COMP-202 – Introduction to Computing 1
Final Exam - Version 1

April 20, 2005

Examiners: Marc Lanctot, Ernesto Posse, Alex Batko

Associate examiner: Joseph Vybihal

Last name	
First name	
Id number	
Section	☐ 1 (Marc Lanctot) ☐ 2 (Ernesto Posse) ☐ 3 (Alex Batko)

Instructions

- No notes, notebooks, textbooks, calculators, cell phones, pagers, laptops or handheld computers are allowed in this exam.
- **For section 1 (multiple choice) mark your answer in the front page. The rest of the questions are to be answered in the booklet.**
- **Read the questions carefully.**
- Language translation dictionaries are permitted.
- Answers may be given in either English or French.

Grading

Section 1: Multiple choice

Section 1 mark: / 20

Question	Q1	Q2	Q3	Q4	Q5	Q6
Your answer						
Mark	/ 2	/ 3	/ 3	/ 2	/ 3	/ 3

Question	Q7	Q8
Your answer		
Mark	/ 2	/ 2

Section 2: Short problems

Section 2 mark: / 40

Question	Q1	Q2	Q3	Q4
Mark	/4	/8	/14	/14

Section 3: Programming

Section 3 mark: / 40

Question	Q1	Q2
Mark	/16	/24

Total: / 100

Terminology reference table

Term	Synonyms/Meaning
Instance	Object
Object instantiation	Object creation, creating an instance of a class
Instance variable	Attribute, non-static variable
Class variables	Static variable
Normal method	Non-static method
Class method	Static method
Method signature	Method header
Method invocation	Method call, sending a message to an object (normal methods), asking an object to perform a task (normal methods).
Argument	Parameter
Base class	Parent class, super class
Derived class	Child class, subclass
Inherits from	Extends

1 Multiple choice

Question 1 Which of the following creates an object?

a) `class A { String s; }`
b) `int n = 1729;`
c) `n = gen.nextInt();`
d) `String getName()`
e) `Scanner s = new Scanner(System.in);`

Answer e)

Question 2 Which of the following is true?

a) A class is the same thing as an object
b) A class defines a data-type
c) All objects in a class have the same state
d) The values of the attributes of a class are shared by all its instances
e) A method is a class with parameters

Answer b)

Question 3 Suppose that there is a class `BankAccount` which has an attribute `balance`, and a class `CheckingAccount` that extends `BankAccount` and has an attribute `monthly_fee`. Which of the following statements is false?

a) `CheckingAccount` cannot extend any other class.
b) If `balance` is shadowed (redefined in `CheckingAccount`,) then it is still possible to access `balance` from `BankAccount.`
c) It is not possible to access `monthly_fee` directly in class `BankAccount`.
d) A `BankAccount` object is the same as a `CheckingAccount` object where the `monthly_fee` is `0.0`.
e) If there is a class `SavingsAccount` which extends `BankAccount` and you modify the `balance` of a `CheckingAccount` object, then the `balance` of `SavingsAccount` is not updated.

Answer d)

Question 4 Which of the following is true?

a) Both an interface and an abstract class can be instantiated
b) Neither an interface nor an abstract class can be instantiated
c) An interface can be instantiated, while an abstract class cannot
d) An interface cannot be instantiated, while an abstract class can
e) Abstract classes cannot have subclasses

Answer b)

Question 5 Which of the following is false?

a) If a method expects a parameter of type `T`, then it can be passed as argument any instance of the parent class of `T`.
b) If a method expects a parameter of type `T`, then it can be passed as argument any instance of `T` or any of its subclasses.
c) If a method has a return type `T`, then it can return an instance of a subclass of `T`.
d) If `v` is of type `S`, and `S` is a subclass of `T`, then the casting `(T)v` is possible.
e) If `v` is of type `T`, and `S` is a subclass of `T`, then the casting `(S)v` is possible, provided that `v` has been assigned a reference of type `S.`

Answer a)

Question 6 Consider the following program fragment:

```
Scanner scanner = new Scanner(System.in);     // stmt 1
int num = 0, sum = 0;                         // stmt 2
try {                                         // stmt 3
    while (true) {                            // stmt 4
        num = scanner.nextInt();              // stmt 5
        if (num == -1) {                      // stmt 6
            break;                            // stmt 7
        }
        else if (num < 0) {                   // stmt 8
            throw new Exception();            // stmt 9
        }
        sum = sum + num;                      // stmt 10
    }
}
catch (Exception e) {                         // stmt 11
    System.our.println(“Oops! ” + e);         // stmt 12
}
System.out.println(“sum = ” + sum);           // stmt 13
```

Suppose that this fragment is executed, and the user types 34, and then types -82. In which order will the statements be executed?

a) 1, 2, 3, 4, 11, 12, 13
b) 1, 2, 3, 4, 5, 6, 8, 10, 13
c) 1, 2, 3, 4, 5, 6, 8, 10, 4, 5, 6, 7, 11, 12, 13
d) 1, 2, 3, 4, 5, 6, 8, 10, 4, 5, 6, 8, 9, 11, 12, 13
e) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

Answer d)

Question 7 It is a good idea to make attributes of a class private because:

a) Because making attributes public forces them to be shared by all instances of its class.
b) Because they can be inherited.
c) Because private attributes are accessible from other classes, allowing other classes to modify the state of an object any time.
d) Because public attributes are accessible from other classes, which could modify the state of an object in some unwanted way.
e) Because otherwise we could define only public and protected methods in the class.

Answer d)

Question 8 In large applications it is often desirable to have general *helper* functions that do not implement actions performed on, or by objects. This is, in addition to having normal methods it is useful to have object-independent methods that perform general tasks. For example, in a statistical application, a helper function could be a method that returns the average of an array of integers. Since helper functions do not act on objects, it is unnecessary to create an object to perform the task. What is the best way to organize a program that uses helper functions and avoids creating objects of the class(es) containing helper functions?

a) Use an array of helper methods.
b) Define an abstract class (or classes) containing the helper functions implemented as abstract methods.
c) Define one or more helper classes containing the helper functions implemented as static methods.
d) Define a base class containing the helper methods, and making sure that all other classes extend this base class.
e) Define a helper interface, making sure that any class that needs a helper function implements that interface.

Answer c)

2 Short problems

Question 1 Consider the following class definition:

```
class Person
{
    String name;
    Person(String n) { name = n; }
    void wakeup(Brain x)
    {
        System.out.print( name );
        x.activate( “starts” );
    }
}
public class Test
{
    public static void main(String[] args)
    {
        Person a = new Person(“Newton”);
        Brain b = new Brain();
        a.wakeup(b);
    }
}
```

This program is missing a class `Brain`. Write a definition of class `Brain` so that the program prints “`Newton starts the day`”.

Answer Class brain must have a method activate that expects a String as argument and doesn't return anything:

```
class Brain
{
    void activate(String s)
    {
        System.out.println(“ ” + s + “ the day”);
    }
}
```

Question 2 What will the following print if method `test1` in class `Test` is invoked?

```
class Banner {
    String message;
    void set(String w) { message = w; }
    String get() { return message; }
}
class Test {
    static String w;
    static void test1()
    {
        w = “1 ”;
        String x = “2 ”;
        Banner y = new Banner();
        y.set(“3 ”);
        String[] z = new String[2];
        z[0] = “4 ”;
        z[1] = “5 ”;
        process( x, y, z );
        System.out.println( w + x + y.get() + z[0] + z[1] );
    }
    static void process(String a, Banner b, String[] c)
    {
        System.out.println( w + a + b.get() + c[0] + c[1] );
        w = “6 ”;
        a = “7 ”;
        b.set(“8 ”);
        c[0] = “9 ”;
    }
}
```

Answer static variables are accessible in all static methods in the class. Parameters of a method have local scope. Values whose type is an object are passed by reference. So are arrays.

```
1 2 3 4 5
6 2 8 9 5
```

Question 3 Suppose that you are developing a program to keep track of all the items in stock at some store. There are many different kinds of items, but all of them have an ID number, a name and a cost. Therefore, each item belongs to a class which implements the following interface:

```
public interface IdentifiableItem
{
    public int getId();
    public String getName();
    public double getCost();
}
```

Given an array of `IdentifiableItems`, an item x is said to be unique if there is no other item in the array with the same name and ID as x. Write a method that receives an array of `IdentifiableItems`, and returns the number of unique items. This is, duplicate items should not be counted.

Answer There are several possible algorithms. A simple one would check for each element in the array if there is another one (in a different position) with the same Id and name. If so, it is duplicated and therefore it should not be counted. If the element is not duplicated, it is unique, and therefore a counter is incremented.
The full implementation is in the next page.

```
static int countUniques(IdentifiableItem[] list)
{
    int counter, main_index, secondary_index;
    boolean unique;
    counter = 0;
    main_index = 0;
    while (main_index < list.length)
    {
        unique = true;
        secondary_index = 0;
        while (secondary_index < list.length)
        {
            if (secondary_index != main_index)
            {
                if (list[main_index].getId() == list[secondary_index].getId()
                 && list[main_index].getName() == list[secondary_index].getName
                {
                    unique = false;
                }
            }
            secondary_index++;
        }
        if (unique)
        {
            counter++;
        }
        main_index++;
    }
    return counter;
}
```

Question 4 Write a method called `merge` that takes as argument two *sorted* arrays of integers a and b, and returns a *sorted* array of integers that combines all of the values stored in a and b. For example, if a is [3,4,7,7,9,10,13] and b is [2, 2, 2, 5, 8, 9, 11, 11] then the method returns an array [2, 2, 3, 4, 5, 7, 7, 8, 9, 9, 10, 11, 11, 13]. Hint: take advantage of the fact that the two given arrays are already sorted.

Answer Create an array big enough to hold both arrays, and traverse both arrays in parallel. At each time, decide which is the smallest element between the current element from a and the current element from b. Put that smallest in the resulting array, and move only the index of the array from which the smallest was chosen. Repeat until all elements have been traversed.

```
static int[] merge(int[] a, int[] b)
{
    int total_length = a.length + b.length;
    int[] result = new int[total_length];
    int i, j, k, min;
    i = 0;
    j = 0;
    k = 0;
    while (k < total_length)
    {
        if (i >= a.length)
        {
            min = b[j];
            j++;
        }
        else if (j >= b.length)
        {
            min = a[i];
            i++;
        }
        else if (a[i] <= b[j])
        {
            min = a[i];
            i++;
        }
        else
        {
            min = b[j];
            j++;
        }
        result[k] = min;
        k++;
    }
    return result;
}
```

3 Programming problems

Question 1 A *Matryoshka doll* is a Russian nesting doll. A set of Matryoshka dolls consists of a wooden figure which can be pulled apart to reveal another similar but of course slightly smaller figure inside. The smaller figure will in turn have another figure inside it, and so on, such that there are usually six or more nested dolls in a set. In other words, a Matryoshka doll *has a* Matryoshka doll inside it.

Implement this recursive concept of a Matryoshka doll by writing a class `MatryoshkaDoll`. A Matryoshka doll of size n *has a* (stores a) Matryoshka doll of size $n - 1$ inside it. The constructor of this class must take n the number of dolls in the set as an argument, and it must create the nested set of dolls by creating a Matryoshka doll with $n - 1$ dolls inside, or no dolls if $n = 0$. The class should also have two additional methods; one named `count` that recursively counts and returns the number of dolls in the set; and one named `toString` that recursively traverses the set of dolls and returns the word "pretty" for each doll.

Using this class, the following statements...

```
MatryoshkaDoll prettyDolls = new MatryoshkaDoll(5);
System.out.println( "Number of dolls: " + prettyDolls.count() );
System.out.println( "My pretty dolls: " + prettyDolls );
```

...should produce the following output:

```
Number of dolls: 5
My pretty dolls: pretty pretty pretty pretty pretty
```

You do not need to write a driver class with a mein method for this problem.

Answer Since a Matryoshka doll *has a* doll inside it, there is an *aggregation* relationship between doll objects: each doll object must have a reference to another doll object. In addition there must be a constructor and two methods: count and toString.
The full implementation is on the next page.

```
public class MatryoshkaDoll
{
    private MatryoshkaDoll doll_inside;

    public MatryoshkaDoll( int numberOfDolls )
    {
        if ( numberOfDolls > 0 )
        {
            doll_inside = new MatryoshkaDoll( numberOfDolls - 1 );
        }
        else
        {
            doll_inside = null;
        }
    }

    public int count()
    {
        if ( doll_inside == null )
        {
            return 0;
        }
        else
        {
            return doll_inside.count() + 1;
        }
    }

    public String toString()
    {
        if ( doll_inside == null )
        {
            return "";
        }
        else
        {
            return "pretty " + doll_inside.toString();
        }
    }
}
```

Question 2 Many computer applications deal with representations of geometrical *shapes*, such as circles, triangles, rectangles, etc. Suppose that you are writing an application that deals with such geometrical shapes. Initially the application deals only with *circles*, *rectangles* and *composite* shapes. But, new types may be added later.

- A rectangle *is a* shape that *has* a width and a height, and the location (x, y) of its top-left corner (for simplicity, assume that shapes are aligned with the X and Y axis, and not rotated.)
- A circle *is a* shape that *has* a *radius*, and the location (x, y) of its center.
- A composite shape *is a* shape that *has* up to 100 shapes. It can contain any shape: circles, rectangles and other composite shapes. It should be possible to *add* a shape (any kind of shape) to a composite shape.
- Any shape *has a bounding box*. The bounding box of a shape is the smallest rectangle that contains the shape.

Write a set of classes to represent these shapes, such that for any shape it is possible to ask the shape what is its bounding box.

It should be possible to define new types of shapes so that your classes work without modifications.

Hint: it might be easier to compute the bounding box of a composite shape whenever you add a new shape to the composite. The composite shape asks the shape being added which is its bounding box, and using this, it updates its overall bounding box.

You may use the static methods `min` and `max` from the `Math` class, that compute the smallest (resp. largest) of two given numbers. Their signatures are

```
public static double min(double a, double b)
public static double max(double a, double b)
```

You do not need to write a driver class with a mein method for this problem.

Answer There are several possible solutions, but all of them share the same characteristics:

- There must be classes: Shape, Rectangle, Circle and CompositeShape,
- Rectangle, Circle and CompositeShape must be subclasses of Shape (inheritance is the “is-a” relationship.)
- All classes must have at least a method that returns the shape’s bounding box, which is a Rectangle.
- The CompositeShape class must have a collection of Shapes as attribute (aggregation) and a method to add any Shape.
- None of the solutions rely on Shape-specific attributes, only on the fact that they can give you their bounding box. This is fundamental in order to allow the possible definition of new types of Shapes.

Here we provide three different solutions: two basic solutions (as expected,) and a solution that deals with the subtle problem of adding shapes to composite shapes inside other composite shapes (for bonus points.)

Solution 1 Since every shape has a bounding box, Shape objects should have a reference to its bounding box (a Rectangle), and a method that returns this reference. This is a default method, and is inherited by the subclasses, so there is no need for Rectangle, Circle and CompositeShape to redefine it. The bounding box reference is protected rather than private so that the subclasses can modify it. The subclasses define the value of the bounding box in their constructor, and CompositeShape updates its bounding box every time a new shape is added to it.

```
class Shape
{
    protected Rectangle bounding_box;
    public Rectangle getBoundingBox()
    {
        return bounding_box;
    }
}

class Rectangle extends Shape
{
    private double x, y, width, height;
    public Rectangle(double x, double y, double width, double height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        bounding_box = this;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
}

class Circle extends Shape
{
    private double x, y, radius;
    public Circle(double x, double y, double radius)
    {
        this.x = x;
        this.y = x;
        this.radius = radius;
        bounding_box = new Rectangle(x - radius, y + radius,
                                     2 * radius, 2 * radius);
    }
}
```

```
class CompositeShape extends Shape
{
    private Shape[] shape_list;
    private int first_available;

    public CompositeShape()
    {
        shape_list = new Shape[100];
        first_available = 0;
        bounding_box = new Rectangle(0, 0, 0, 0);
    }
    public void add_shape(Shape s)
    {
        if (first_available < shape_list.length)
        {
            shape_list[first_available] = s;
            first_available++;
            updateBoundingBox(s);
        }
    }
    private void updateBoundingBox(Shape s)
    {
        Rectangle new_shape_box = s.getBoundingBox();
        double s_x1 = new_shape_box.getX();
        double s_y1 = new_shape_box.getY();
        double s_x2 = s_x1 + new_shape_box.getWidth();
        double s_y2 = s_y1 - new_shape_box.getHeight();
        double my_x1 = bounding_box.getX();
        double my_y1 = bounding_box.getY();
        double my_x2 = my_x1 + bounding_box.getWidth();
        double my_y2 = my_y1 - bounding_box.getHeight();
        double new_x1 = Math.min(s_x1, my_x1);
        double new_y1 = Math.max(s_y1, my_y1);
        double new_x2 = Math.max(s_x2, my_x2);
        double new_y2 = Math.min(s_y2, my_y2);
        bounding_box = new Rectangle(new_x1, new_y1,
                                     new_x2 - new_x1, new_y1 - new_y2);
    }
}
```

Solution 2 An alternative approach is that shapes do not necessarily need to *store* the actual bounding box, they only need the *capability* to provide it to whomever asks for it. Therefore, Shape does not have a bounding box attribute, but it has an abstract method that returns the bounding box. This method is abstract to force the subclasses to define it, since computing the bounding box is specific to each type of shape. The Shape abstract class could also be an interface. In this approach, the composite shape as any other, doesn't store the bounding box, so it needs to recompute its entire bounding box every time it is asked for it.

```
abstract class Shape
{
    public abstract Rectangle getBoundingBox();
}

class Rectangle extends Shape
{
    private double x, y, width, height;
    public Rectangle(double x, double y, double width, double height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    public Rectangle getBoundingBox()
    {
        return this;  // or new Rectangle(x, y, width, height);
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
}

class Circle extends Shape
{
    private double x, y, radius;
    public Circle(double x, double y, double radius)
    {
        this.x = x;
        this.y = x;
        this.radius = radius;
    }
    public Rectangle getBoundingBox()
    {
       return new Rectangle(x - radius, y + radius, 2 * radius, 2 * radius);
    }
}
```

```
class CompositeShape extends Shape
{
    private Shape[] shape_list;
    private int first_available;
    public CompositeShape()
    {
        shape_list = new Shape[100];
        first_available = 0;
    }
    public void add_shape(Shape s)
    {
        if (first_available < shape_list.length)
        {
            shape_list[first_available] = s;
            first_available++;
        }
    }
    public Rectangle getBoundingBox()
    {
        Rectangle bounding_box = new Rectangle(0, 0, 0, 0);
        int i = 0;
        while (i < first_available)
        {
            Rectangle r = shape_list[i].getBoundingBox();
            bounding_box = combineBoxes(bounding_box, r);
            i++;
        }
        return bounding_box;
    }
    private Rectangle combineBoxes(Rectangle a, Rectangle b)
    {
        double a_x1 = a.getX();
        double a_y1 = a.getY();
        double a_x2 = a_x1 + a.getWidth();
        double a_y2 = a_y1 - a.getHeight();
        double b_x1 = b.getX();
        double b_y1 = b.getY();
        double b_x2 = b_x1 + b.getWidth();
        double b_y2 = b_y1 - b.getHeight();
        double new_x1 = Math.min(a_x1, b_x1);
        double new_y1 = Math.max(a_y1, b_y1);
        double new_x2 = Math.max(a_x2, b_x2);
        double new_y2 = Math.min(a_y2, b_y2);
        return new Rectangle(new_x1, new_y1,
                             new_x2 - new_x1, new_y1 - new_y2);
    }
}
```

Solution 3 This solution deals with a sublte problem (we were not expecting you to figure this out:) What happens when we add a shape to a composite shape that is already inside some (parent) composite shape? The answer is that it depends on when do you compute the bounding box. The two previous solutions provided two basic alternatives: in solution 1, whenever a shape was added to the composite shape, the bounding box was updated; in solution 2, the entire bounding box is recomputed every time we invoke the getBoundingBox method. The second solution does not suffer from the problem described above, since the entire bounding box of the parent is recomputed whenever we demand what is the parent's bounding box. The problem only arises in the first solution.

So, how do we take advantage of the first solution and avoid the problem of modifying inner shapes? The solution is that if each CompositeShape knows who its "parent" is (the CompositeShape that contains it,) then whenever we update a shape, a message is sent to its parent to update its bounding box as well, which in turn may ask its own parent to update itself. This recursive process stops when a composite shape has no parent. In this solution, it is necessary to assign a parent to any composite shape. This could be done in the constructor, or in a setter method. Here we simply specify it in the constructor. If a composite shape has no parent, its parent attribute is set to null.

In this solution, classes Shape, Rectangle and Circle are the same as in solution 1.

```
class Shape
{
    protected Rectangle bounding_box;

    public Rectangle getBoundingBox()
    {
        return bounding_box;
    }
}

class Rectangle extends Shape
{
    private double x, y, width, height;

    public Rectangle(double x, double y, double width, double height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
        bounding_box = this;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
}
```

```
class Circle extends Shape
{
    private double x, y, radius;

    public Circle(double x, double y, double radius)
    {
        this.x = x;
        this.y = x;
        this.radius = radius;
        bounding_box = new Rectangle(x - radius, y + radius,
                                     2 * radius, 2 * radius);
    }
}
```

```
class CompositeShape extends Shape {
    private Shape[] shape_list;
    private int first_available;
    private CompositeShape parent;
    public CompositeShape()
    {
        shape_list = new Shape[100];
        first_available = 0;
        bounding_box = new Rectangle(0, 0, 0, 0);
        parent = null;
    }
    public CompositeShape(CompositeShape container)
    {
        shape_list = new Shape[100];
        first_available = 0;
        bounding_box = new Rectangle(0, 0, 0, 0);
        parent = container;
    }
    public void add_shape(Shape s)
    {
        if (first_available < shape_list.length)
        {
            shape_list[first_available] = s;
            first_available++;
            updateBoundingBox(s);
        }
    }
    private void updateBoundingBox(Shape s)
    {
        Rectangle new_shape_box = s.getBoundingBox();
        double s_x1 = new_shape_box.getX();
        double s_y1 = new_shape_box.getY();
        double s_x2 = s_x1 + new_shape_box.getWidth();
        double s_y2 = s_y1 - new_shape_box.getHeight();
        double my_x1 = bounding_box.getX();
        double my_y1 = bounding_box.getY();
        double my_x2 = my_x1 + bounding_box.getWidth();
        double my_y2 = my_y1 - bounding_box.getHeight();
        double new_x1 = Math.min(s_x1, my_x1);
        double new_y1 = Math.max(s_y1, my_y1);
        double new_x2 = Math.max(s_x2, my_x2);
        double new_y2 = Math.min(s_y2, my_y2);
        bounding_box = new Rectangle(new_x1, new_y1,
                                     new_x2 - new_x1, new_y1 - new_y2);
        if (parent != null) parent.updateBoundingBox(s);
    }
}
```