PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

DECEMBER 2017 EXAMINATIONS

CSC 207 H1F
Instructor(s): Shorser, Gries

Duration—2 hours

No Aids Allowed

You must earn at least 21 out of 52 marks (40%) on this final examination in order to pass the course. Otherwise, your final course grade will be no higher than 47%.

Student Number: |_|_|_|_|_|_|_|_|_|_|

Last (Family) Name(s): ______________________

First (Given) Name(s): ______________________

UTORid: ______________________

---

*Do **not** turn this page until you have received the signal to start.*
In the meantime, please read the instructions below *carefully*.

This Final Examination paper consists of 6 questions on 17 pages (including this one), printed on both sides of the paper. *When you receive the signal to start, please make sure that your copy of the paper is complete and fill in your name and student number above.*

- Comments and Javadoc are not required except where indicated, although they may help us mark your answers.
- If you use any space for rough work, indicate clearly what you want marked.

MARKING GUIDE

# 1:	______/10
# 2:	______/10
# 3:	______/ 6
# 4:	______/ 8
# 5:	______/ 8
# 6:	______/10
TOTAL:	______/52

Good Luck!

Question 1. [10 MARKS]

A.	checked `Exception`	H.	`RuntimeException`	O.	`super();`
B.	UML diagram	I.	`List`	P.	`return isEmpty == true;`
C.	CRC cards	J.	Encapsulation	Q.	`this.idCount = new Integer("5");`
D.	Casting	K.	Model-View-Controller	R.	Interface
E.	Generics	L.	Strategy	S.	Abstract class
F.	Array	M.	`==`	T.	Default value
G.	ArrayList	N.	`equals`	U.	Print the stack trace

The following are descriptions of items listed above. On the underscore beside each description, write the letter of the item it describes best. No two descriptions should be matched with the same item above.

Part (a) [1 MARK] ______ A line of code that will be written automatically by the compiler if you do not include it in a constructor.

Part (b) [1 MARK] ______ A line of code that could appear in the body of a constructor as the second line, after the line from part (a).

Part (c) [1 MARK] ______ An object that is thrown under exceptional circumstances that cannot be prevented from occurring.

Part (d) [1 MARK] ______ An object that is thrown under exceptional circumstances that could have been prevented with different code.

Part (e) [1 MARK] ______ A representation of code that summarizes the names and types of variables; names, arguments, and return types of methods; accessibility modifiers, inheritance relationships, and dependancy relationships, among other features.

Part (f) [1 MARK] ______ A representation of code that has not yet been written that focusses on the responsibilities of each class and how various classes interact.

Part (g) [1 MARK] ______ A generic interface that extends `Collection`.

Part (h) [1 MARK] ______ A boolean operator that checks to see if the memory addresses of two objects are the same.

Part (i) [1 MARK] ______ The design pattern that separates an algorithm from its context.

Part (j) [1 MARK] ______ If one of these classes appears in your program, at least one of its subclasses must also be included.

Question 2. [10 MARKS]

Part (a) [5 MARKS]

Assume that the following main method appears in a class that is not given in the Supplementary Code, but exists in the same package. Method main includes the following code:

```
Ticket t1 = new Ticket("concert", "AB");
Ticket t2 = new TrainTicket("Montreal", "Toronto", "CD");
TrainTicket t3 = new TrainTicket("Vancouver", "Halifax", "EF");
t1.returnTicket();
System.out.println(t1.getNumSales()); // prints: 3
System.out.println(t2.getNumSales()); // prints: 3
System.out.println(t3.getNumSales()); // prints: 3
t2.returnTicket();
t2.sellTicket("GH"); // prints: This ticket is for sale again
System.out.println(t1.getNumSales()); // prints: 4
System.out.println(t2.getNumSales()); // prints: 4
System.out.println(t3.getNumSales()); // prints: 4
```

Without modifying method main, what is the smallest modification you can make to the Supplementary Code that will result in the following output:

```
1
1
1
This ticket is for sale again
1
2
1
```

Include the line number(s) of any code you modify and explain why your modification will work.

Part (b) [5 MARKS]

Write two classes. The first class is a generic class called `Organizer` that can store an unlimited number of objects of the same type. The constructor for `Organizer` should require one of these objects as an argument. Include a `toString` method that returns the contents of the organizer as a single string.

The second class is not generic. It is called `TicketOrganizer` and only stores instances of the `Ticket` class from the Supplementary Code, or subclasses of `Ticket`. Override the `toString` method from `Organizer` so that it now returns a string that contains only the number of `Tickets` currently stored in the `TicketOrganizer`.
You may need to write additional methods to satisfy the requirements.

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number, and refer to this answer on the original question page.

Question 3. [6 MARKS]

Part (a) [3 MARKS]

In the Supplementary Code, at the back of this booklet, identify one example of a dependency. Write the line(s) on which it appears and also describe the dependancy in words.

Part (b) [3 MARKS]

Implement the dependency injection design pattern to eliminate the dependency you found in part (a) of this question. You should only rewrite the line(s) of code that you want to change. Please include the line numbers to indicate where you changed existing code. If you want to introduce a new line of code, you are not required to give it a number.

Question 4. [8 MARKS]

As we discussed in class, the SOLID design principles are:

1. Single responsibility principle
2. Open/closed principle
3. Liskov substitution principle
4. Interface segregation principle
5. Dependency inversion principle

Part (a) [4 MARKS]

In the SOLID principles of design, the L stands for the Liskov Substitution Principle. In the Supplementary Code, at the end of this booklet, see if you can find code that violates the Liskov Substitution Principle. If you do not find any such code, explain how the Liskov Substitution Principle works. If, instead, you find a violation of this principle, write the line(s) on which it appears and explain how these lines violate the principle. Only one violation will be graded.

Part (b) [4 MARKS]

Choose one of the other four SOLID principles of design (other than Liskov) and explain how it works. If you would like to discuss an example, you can refer to any code from this course including Assignment 1 code, Assignment 2 starter code, or any code that was posted to the "Lectures" part of the website.

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number, and refer to this answer on the original question page.

Question 5. [8 MARKS]

Part (a) [4 MARKS]

The Supplementary Code includes a class called `Ticket` and a subclass called `TrainTicket`. Write a new class **`AirplaneTicket`** as a subclass of `Ticket`, and have that class also keep track of the source and destination cities, and also include a seat zone, one of "Economy", "Business", and "First". All tickets by default start at "Economy". Provide a way to upgrade.

Part (b) [4 MARKS]

The following is a separate class in the same package called `TicketFactory`. Rewrite method **`getTicket`** so that it can return an instance of **`TrainTicket`** or an instance of **`AirplaneTicket`** based on a key that the user sends in.

```
public class TicketFactory {

  public static Ticket getTicket(String key, String buyer, String city1, String city2) {
    if (key.equals("TrainTicket")) {
      return new TrainTicket(city1, city2, buyer);
    } else {
      return null;
    }
  }
}
```

Question 6. [10 MARKS]

Part (a) [3 MARKS]

Write the substring(s) of `xx1yXzxyX4x2yzX` that satisfy each of the following regular expressions. If no such substrings exist, write "None".

(i) `x.{2}\d`

(ii) `a*y*(([XYZ]).*)\2$`

(iii) `(x|y|z)?(X|Y|Z)[xyz][xyz]`

Part (b) [4 MARKS]

IPv4 (Internet Protocol Version 4) addresses are written as four sets of four 1's and 0's. Each set is separated by a period.

Write a regular expression that is satisfied by any IPv4 address that is written inside a set of parentheses where the last set of four digits is identical to the first set of four digits. Here is an example of a string that will satisfy your regular expression: `(1011.0101.0001.1011)` Your solution should not match any other IPv4 addresses nor strings with digits other than 0's or 1's.

Part (c) [3 MARKS]

Fill in each of the squares with a character so that each resulting horizontal string satisfies the regular expressions to its left and each vertical string satisfies the regular expression above it. Draw this symbol: ϕ in any box that you think should have a whitespace character.

	`([abc])\1[efg]`	`[l-p]{2}.`	`m?[aeiou](\s)*r`	`(n|m).\1`
`(ani|si)m+`				
`a?[left](\s){2,4}`				
`[^ergo]a.*`				

Use the space on this "blank" page for scratch work, or for any answer that did not fit elsewhere.
Clearly label each such answer with the appropriate question and part number, and refer to this answer on the original question page.

Short Java APIs:

```
class Throwable:
    // the superclass of all Errors and Exceptions
    Throwable getCause() // returns the Throwable that caused this Throwable to get thrown
    String getMessage() // returns the detail message of this Throwable
    StackTraceElement[] getStackTrace() // returns the stack trace info
class Exception extends Throwable:
    Exception()          // constructs a new Exception with detail message null
    Exception(String m) // constructs a new Exception with detail message m
    Exception(String m, Throwable c) // constructs a new Exception with detail message m caused by c
class RuntimeException extends Exception:
    // The superclass of exceptions that don't have to be declared to be thrown
class Error extends Throwable
    // something really bad
class Object:
    String toString() // returns a String representation
    boolean equals(Object o) // returns true iff "this is o"
interface Comparable<T>:
    int compareTo(T o) // returns < 0 if this < o, = 0 if this is o, > 0 if this > o
interface Iterable<T>:
    // Allows an object to be the target of the "foreach" statement.
    Iterator<T> iterator()
interface Iterator<T>:
    // An iterator over a collection.
    boolean hasNext() // returns true iff the iteration has more elements
    T next() // returns the next element in the iteration
    void remove() // removes from the underlying collection the last element returned or
                  // throws UnsupportedOperationException
interface Collection<E> extends Iterable<E>:
    boolean add(E e) // adds e to the Collection
    void clear() // removes all the items in this Collection
    boolean contains(Object o) // returns true iff this Collection contains o
    boolean isEmpty() // returns true iff this Collection is empty
    Iterator<E> iterator() // returns an Iterator of the items in this Collection
    boolean remove(E e) // removes e from this Collection
    int size() // returns the number of items in this Collection
    Object[] toArray() // returns an array containing all of the elements in this collection
interface List<E> extends Collection<E>, Iteratable<E>:
    // An ordered Collection. Allows duplicate items.
    boolean add(E elem) // appends elem to the end
    void add(int i, E elem) // inserts elem at index i
    boolean contains(Object o) // returns true iff this List contains o
    E get(int i) // returns the item at index i
    int indexOf(Object o) // returns the index of the first occurrence of o, or -1 if not in List
    boolean isEmpty() // returns true iff this List contains no elements
    E remove(int i) // removes the item at index i
    int size() // returns the number of elements in this List
class ArrayList<E> implements List<E>
class Arrays
    static List<T> asList(T a, ...) // returns a List containing the given arguments
interface Map<K,V>:
    // An object that maps keys to values.
    boolean containsKey(Object k) // returns true iff this Map has k as a key
    boolean containsValue(Object v) // returns true iff this Map has v as a value
    V get(Object k) // returns the value associated with k, or null if k is not a key
    boolean isEmpty() // returns true iff this Map is empty
    Set<K> keySet() // returns the Set of keys of this Map
    V put(K k, V v) // adds the mapping k -> v to this Map
    V remove(Object k) // removes the key/value pair for key k from this Map
    int size() // returns the number of key/value pairs in this Map
    Collection<V> values() // returns a Collection of the values in this Map
class HashMap<K,V> implements Map<K,V>
class File:
    File(String pathname) // constructs a new File for the given pathname
class Scanner:
    Scanner(File file) // constructs a new Scanner that scans from file
    void close() // closes this Scanner
    boolean hasNext() // returns true iff this Scanner has another token in its input
    boolean hasNextInt() // returns true iff the next token in the input is can be
                         // interpreted as an int
    boolean hasNextLine() // returns true iff this Scanner has another line in its input
    String next() // returns the next complete token and advances the Scanner
    String nextLine() // returns the next line and advances the Scanner
    int nextInt() // returns the next int and advances the Scanner
class Integer implements Comparable<Integer>:
    static int parseInt(String s) // returns the int contained in s
        throw a NumberFormatException if that isn't possible
    Integer(int v) // constructs an Integer that wraps v
    Integer(String s) // constructs on Integer that wraps s.
    int compareTo(Object o) // returns < 0 if this < o, = 0 if this == o, > 0 otherwise
    int intValue() // returns the int value
class String implements Comparable<String>:
    char charAt(int i) // returns the char at index i.
    int compareTo(Object o) // returns < 0 if this < o, = 0 if this == o, > 0 otherwise
    int compareToIgnoreCase(String s) // returns the same as compareTo, but ignores case
    boolean endsWith(String s) // returns true iff this String ends with s
    boolean startsWith(String s) // returns true iff this String begins with s
    boolean equals(String s) // returns true iff this String contains the same chars as s
    int indexOf(String s) // returns the index of s in this String, or -1 if s is not a substring
    int indexOf(char c) // returns the index of c in this String, or -1 if c does not occur
    String substring(int b) // returns a substring of this String: s[b .. ]
    String substring(int b, int e) // returns a substring of this String: s[b .. e)
    String toLowerCase() // returns a lowercase version of this String
    String toUpperCase() // returns an uppercase version of this String
    String trim() // returns a version of this String with whitespace removed from the ends
class System:
    static PrintStream out // standard output stream
    static PrintStream err // error output stream
    static InputStream in // standard input stream
class PrintStream:
    print(Object o) // prints o without a newline
    println(Object o) // prints o followed by a newline
class Pattern:
   static boolean matches(String regex, CharSequence input) // compiles regex and returns
                                                            // true iff input matches it
   static Pattern compile(String regex) // compiles regex into a pattern
   Matcher matcher(CharSequence input) // creates a matcher that will match
                                       // input against this pattern
class Matcher:
   boolean find() // returns true iff there is another subsequence of the
                  // input sequence that matches the pattern.
   String group() // returns the input subsequence matched by the previous match
   String group(int group) // returns the input subsequence captured by the given group
```

```
                               //during the previous match operation
   boolean matches() // attempts to match the entire region against the pattern.
class Observable:
    void addObserver(Observer o) // adds o to the set of observers if it isn't already there
    void clearChanged() // indicates that this object has no longer changed
    boolean hasChanged() // returns true iff this object has changed
    void notifyObservers(Object arg) // if this object has changed, as indicated by
        the hasChanged method, then notifies all of its observers by calling update(arg)
        and then calls the clearChanged method to indicate that this object has no longer changed
    void setChanged() // marks this object as having been changed
interface Observer:
    void update(Observable o, Object arg) // called by Observable's notifyObservers;
        // o is the Observable and arg is any information that o wants to pass along
```

Regular expressions:

Here are some predefined character classes.

`.`	`Any character`
`\d`	`A digit: [0-9]`
`\D`	`A non-digit: [^0-9]`
`\s`	`A whitespace character: [ \t\n\x0B\f\r]`
`\S`	`A non-whitespace character: [^\s]`
`\w`	`A word character: [a-zA-Z_0-9]`
`\W`	`A non-word character: [^\w]`
`\b`	`A word boundary: any change from \w to \W or \W to \w`

Here are some quantifiers:

Quantifier	Meaning
`X?`	`X, once or not at all`
`X*`	`X, zero or more times`
`X+`	`X, one or more times`
`X{n}`	`X, exactly n times`
`X{n,}`	`X, at least n times`
`X{n,m}`	`X, at least n; not more than m times`

```java
public class Ticket {

  private static int numSales;
  private String event;
  private String buyer;
  private boolean isForSale;

  public Ticket(String event, String buyer) {
    this.event = event;
    this.buyer = buyer;
    isForSale = false;
    numSales++;
  }

  public void returnTicket() {
    buyer = "";
    isForSale = true;
  }

  public void sellTicket(String buyer) {
    this.buyer = buyer;
    isForSale = false;
    numSales++;
  }

  public String toString() {
    return 'this ticket for ' + event + ' belongs to ' + buyer;
  }

  public int getNumSales() {
    return numSales;
  }
}
```

```java
public class TrainTicket extends Ticket {

  private String fromCity;
  private String toCity;

  public TrainTicket(String fromCity, String toCity, String buyer) {
    super('train ride', buyer);
    this.fromCity = fromCity;
    this.toCity = toCity;
  }

  public void returnTicket() {
    System.out.println('This ticket is for sale again');
  }

  public String getToCity() {
    return toCity;
  }

  public void setToCity(String toCity) {
    this.toCity = toCity;
  }
}
```

CONT'D...

```
public class TwoWayTrip {

  private TrainTicket departTicket;
  private TrainTicket returnTicket;
  private String startDate;
  private String endDate;

  public TwoWayTrip(
      String startDate, String endDate, String fromCity, String toCity, String buyer) {
    departTicket = new TrainTicket(fromCity, toCity, buyer);
    returnTicket = new TrainTicket(toCity, fromCity, buyer);
    this.startDate = startDate;
    this.endDate = endDate;
  }

  public void printItinerary() {
    System.out.println("Start date: " + startDate + ", End date: " + endDate);
  }
}
```