NAME Daniel Schnell Schnod

Exam 2
CSCI 2600 Principles of Software
March 29, 2018

- DO NOT OPEN THIS EXAM UNTIL TOLD TO DO SO!
- READ THROUGH THE ENTIRE EXAM BEFORE STARTING TO WORK.
- YOU ARE ALLOWED ONLY 2 "CHEAT" PAGES. NO OTHER MATERIAL IS ALLOWED.

This exam is worth 150 points.

Make sure you have 15 pages counting this one. There are 3 parts, each including multiple questions for a total of 18 questions. If you need more room for an answer than is provided, please use the back of the page and indicate that you have done so. If you re-do a question, please make clear what is your final answer.

Be clear and brief in your explanations—rambling and lengthy answers will be penalized. All questions have short answers.

The following is for the use of graders

1. _____/45
2. _____/69
3. _____/36

TOTAL:_____/150

Name: _____

Part I. Exceptions and Assertions

Question 1. (12 pts, 2 pts each) Choose the better one: checked or unchecked exception?

a) **`Map.put`** when the key already exists in the map unchecked **checked**

b) **`Files.createDirectory`** when the directory already exists checked

c) **`CharBuffer.put`** when the buffer is full unchecked

d) **`CharBuffer.put`** when the buffer is read-only checked **unchecked**

e) **`BigDecimal.add`** when the result is inexact and cannot be stored unchecked

f) **`URL.openConnection()`** when a connection to the remote object fails unchecked **checked**

Question 2. (12 pts, 2 pts each) Circle TRUE or FALSE.

a) (TRUE/FALSE) One of the defensive programming practices is to check critical assignments with assert conditions (e.g., **`y = x + z; assert(y==x+z);`**).

b) (TRUE/FALSE) Only objects of type Throwable or subclasses of Throwable can be used as exceptions in Java.

c) (TRUE/FALSE) A **`RuntimeException`** typically indicates an unrecoverable programming error.

d) (TRUE/FALSE) Unchecked exceptions cannot be caught using the standard **`try catch finally`** syntax.

e) (TRUE/FALSE) It is better practice to check parameters of **private** methods with preconditions and assertions, rather than throw exceptions (such as for example **`IllegalArgumentException`**, or **`NullPointerException`**).

f) (TRUE/FALSE) Java assertions are special methods which have one parameter and the return type of void. assert (true)

Question 3. (15 pts) You are given code for the following method which creates a connection to the HTTP server, executes a request, reads the response, and then closes the response stream.

```
public String sendGet(){
    URL obj = new URL("http://www.google.com/search?q=CSCI2600");
    HttpURLConnection con = (HttpURLConnection) obj.openConnection();
    con.setRequestMethod("GET");
    con.setRequestProperty("User-Agent", "Mozilla/5.0");
    BufferedReader in = new BufferedReader(
            new InputStreamReader(con.getInputStream()));
    String inputLine;
    StringBuffer response = new StringBuffer();
    while ((inputLine = in.readLine()) != null) {
        response.append(inputLine);     }
    if (con.getResponseCode() != 200) return "Error";
    in.close();
    return response.toString();
}
```

Name: _____

Make the following changes to the code:

a) Rewrite the code to ensure proper handling of all required exceptions but do not handle **`java.net.ProtocolException`** in sendGet(). Always handle the most specific exception possible.

b) Properly dispose of (close) all resources

c) Do not make any other changes to the code

You may use the following information from the JDK documentation:

a) **`URL`** constructor throws **`java.net.MalformedURLException`** which is a subclass of **`java.io.IOException`**

b) Casting a reference may throw a **`java.lang.ClassCastException`**

c) **`openConnection()`**, **`readLine()`**, **`getResponseCode()`**, and **`close()`** may throw a **`java.io.IOException`**.

d) **`setRequestMethod()`** may throw **`java.net.ProtocolException`** and **`java.lang.SecurityException`** (**`java.net.ProtocolException`** is a subclass of **`java.io.IOException`**; **`java.lang.SecurityException`** is a subclass of **`java.lang.RuntimeException`**)

e) **`setRequestProperty()`** may throw **`java.lang.IllegalStateException`** and **`java.lang.NullPointerException`** (**`java.lang.IllegalStateException`** is a subclass of **`java.lang.RuntimeException`**)

f) **`getInputStream()`** may throw **`java.io.IOException`** and **`java.net.UnknownServiceException`** (**`java.net.UnknownServiceException`** is a subclass of **`java.io.IOException`**)

Write the entire updated source code of method **`sendGet`** () below:

```
public String sendGet() {
    try {
        URL obj = new URL("http://google.com");
    catch (MalformedURLException e) {
        throw new RuntimeException("bad URL");
    }
    HTTPURLConnection con = (HTTPURLConnection) obj.openConnection();
    try {
        con.setRequestMethod("GET");
    catch (ProtocolException e) {
        throw new RuntimeException("bad request");
    }
}
```

DNF

Name: _____

Name: _____

Question 4. (6 pts) Consider 5 specifications of function **`int binarySearch(int[] a, int val)`**.

Spec A: requires: **`a`** is sorted in ascending order, **`val`** is in **`a`**
returns: **`index`** such that **`a[index] = key`**

Spec B: requires: **`a`** is sorted in ascending order
returns: **`index`** such that **`a[index] = val`**
throws: **`IllegalArgumentException`** if **`key`** is not in **`a`**

Spec C: requires: **`val`** is in **`a`**
returns: smallest **`index`** such that **`a[index] = key`**

Spec D: returns: smallest **`index`** such that **`a[index] = key`** if **`key`** is in **`a`**,
and **`-1`** if **`key`** is not in **`a`**

Spec E: returns: smallest **`index`** such that **`a[index] = key`** if **`key`** is in **`a`**
throws: **`KeyNotFoundException`** if **`key`** is not in **`a`**

Order the specifications from best choice to worst choice (from the point of view of a client of this spec). Note, that **`KeyNotFoundException`** is not related by inheritance to **`IllegalArgumentException`**. Explain your answer. Use one-line bullets.

Answer: E, B, D, C, A

- E is > B because B requires more in the precondition, B > D because of the throws clause, D > C because C has requires, and C > A because A requires more than C

Part II. Overloading, Subtyping, and Equality

Question 5. (12 pts) Consider the Java hierarchies and client below. At the designated places, write what gets printed.

```
class X { … }
class Y extends X { … }
class Z extends Y { … }
class A {
    X m(Object o) { System.out.println("AO!"); return null; }
    X m(Z z) { System.out.println("AZ!"); return null; }
    Y m(Y y){ System.out.println("AY!"); return null; }
}
class B extends A {
    X m(Z z) { System.out.println("BZ!"); return null; }
    Z m(X x) { System.out.println("BX!"); return null; }
}
class C extends B {
    Y m(Z z) { System.out.println("CZ!"); return null; }
    Z m(X x) { System.out.println("CX!"); return null; }
}
```

Name: _____

```
A a = new A();
A ab = new B();
A ac = new C();
Y y = new Z();
Object o = new Z();
Z z = new Z();

X x = a.m(o); // What gets printed here?
o = ab.m(z); // What gets printed here?
x = ac.m(y); // What gets printed here?
```

A O!
B Z!
A Y!

Now consider these slightly different Java hierarchies and client below:

```
class X { … }
class Z { … }
class Y extends X { … }
class W extends Z { … }
class V extends W { … }
class A {
   void m(Z z, Y y) { System.out.println("AZY!"); }
}
class B extends A {
   void m(Z z, X x) { System.out.println("BWX!"); }
}
class C extends B {
   void m(W w, X x) { System.out.println("CWX!"); }
}
class D extends C {
   void m(Z z, Y y) { System.out.println("DZY!"); }
}
A a = new C();
Z z = new W();
Y y = new Y();
B b = new D();
W w = new V();
X x = new Y();
V v = new V();

a.m(w,y); // What gets printed here?
b.m(z,x); // What gets printed here?
b.m(w,y); // What gets printed here?
```

C W X!
B W X!
D Z Y!

AZY!

Name: _____

Question 6. (12 pts, 2 pts each) Circle TRUE or FALSE.

a) (TRUE/FALSE) A true subtype can always be implemented using Java subclassing.

b) (TRUE/FALSE) In Java, an overriding method is guaranteed to be a function subtype of the method that it overrides.

c) (TRUE/FALSE) If **`equals(Object)`** is reflexive and non symmetric, then it is guaranteed that **`equals(Object)`** is non transitive.

d) (TRUE/FALSE) In Java, it is required that if two objects are unequal according to the **`equals(Object)`** method, then calling the **`hashCode`** method on each of the two objects must produce distinct integer values.

e) (TRUE/FALSE) If you defined **`class Person {public String first; public String last; public boolean equals(Person p) {return this.first==p.first && this.last==p.last;}}`** then **`@Override public int hashCode() { return super.hashCode(); }`** is a legal hash function (by legal, we mean that it does not violate the consistency property).

f) (TRUE/FALSE) A function can be a function subtype of another function only if it overrides it or is overridden by it.

Questions 7 – 10 below are based on the following code for a Person with the first and last name.

```
public class Person {
    protected final String first;
    protected final String last;
    public Person(String first, String last) {
        this.first = first;
        this.last = last;
    }
    @Override public boolean equals(Object o) {
        if (!(o instanceof Person))
          return false;
        Person p = (Person)o;
        return p.first == first && p.last == last;
    }
    // Remainder implements hashCode(), first(),last(), etc.
}
```

Now, let's extend this class adding the notion of a employee who is the person with the salary:

```
public class Employee extends Person {
    protected final int salary;
    public Employee(String first, String last, int salary) {
        super(first, last);
        this.salary = salary;
    }
    // Remainder omitted
}
```

Name: _____

Question 7. (5 pts) In our first attempt, we implement **`Employee.equals`** as follows.

```
@Override public boolean equals(Object o) {
    return super.equals(o);
}
```

(a) Is **`equals`** of **`Person`** hierarchy reflexive? If not, give a counterexample below.

Yes

(b) Is **`equals`** of **`Person`** hierarchy symmetric? If not, give a counterexample.

Yes

(c) Is **`equals`** of **`Person`** hierarchy transitive? If not, give a counterexample.

Yes

(d) List one advantage and one disadvantage of this solution.

Advantage: works for all Employee objects

Disadvantage: doesn't truly represent equality between employees as they have different salaries.

Question 8. (5 pts) In our second attempt, we implement **`Employee.equals`** as follows:

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Employee))
        return false;
    return super.equals(o) && ((Employee)o).salary == salary;
}
```

(a) Is **`equals`** reflexive? If not, give a counterexample below.

Yes

(b) Is **`equals`** symmetric? If not, give a counterexample.

No

Person a = new Person("Bob", "Jones"); Employee b = new Employee("Bob", "Jones", 5);
a.equals(b) true, b.equals(a) False

(c) Is **`equals`** transitive? If not, give a counterexample.

Yes

(d) List one advantage and one disadvantage of this solution.

Advantage - compares Employees w/ each other just fine

Disadvantage - doesn't account for differences between Person & Employee correctly

Name: _____

Question 9. (5 pts) In a third attempt we implement `Employee.equals` as follows:

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Person))
        return false;
    if (!(o instanceof Employee))
        return o.equals(this);
    return super.equals(o) && ((Employee)o).salary == salary;
}
```

(a) Is `equals` reflexive? If not, give a counterexample below.

Yes

(b) Is `equals` symmetric? If not, give a counterexample.

Yes

(c) Is `equals` transitive? Again if not, give a counterexample.

No Employee a = new Employee("A", "B", 5); Person b = new Person("A", "B");
Employee c = new Employee("A", "B", 6);
a.equals(b) ✓
b.equals(c) ✓
a.equals(c) ✗

(d) List one advantage and one disadvantage of this solution.

Advantage: Compares Employees to People just fine
Disadvantage: May run into issue when comparing Employees to Employees

Question 10. (5 pts) Our next attempt changes both `Person.equals` and `Employee.equals` (recall that `o.getClass()` returns the runtime class of receiver `o`):

```
@Override public boolean equals(Object o) { // Person
    if (o == null || o.getClass() != getClass())
        return false;
    Person p = (Person)o;
    return p.first == first && p.last == last;
}

@Override public boolean equals(Object o) { // Employee
    if (o == null || o.getClass() != getClass())
        return false;
    Employee em = (Employee)o;
    return em.first == first && em.last == last && em.salary ==
        salary;
}
```

Name: _____

(a) Is **`equals`** reflexive? If not, give a counterexample below.

Yes

(b) Is **`equals`** symmetric? If not, give a counterexample.

Yes

(c) Is **`equals`** transitive? If not, give a counterexample.

Yes

(d) List one advantage and one disadvantage of this solution.

Advantage: works for all cases w/ Employee to Employee or Person to Person

Disadvantage: cannot compare b/t Employee & Person & vice versa

Question 11. (10 pts) Finally, write code to implement the Employee class using the composition. Then answer the questions. You will need to reuse the Person class and provide an implementation for at least one constructor with three parameters (for the first name, last name, and the salary) and **`Employee.equals`**.

public class Employee extends Person {

DNF

}

Name: _____

(a) Is **equals** reflexive? If not, give a counterexample below.

Yes

(b) Is **equals** symmetric? If not, give a counterexample.

Yes

(c) Is **equals** transitive? If not, give a counterexample.

Yes

(d) List one advantage and one disadvantage of this solution.

Questions 12 – 14 below use this code.

```
import java.util.*;
// Letter represents a single, case insensitive letter in the English
// alphabet.
// @specfield value: The value of the letter.
public class Letter {

  private char c;

  // Constructs a Letter object
  // @param c gives the letter, either in lower case or upper case
  // @requires 'a' <= c <= 'z' or 'A' <= c <= 'Z'
  public Letter(char c) {
     if (!(('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z')))
        throw new IllegalArgumentException();
     this.c = c;
  }

  public int getValue() {
     return this.c;
  }


  // numChars counts the number of unique letters in a given string.
  // Note that since letters are case insensitive, upper and lower
  // case ones are treated as same.
  // For example, string "Anna" has 2 unique letters: 'A' and 'N'.

  // @requires str is not null, and contains no null elements
  // @param str a string, represented as a list of Letters
  // @returns the number of unique letters in the given string;
  // the result is >= 0 and <= 26.
  public static int numChars(List<Letter> str) {
     Set<Letter> s = new HashSet<Letter>();
```

Name: _____

```
    for (Letter l : str) {
      s.add(l);
    }
    return s.size();
  }
}
```

Question 12. (4 pts) A client calls **`numChars`**. The client is surprised when **`numChars`** returns 30. Explain how this failure happens.

Equals() and hashCode() are not defined, so the HashSet<> can contain duplicate letters

Question 13. (4 pts) Write a minimal test case that exposes this failure. (By minimal test case, we mean one that creates/uses a minimal number of **`Letter`** objects.)

```
@Test
public void testQuestion() {
    Letter l = "aAbBcC ... zZ";
    assertTrue(numChars(l) > 26);
}
```

Question 14. (7 pts) Add a method (or methods) to the **`Letter`** class to prevent this failure. *You are not allowed to modify the specification or any of the existing methods in* **`Letter`**. Show exact code for each method you add. (You may find **`Character`**'s static methods **`char toLowerCase(char ch)`** and **`char toUpperCase(char ch)`** useful.)

```
public void changeToLower(List<String> lst) {
    for (Letter l : str) {
        l = toLowerCase(l);
    }
}
```

Name: _____

Part III. Testing

Question 15. (12 pts, 2 pts each) Circle TRUE or FALSE.

a) (TRUE/FALSE) A def-use pair can never refer to the same node (e.g., (2,2) would never be a valid def-use pair). X = X + y includes LHS + RHS

b) (TRUE/FALSE) Let **I1** and **I2** be two correct implementations of a given specification. A test suite that achieves 100% branch coverage on **I1**, is guaranteed to achieve 100% branch coverage on **I2**.

c) (TRUE/FALSE) A test suite written against a specification that uses preconditions to specify valid inputs, would leave uncovered code that checks inputs, and throws exceptions when those inputs are invalid.

d) (TRUE/FALSE) Coverage target All-du-paths implies coverage target All-defs. All du ⇒ All use ⇒ All def

e) (TRUE/FALSE) Tests for white-box testing can be developed before the code is written. (black box)

f) (TRUE/FALSE) Regression testing would require you to perform equivalence partitioning and select test inputs from each equivalence class.

Consider the specification for **next6**. Questions 16 – 18 below concern **next6**.

```
/**
 * @requires: k is non-negative
 * @returns: m such that m ≥ k and m - k < 6 and m%6 == 0
 */
public static int next6(int k)
```

Question 16. (9 pts) Write 2 JUnit tests and give a brief description of what you are testing.

Here is an example test (you cannot reuse this one):

```
@Test
public void testFromCacheNext6() {
  int k = 2;
  assertEquals(6,FindNumber.next6(k));
  assertEquals(6,FindNumber.next6(k));
}
```

Description: This tests the class of input when the output value is placed in cache and then retrieved from cache.

```
@Test
public void testNext6One(){
    int k=5;
    int m=6;

    assertTrue( next6(k) == m &&
                next6(k) > k);
}
//tests that return value m ≥ k
```

```
@Test
public void testNext6two(){
    for(int i=1; i<6; i++){
        assertTrue( next6(i) %6 == 0)
    }
}
//tests m%6, which implies all m are divisible by 6
```

Name: _____

Now, let's look at one implementation of **next6**

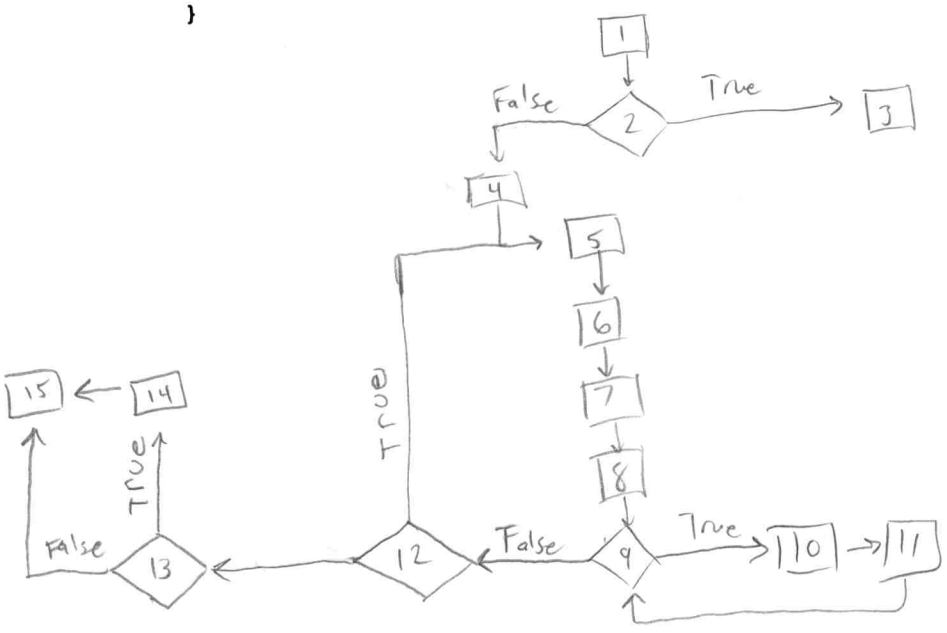
```
Map<Integer, Integer> cache = new HashMap<Integer, Integer>();
int capacity = 5;

int next6(int k) {
1  int num, remainder, sum, last;
2  if (cache.containsKey(k)) {
3    return cache.get(k);
   }
4  num = k - 1;
   do {
5    num = num + 1;
6    remainder = num;
7    sum = 0;
8    last = num % 10;
9    while (remainder > 0) {
10     sum = sum + remainder % 10;
11     remainder = remainder / 10;
     }
12 } while (!(sum % 3 == 0 && last % 2 == 0));
13 if (cache.size() < capacity) {
14   cache.put(k, num);
   }
15 return num;
}
```

k=1

1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12 →

Sum=1, last=1 → 5, 6, 7, 8, 9, 10, 11



Name: _____

Question 17. (10 pts) In the space above adjacent to the code, draw the control-flow graph (CFG) for the **`next6`** routine. Remember to have each statement as a separate node and then perform the contraction of edges, as we did in class.

Question 18. (5 pts)

a) Write the total number of statements and the percentage of statement coverage (for the **`next6`** routine only) achieved by running the following group of tests: **`next6(1), next6(2), next6(3), next6(4), next6(5)`**

15, 100%

b) Write the total number of branch edges and the percentage of branch coverage (for the **`next6`** routine only) achieved by running the following group of tests: **`next6(1), next6(2), next6(2), next6(4), next6(6)`**

17 branches 100%