Programación orientada a objetos. Examen diciembre 2002.

Martes 3 de diciembre de 2002.

1. Programen una clase `Oracion` capaz de contener una cadena que represente una oración. Las oraciones tienen la primera letra en mayúsculas y terminan con un punto. Es posible crear una instancia de la clase `Oracion` con un texto inicial y también cambiar el texto de la oración; en ambos casos la clase pasará la primera letra del texto a mayúsculas y agregará el punto al final si fuera necesario. Deberán programar la clase de forma que incluya:

   - Un atributo privado declarado como `String m_txt` que contendrá el texto de la oración.
   - Un constructor `Oracion(String txt)`. La cadena que se pasa al constructor como argumento es asignada como valor inicial del atributo `m_txt`.
   - Un método `void setTexto(String txt)` que asigna la cadena que se pasa como argumento al atributo `m_txt`.
   - Un método `String getTexto()` que retorna el valor del atributo `m_txt`.

   Algunos de los métodos de la clase `String` son los siguientes:

   - `char charAt(int index)` Retorna el carácter en el índice especificado.
   - `String concat(String str)` Concatena la cadena especificada al final de esta cadena.
   - `boolen endsWith(String suffix)` Prueba si esta cadena termina con el sufijo especificado.
   - `boolean equalsIgnoreCase(String anotherString)` Compara esta cadena con otra cadena, ignorando diferencias en mayúsculas y minúsculas.
   - `int length()` Retorna el largo de esta cadena.
   - `boolean startsWith(String prefix)` Prueba si la cadena comienza con el prefijo especificado.
   - `String substring(int beginIndex)` Retorna una nueva cadena que es una sub-cadena de esta cadena.
   - `String substring(int beginIndex, int endIndex)` Retorna una nueva cadena que es una sub-cadena de esta cadena.
   - `String toLowerCase()` Convierte todos los caracteres de esta cadena a minúsculas usando las reglas del localismo predeterminado.
   - `String toUpperCase()` Convierte todos los caracteres de esta cadena a mayúsculas usando las reglas del localismo predeterminado.

   Tengan en cuenta que la cadena que se pasa al constructor `Oracion()` y al método `setTexto()` no necesariamente tiene la primera letra en mayúsculas ni termina en un punto; la clase transformará la cadena para que cumpla estas restricciones, si fuera necesario, conservando el resto de los caracteres.

   15 puntos

2. Definan en forma breve y precisa invariante de clase, precondición y poscondición. Agreguen a la clase Oración del ejercicio anterior las invariantes, precondiciones y poscondiciones necesarias para asegurar que:

   - El argumento del constructor `Oracion()` y del método `setTexto()` no es una cadena en blanco.
   - El texto de la oración es siempre una cadena con por lo menos la primera letra en mayúsculas y que termina en un punto.
   - El constructor `Oracion()` y el método `setTexto()` asignan la cadena que se recibe como argumento al texto de la oración.

Identifiquen para cada una de las afirmaciones anteriores si se trata de una invariante de clase, una precondición o una poscondición.

Agreguen estas afirmaciones al código del ejercicio anterior usando la cláusula `assert` de Java. No tienen porqué escribir nuevamente toda la clase, mientras indiquen claramente dónde se inserta cada modificación.

15 puntos

3. Hagan un programa `Prueba` en Java que reciba como argumento una cadena y, usando la clase de los ejercicios anteriores, imprima en la consola la oración resultante y luego termine; es decir si se ejecuta el comando

   ```
   java –ea Prueba “hola que tal”
   ```

   se verá en la consola

   ```
   Hola que tal.
   ```

   Además del código fuente del programa deberán indicar el nombre del archivo que debe contener ese código y la sentencia de línea de comando con la que se debe compilar ese archivo para ejecutar el programa tal como aparece más arriba.

   10 puntos

4. Sea una interfaz `Printable` y una clase `Printer` definidas así:

```
interface Printable {
  public void printOn(Stream s);
}

class Printer implements Printable {
  public void printOn(Stream s); {
    …
  }
}
```

El método `printOn(Stream s)` imprime el receptor en la `Stream s`. Se pide a dos programadores que implementen una nueva clase con un método que imprima al receptor en una `Stream`. Un programador propone la clase `MyPrinterObject` y otro la clase `MyPrintableObject` declaradas así:

```
class MyPrinterObject extends Printer {
}

class MyPrintableObject implements Printable {
  private Printer p = new Printer();

  public void printOn(Stream s) {
    p.printOn(s);
  }
}
```

¿Qué usó cada programador? ¿Qué cosas buenas y qué cosas malas, si las hubiera, tiene cada solución?

10 puntos

5. Usando las interfaces y clases del ejercicio anterior, muestren con un pequeño fragmento de código el uso del polimorfismo usando interfaces y clases.

10 puntos

6. Definan encadenamiento estático y dinámico. Usando las interfaces y clases del ejercicio 4, muestren las diferencias entre encadenamiento estático y dinámico con pequeños fragmentos de código en Java, en un caso usando interfaces, y en el otro usando clases.

10 puntos

7. Erich Gamma, autor del libro Design Patterns, afirma que "…la composición es una alternativa a la herencia; nueva funcionalidad se obtiene ensamblando o componiendo objetos para obtener funcionalidad más compleja…". El autor afirma que, en el contexto de la reutilización, toda implementación que use herencia se puede cambiar por una equivalente que use composición y delegación.

   ¿La afirmación contraria es cierta? Es decir, ¿toda solución que use composición y delegación se puede implementar también usando herencia? Justifiquen la respuesta y den un ejemplo.

10 puntos

8. Un objeto puede tener más de un tipo y varios objetos de diferentes clases pueden tener el mismo tipo. Muestren con un pequeño fragmento de código de Java un ejemplo de un objeto con más de un tipo y otro con varios objetos de diferentes clases y el mismo tipo.

10 puntos