CSE 1030 Introduction to Computer Science II

Test 2

10:30 — 11:30, May 1, 2009

Last name First name Student number

Instructions

- No questions are allowed during the test.
- This is a closed book test. No aids are permitted.
- Answer each question in the space provided.
- Make sure that you have answered all 5 questions.
- Manage your time carefully—you do not need to answer the questions in order.
- Do not leave during the last 5 minutes; stay seated and do not talk until all of the tests have been collected.

Question	Total marks available	Mark
1	12	
2	18	
3	10	
4	15	
5	15	
Total	70	

1. [12 marks] Suppose a `Person` has-an age, has-a name, and has-a date of birth:

```
public class Person
{
  private int age;
  private String name;
  private Date birthday;
  // …
}
```

a) [4 marks] Which of the three attributes do not suffer from privacy leaks? Explain why.
Recall that a privacy leak is when a client obtains a reference to a private attribute.
age, because it is a primitive type and Java uses pass-by-value.
name, because it is immutable and immutable objects can be freely shared.

b) [2 marks] Should `Person` use composition with any of the attributes? Which ones? Explain why.
Recall that composition implies ownership; that is the Person object is responsible for the attribute.
birthday, because Date is mutable

c) [3 marks] Complete the following constructor so that `Person` does not have any privacy leaks; the first attribute is done for you. *You do not need to use mutator methods.*

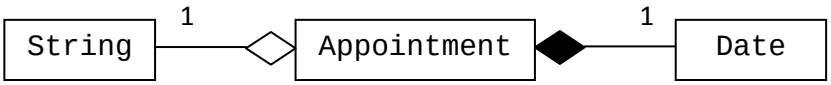
```
public Person(int age, String name, Date birthday)
{
  this.age = age;

  this.name = name;
  this.birthday = new Date(birthday.getTime());

}
```

d) [3 marks] Complete the following two accessors so that `Person` does not have any privacy leaks.

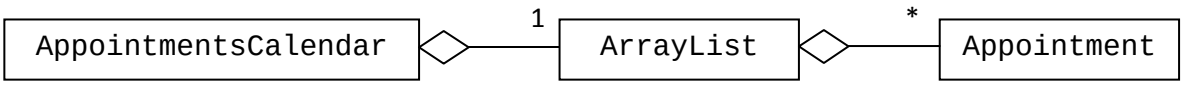
```
public String getName()
{
  return this.name;

}

public Date getDate()
{
  return new Date(this.birthday.getTime());

}
```

2. [18 marks] Questions 2, 3, and 5 refer to the classes `Appointment` and `AppointmentsCalendar`. You will find the implementation of both of these classes on the last page of this test.

a) [3 marks] Complete the UML class diagram for the `Appointment` class.



b) [4 marks] Complete the UML class diagram for the `AppointmentsCalendar` class.



c) [2 marks] `AppointmentsCalendar` promises that it always maintains the appointments in chronological order (sorted by time of the appointment). Can the implementation that was provided to you keep such a promise? Why or why not?
The intent of this question was to see if you realized that the iterator method exposes references to Appointments; I accepted two answers for full marks, although only the first one below is completely correct.

Yes, even though references to Appointment objects are exposed by the iterator method, Appointment is immutable; thus, there is no way to change the date of an appointment once it is added to the calendar. [*Note: Appointment is not final, so it is possible for a client to extend Appointment and circumvent immutability, so really, the answer is no*].

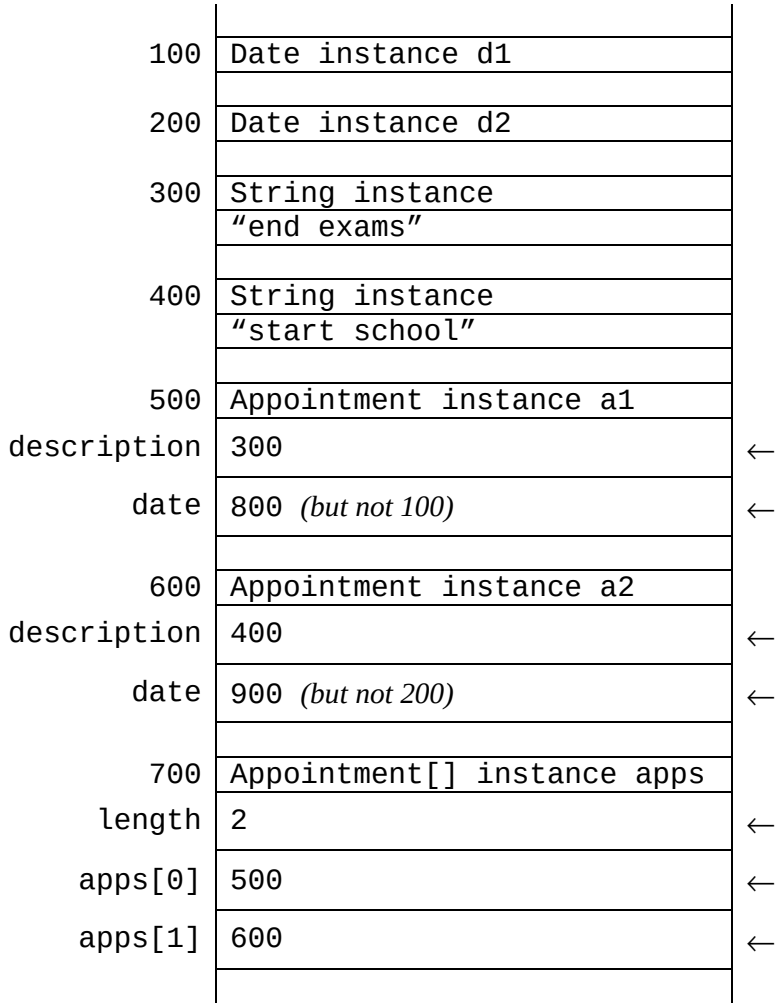
No, because the iterator method exposes references to appointments that might be changed by clients.

d) [2 marks] What type of copying does the `AppointmentsCalendar` copy constructor use?

deep copy

e) [7 marks] Complete the parts of the memory diagram indicated with the ← symbol for the following fragment of code. You may extend the memory diagram if you wish.

```
Date d1 = new Date(109, 5, 2); // June 2, 2009
Date d2 = new Date(109, 8, 9); // Sept 9, 2009

Appointment a1 = new Appointment("end exams", d1);
Appointment a2 = new Appointment("start school", d2);

Appointment[] apps = new Appointment[2];
apps[0] = a1;
apps[1] = a2;
```



The constructor for Appointment creates a new Date instance; thus, the date attribute of a1 and a2 are deep copies of d1 and d2 (and not simply aliases for d1 and d2).

3. [10 marks] This question refers to the `AppointmentsCalendar` class. You will find the implementation of the class on the last page of this test.

   Suppose that the implementer of `AppointmentsCalendar` had chosen to use an array of appointments instead of a list. Because `AppointmentsCalendar` implements `Iterable<Appointment>`, it must supply a method that returns a class that implements `Iterator<Appointment>`. Describe how you would implement a class that represents an iterator on an array of appointments. *You should use one or two short sentences to describe the purpose of each attribute. You should use one or two short sentences to describe the purpose, name, and return type (if any) of each method. Do not provide details regarding the constructors of your class.*

   attribute 1: `int next`, to hold the index of the next element in the iteration

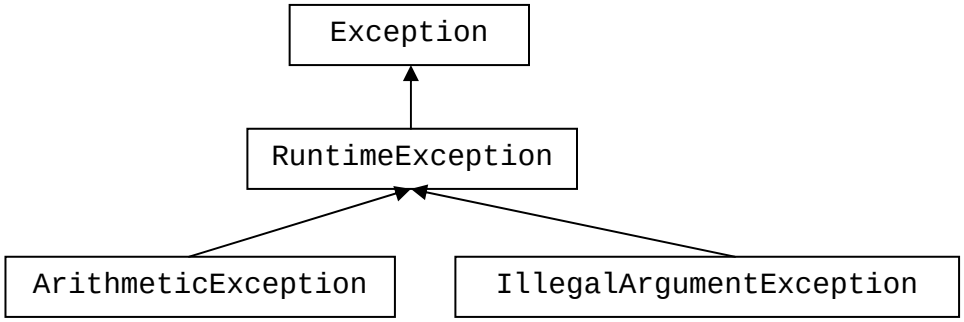
   attribute 2: `Appointment[] app`, a copy of the array to iterate over

   method 1: `boolean hasNext`, returns true if there is another element in the iteration and false otherwise

   method 2: `Appointment next`, returns the next element of the iteration if it exists and throws an exception otherwise

   method 3: `void remove`, removes the last element returned by the iterator; not supported by all iterators

4. [15 marks] Consider the following inheritance hierarchy.



Fill in the blanks to complete the sentences for (a)—(g); (h) and (i) are short answer questions.

a) `RuntimeException` is a __*subclass*_____ of `Exception`.

b) `RuntimeException` is a __*superclass*_____ of `ArithmeticException`.

c) *_Runtime/Arithmetic/IllegalArgumentException_* is substitutable for `Exception`.

d) *_nothing*_____ is substitutable for `ArithmeticException`.

e) *_RuntimeException* _____ must call a constructor of `Exception`. It uses the keyword *_super*_____ to call a constructor of `Exception`.

f) *_ArithmeticException*_____ is allowed to call a constructor of `ArithmeticException`. It uses the keyword __*this*_____ to call a constructor of `ArithmeticException`.

g) `IllegalArgumentException` can call non-overridden methods of `Exception` that have the access modifiers __*public or protected*_____.

h) A class `X` defines a method `doSomething` that says it may throw an exception of type `RuntimeException`. List all of the types of exceptions that `doSomething` can throw.

Runtime/Arithmetic/IllegalArgumentException

i) A class `Y` extends `X` and overrides `doSomething`. The `Y` version of `doSomething` can say that it throws which types of exceptions without surprising clients of `X`?

Runtime/Arithmetic/IllegalArgumentException

5. [15 marks] This question refers to the `Appointment` class. You will find the implementation of the class on the last page of this test. Complete the implementation of the three methods in `PeriodicAppointment` class below. *Where possible, you should use the facilities of the superclass, and you should prevent privacy leaks.*

```
/*
 * A class that represents an appointment that regularly
 * repeats (examples: weekly meeting, monthly mortgage
 * payment deadline, birthdays).
 */
public class PeriodicAppointment extends Appointment
{
  protected Date next;  // next time that the appointment will occur

  public PeriodicAppointment(String description, Date date, Date next)
  {
    super(description, date);
    this.next = new Date(next.getTime());

  }

  public PeriodicAppointment(PeriodicAppointment other)
  {
    this(other.getDescription(), other.getDate(), other.next);

  }

  /*
   * Two PeriodicAppointments are considered equal if their
   * descriptions are the same, start times are the same,
   * and next times are the same.
   */
  @Override public boolean equals(Object obj)
  {
    boolean eq = super.equals(obj);
    if(eq) {
      PeriodicAppointment other = (PeriodicAppointment)obj;
      if(!this.next.equals(other.next)) {
        eq = false;
      }
    }
    return eq;
  }
}
```

Appendix: The `Appointment` and `AppointmentsCalendar` classes

```
/*
 * A class representing an appointment event. An appointment has
 * a description and a starting date/time. Implements the Comparable
 * interface so that clients can sort appointments based on the
 * starting time.
 */
public class Appointment implements Comparable<Appointment> {
     protected String description;  // reason for the appointment
     protected Date   date;         // date/time of the appointment

     public Appointment(String description, Date date) {
          this.description = description;
          this.date = new Date(date.getTime());
     }

     public Appointment(Appointment other) {
          this.description = other.description;
          this.date = new Date(other.date.getTime());
     }

     public String getDescription()
     { return this.description; }

     public Date getDate()
     { return new Date(this.date.getTime()); }

     @Override public int compareTo(Appointment other)
     { return this.date.compareTo(other.date); }

     /*
      * Two appointments are considered equal if both are not null
      * and their descriptions and start date/times are the same.
      */
     @Override public boolean equals(Object obj) {
          boolean eq = false;
          if(obj != null && this.getClass() == obj.getClass()) {
               Appointment other = (Appointment) obj;
               eq = this.description.equals(other.description) &&
                    this.date.equals(other.date);
          }
          return eq;
     }
}
```

```
/*
 * A class representing a calendar of zero or more appointments. The
 * class always maintains the appointments in chronological order
 * (i.e. sorted by the starting date and time of the appointment).
 */

public class AppointmentsCalendar implements Iterable<Appointment>
{
     private ArrayList<Appointment> apps;

     // default constructor
     public AppointmentsCalendar()
     {
          this.apps = new ArrayList<Appointment>();
     }

     // copy constructor
     public AppointmentsCalendar(AppointmentsCalendar other)
     {
          this.apps = new ArrayList<Appointment>();
          for(Appointment a : other.apps)
          {
               this.apps.add(new Appointment(a));
          }
     }

     /*
      * Adds an appointment to the calendar, maintaining
      * the appointments in order sorted by the appointment date.
      */
     public void addAppointment(Appointment a)
     {
          // implementation not shown
     }

     @Override
     public Iterator<Appointment> iterator()
     {
          return this.apps.iterator();
     }
}
```