *subtype* of `Pipe<Object>`.

```
1 Pipe<Colour> pipeCol = new Buffer<Colour>(10);
2 Pipe<Object> pipeObj = pipeCol;
3 pipeObj.write("Hello");
```

(i) **(2 marks)** Identify the line number above to which the compilation error would refer.

Line 2

(ii) **(3 marks)** Suppose the Java compiler allowed the above program to compile. What problem would arise when executing the program?

Variable `pipeCol` refers to a `Pipe` which can only accept `Colour` objects. In contrast, variable `pipeObj` refers to a `Pipe` which can accept any kind of `Object`. If the above were allowed, then `pipeObj` would refer to the same object as `pipeCol`. Thus, on line 3, a `String` object would be written into a `Pipe<Colour>` object, which breaks its invariant.

(d) This question concerns Java's *wildcard types* (e.g. `Pipe<?>`).

(i) **(3 marks)** Briefly, explain what the wildcard `?` in type `Pipe<?>` means.

A wildcard type represents a concrete type which is unknown. This differs from a generic type `T` as there are restrictions placed on what operations can be performed with a wildcard. In particular, we cannot write anything to an instance of `Pipe<?>`.

(ii) **(5 marks)** Briefly, explain why the type `Pipe<String>` is a subtype of `Pipe<?>`.

The type `Pipe<String>` is a subtype of `Pipe<?>`, meaning we can write things like this:

```
Pipe<String> pipeStr = ...
Pipe<?> pipeUnknown = pipeStr;
```

This is safe because of the restrictions placed on what we can do with a variable of type `Pipe<?>`. In particular, since we cannot write *anything* to such a pipe, we cannot write an object which would break the `Pipe<String>` invariant.

```
interface Pipe<T> {
  void write(T item);
}
```

```
class Buffer<T> implements Pipe<T> {

    private T[] items;

    private int writePos;

    private int readPos;

    public Buffer(int len) {

        items = (T[]) new Object[len];
    }

    public void write(T x) {

        items[writePos] = x;

        writePos = writePos + 1;
    }

    public T read() {

        int x = items[readPos];

        readPos = readPos + 1;

        return x;
    }
}
```

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 Masterclass **(30 marks)**

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

(a) **(5 marks)**

```
1  //The answer must have balanced parenthesis
2  class Hi{
3      public String speak(){return "hi";}
4  }
5  class Hello{
6      public String speak(){return "hello";}
7  }
8  class LoudHi extends Hi{
9      [???]
10 }
11 class LoudHello extends Hello{
12     [???]
13 }
14
15 public class Q1 {
16   public static void main(String[]arg){
17     assert new LoudHello().speak().equals("hello!!");
18     assert new LoudHi().speak().equals("hi!!");
19   }
20 }
```

```
public String speak(){return super.speak()+"!!";}
```

(b) **(5 marks)**

```
1  //The answer must have balanced parenthesis
2  import java.util.*;
3
4  class Printer{
5      public String sep(){return "@";}
6      final public <T> String format(List<T> l){
7          String res="";
8          for(T t : l){res += t + sep();}
9          return res;
10     }
11 }
12 [???]
13 public class Q2 {
14   public static void main(String[]arg){
15   Printer p=new HashPrinter();
16   assert p.format(Arrays.asList(1,2,3)).equals("1#2#3#");
17   }
18 }
```

```
class HashPrinter extends Printer{
public String sep(){return "#";}}
```

(c) **(5 marks)**

```
1  //The answer must have balanced parenthesis
2  class A{
3      public int f;
4      A(int f){this.f=f;}
5  }
6  [???]
7
8  public class Q3 {
9    public static void main(String[]arg){
10    B b=new B(2,"Hi");
11    A a=b;
12    assert a.f==2 && b.g.equals("Hi");
13   }
14 }
```

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

(d) **(5 marks)**

```
1  //The answer must have balanced parenthesis
2
3  interface Counter{
4    int nextNum();
5  }
6
7  public class Q4 {
8    public static void main(String[] arg){
9        Counter c1=[???];
10       assert "10,11".equals(c1.nextNum()+","+c1.nextNum());
11       assert "12,13".equals(c1.nextNum()+","+c1.nextNum());
12   }
13 }
```

```
new Counter(){int i=10;
public int nextNum(){return i++;} }
```

(e) **(5 marks)**

```
1  //The answer must have balanced parenthesis
2
3  import java.awt.Color;
4
5  interface HasColor{
6      default Color color(){return Color.BLUE;}
7  }
8  interface HasWeight{
9      [???]
10 }
11 class Whale implements HasColor,HasWeight{
12     public String toString(){
13       return "Whale[" + this.color() + ";" + this.weight() + "]";}
14 }
15 public class Q5 {
16     public static void main(String[]arg){
17       assert "Whale[java.awt.Color[r=0,g=0,b=255];100000]"
18         .equals(new Whale().toString());
19     }
20 }
```

```
default int weight(){return 100000;}
```

(f) **(5 marks)**

```
1  //The answer must have balanced parenthesis
2  //Hard!!
3  [???]
4
5  public class Q6 {
6    public static void main(String[]arg){
7        try{throw new Ex();}
8        catch(Ex ex){assert false;}
9        catch(Error e){assert true;}
10   }
11 }
```

```
class Ex extends Exception{public Ex(){throw new Error();}}
```

4. Java 8 **(30 marks)**

(a) For each of the following code snippets, write an equivalent version of such code without using the Java 8 features. Write in the style that was possible/recommended before Java 8.

(i) **(4 marks)**

```
List<String>res1=new ArrayList<>(Arrays.asList("foo","bar","beer"));
res1.sort((s1,s2)->s1.charAt(0)-s2.charAt(0));
// res1.sort requires a Comparator<String>
```

```
List<String>res1=new ArrayList<>(Arrays.asList("foo","bar","beer"));
res1.sort(new Comparator<String>(){
  public int compare(String s1,String s2){
    return s1.charAt(0)-s2.charAt(0);
  }
});
```

(ii) **(4 marks)**

```
List<Integer>res2=Arrays.asList("foo","bar","beer").stream()
.map(s->s.length()).collect(Collectors.toList());
```

```
List<Integer>res2=new ArrayList<>();
for(String s: Arrays.asList("foo","bar","beer")){
  res2.add(s.length());
}
```

(iii) **(4 marks)**

```
List<String>res3=Arrays.asList("foo","bar","beer","qwerty").stream()
.filter(s->s.length()<4)
.filter(s->s.startsWith("b"))
.collect(Collectors.toList());
```

```
List<String>res3=new ArrayList<>();
for(String s: Arrays.asList("foo","bar","beer","qwerty")){
  if(s.length()>=4){continue;}
  if (!s.startsWith("b")){continue;}
  res3.add(s);
}
```

(iv) **(4 marks)**

```
String res4=Arrays.asList("foo","bar","beer","qwerty").stream()
.reduce("Elements:␣",(a,b)->a+b);
```

```
  String res4="Elements:␣";
  for(String s: Arrays.asList("foo","bar","beer","qwerty")){
    res4+=s;
  }
```

(b) **(6 marks)** By writing neatly in the box below, update `PersonInfo` code using where appropriate `Optional` from Java 8.

```
1
2  public class PersonInfo{
3
4    private long id;// mandatory
5
6    private Optional<Date> birth; // before could be null,
7                                  // now can be Optional.empty()
8
9    private String fullName;// mandatory
10
11   public PersonInfo(long id,String fullName, Optional<Date> birth){
12     assert fullName!=null; // still needed
13     assert birth!=null;    // birth==null would defeat the
14                            // purpose of Optional
15
16     this.id=id;
17
18     this.birth=birth; // can be null
19
20     this.fullName=fullName; // can not be null
21
22   }
23
24   public String toString(){
25
26     String res="id="+id+", fullName="+fullName;
27
28     if(birth.isPresent()){res+=", birth="+birth;}
29
30     return res;
31
32   }
33 }
```

(c) **(2 marks)** Declare an *unchecked* exception `ElementNotFound` with a constructor taking an input `String` message and calling the super-constructor.

```
1 class ElementNotFound extends RuntimeException{
2   public ElementNotFound(String message){super(message);}
3 }
```

(d) **(6 marks)** Using assertions, rewrite the code below by adding runtime checks which verify the pre/post conditions:

- Precondition: `l!=null`
- Postcondition:

  if `res==indexOf(elem,l)`, then `l.get(res).equals(elem)`

  if `indexOf(elem,l)` throws `ElementNotFound`, then
  no `i` exists such that `l.get(i).equals(elem)`

```
int indexOf(String elem,List<String>l){
  for(int i=0;i<l.size();i++){
    if(elem.equals(l.get(i))){return i;}
  }
  throw new ElementNotFound(elem+" not present in"+l);
}
```

```
int indexOf(String elem,List<String>l){
 assert l!=null; // Precondition
 // note, do not use here assertTrue or other junit features.
 for(int i=0;i<l.size();i++){
   if(elem.equals(l.get(i))){
     assert l.get(i).equals(elem);
     // possibly redundant, since the only "return" is guarded by such "if"
     return i;
   }
 }
 assert !l.contains(elem);// no i exists such that l.get(i).equals(elem)
 throw new ElementNotFound(elem+" not present in"+l);
 // keep the throwing in place, do not alter the behavior of the 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.

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.