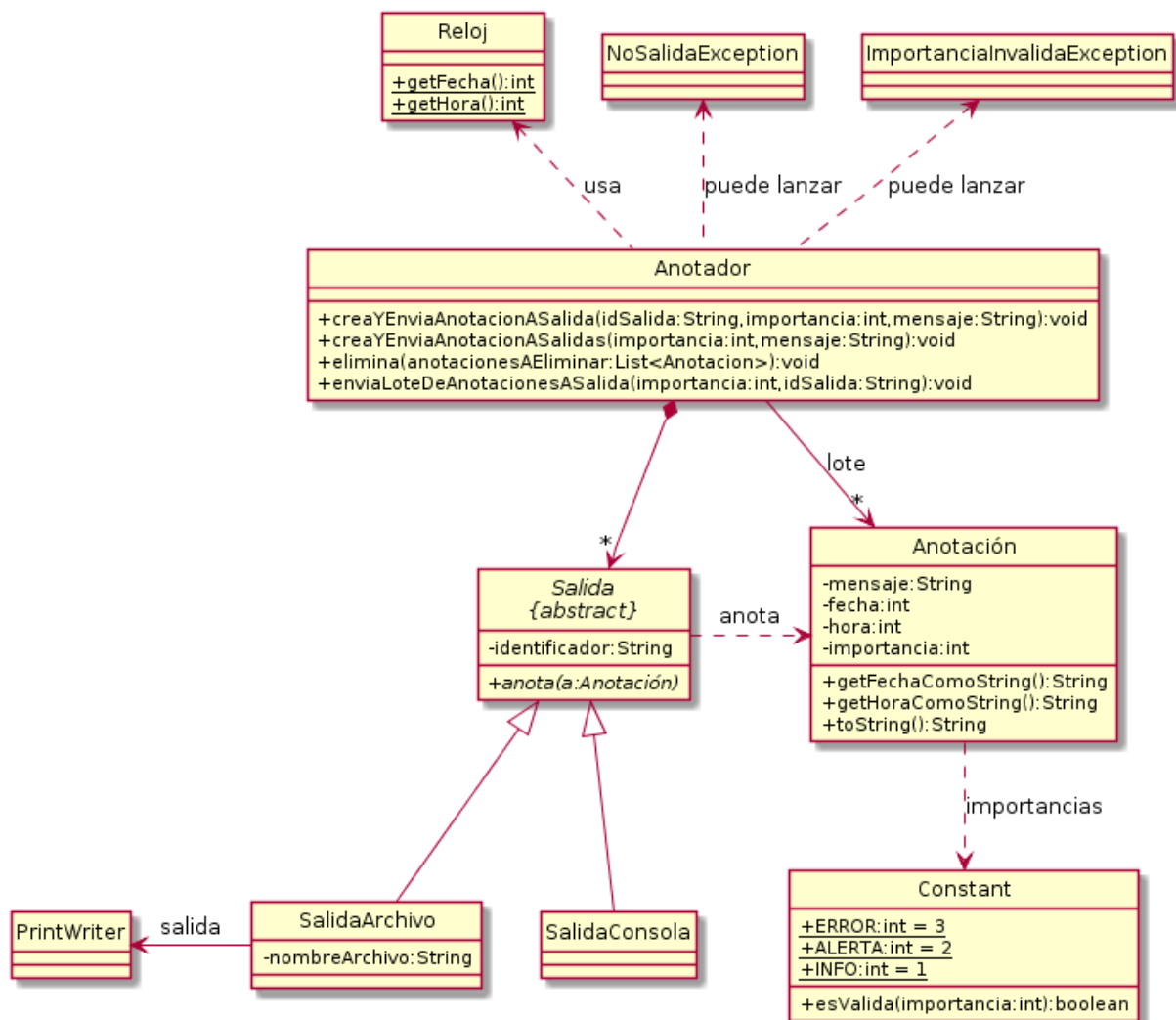


Curso 2015-2016. Cuatrimestre Primavera. 2 de Junio de 2016
Programación Orientada a Objetos (POO). Examen Final.
Publicación notas provisionales: viernes 17 de junio.
Revisión de examen: lunes 20 de junio. Lugar: D6-221 (D6, 2ª planta). Hora: 11:00 am.

Cuando los programas profesionales desean mostrar por pantalla, o guardar en archivos, mensajes de texto con detalles de la ejecución (las “trazas de ejecución”), utilizan unas herramientas especiales diseñadas para tales fines: los anotadores o registradores (“Loggers”, en inglés). Se trata de herramientas que disponen de una potente funcionalidad que permite generar trazas de ejecución con diferentes niveles de detalle, enviarlas a diferentes lugares (por ej., pantalla, archivos), generarlas en diferentes formatos, etc... Y todo ello con una sobrecarga de código mínima.

La figura que sigue muestra el diagrama de clases UML correspondiente a una parte de una de estas herramientas.



Cada objeto instancia de `Anotacion` contiene la información correspondiente a una de las anotaciones realizadas en la traza de ejecución. Incluye un nivel de importancia (identificado por un número entero; observad los valores en la clase `Constant` del diagrama), la fecha y la hora en la que la anotación se generó y un mensaje. Cada objeto instancia de `Anotacion` dará lugar en las salidas a una línea de texto en la que se indicará la importancia, la hora, la fecha y el mensaje.

El objeto instancia de `Anotador` es el encargado de crear los objetos instancia de `Anotacion` y de escribir la parte de traza correspondiente a cada anotación en los destinos elegidos. El objeto instancia de `Anotacion` incorpora varios objetos instancias de subclases de la clase `Salida`.

Cada objeto instancia de una subclase de `Salida` dispone de un identificador único y representa a una salida específica: consola y archivo de disco.

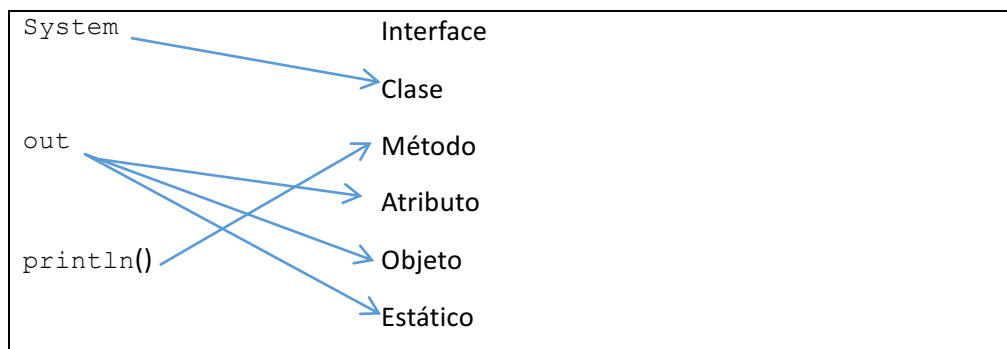
El objeto instancia de `Anotador` dispone de dos modos de envío de anotaciones a las salidas: el modo directo y el modo en lotes. En el primer modo, el anotador crea una nueva anotación y la envía inmediatamente después de crearla. En el modo por lotes, las anotaciones se guardan por orden de llegada en un contenedor. Así, cuando se requiere, se envían las anotaciones o una parte de ellas a la/s salida/s que se elige.

PREGUNTA 1 (1,5 PUNTOS) Responde a las siguientes preguntas:

- a) **(0,5 PUNTOS)** ¿Por qué los métodos de la clase `Reloj` son `static`? Justifica brevemente (5 líneas como máximo) tu respuesta.

Porque no se necesita objeto (estado) alguno de la clase `Reloj`.

- b) **(0,5 PUNTOS)** En la invocación: `System.out.println()`, relaciona con flechas sus componentes con los términos de la derecha:



- c) **(0,5 PUNTOS)** Decid cuál (o cuáles) de las siguientes 5 líneas de código son correctas, teniendo en cuenta el diagrama de clases UML de la aplicación mostrado anteriormente:

```
Salida s = new Salida(); ..... errónea  
Salida s = new SalidaConsola(); ..... correcta  
SalidaConsola s = new Salida(); ..... errónea  
SalidaArchivo s = new SalidaConsola(); ..... errónea  
Salida s = new Salida("identificador"); ..... errónea
```

PREGUNTA 2 (1 PUNTO) A partir del diagrama de clases de la figura, escribid para todas las clases la parte de código de la definición que comienza con el `"public class...."` correspondiente a la declaración de los todos los atributos de las clases, incluyendo aquellos que aparecen al implementar las relaciones mostradas en el diagrama, excepto para la clase `PrintWriter`. NO debéis añadir nada más en el código de estas clases en vuestras respuestas a esta pregunta.

```
public class Anotacion {  
    private String mensaje;  
    private int fecha, hora, importancia;  
}
```

```

public class Anotador {
    private Map<String, Salida> salidas;    //Clave: Identificador de cada salida
    private List<Anotacion> lotes;
}

public class Constant {
    public static final int ERROR = 3;
    public static final int ALERTA = 2;
    public static final int INFO = 1;
}

public abstract class Salida {
    private String identificador;
}

public class SalidaArchivo extends Salida {
    private String nombreArchivo;
    private PrintWriter salida;
}

public class SalidaConsola extends Salida {}

public abstract class Reloj {}

public class NoSalidaException extends Exception {}

public class ImportanciaInvalidaException extends Exception {}

```

PREGUNTA 3 (1 PUNTO) Implementad el constructor de la clase `Anotador`, cuya cabecera se muestra a continuación:

```
public Anotador();
```

```

public Anotador() {
    this.salidas = new HashMap<>();
    this.lotes = new ArrayList<>();
}

```

Además, proponed e implementad un constructor para cada una de las clases siguientes: `Salida` y `SalidaArchivo`.

```

public Salida(String identificador) {
    this.identificador = identificador;
}

public SalidaArchivo(String identificador, String nombreArchivo) throws IOException {
    super(identificador);
    this.nombreArchivo = nombreArchivo;
    salida = new PrintWriter(nombreArchivo);
    /*o con alguna variante adecuada tipo
    new PrintWriter (new FileOutputStream(nombreArchivo)) ;
    new PrintWriter (new BufferedOutputStream(new FileOutputStream(nombreArchivo))) ;
    new PrintWriter (new File(nombreArchivo)) ;
    */
}

```

PREGUNTA 4 (1 PUNTO) Implementad, en la clase `Anotacion` el siguiente método:

```
public String getFechaComoString();
```

Este método, genera un `String` representando una fecha con el formato DD/MM/AAAA, en el que DD es el día, MM el mes y AAAA el año.

Este `String` debe ser generado a partir del valor del atributo `fecha` de la clase `Anotacion`. Dicho atributo es un número entero cuyos dígitos guardan una fecha según el formato AAAAMMDD. Por ejemplo, el valor entero 20160602 representa la fecha 2 de junio de 2016 y el `String` generado debería ser "02/06/2016".

```
public String getFechaComoString() {
    int dia = fecha%100;
    int mes = (fecha/100)%100;
    int anyo = (fecha/10000);
    return (dia < 10 ? "0" : "") + dia + "/" + (mes < 10 ? "0" : "") + "/" + anyo;
}
```

PREGUNTA 5 (0,5 PUNTOS) Implementad, en la clase `Anotacion`, el siguiente método:

```
public String toString();
```

Este método genera un `String` en el que se concatenan el nivel de importancia de la anotación, su fecha y hora de creación y el mensaje. A continuación se muestra un ejemplo de `String` generado por dicho método:

```
[ALERTA] 19/5/2016 18:31 Un mensaje con importancia 'alerta'
```

Asumid que la clase `Anotacion` también tiene un método:

```
public String getHoraComoString();
```

Que genera un `String` en formato HH:MM, tal y como se muestra en el ejemplo. No debéis implementarlo.

```
public String toString() {
    String txtImportancia;
    switch(importancia) {
        case Constant.IMP_ERROR:
            txtImportancia = "ERROR";
            break;
        case Constant.IMP_ALERTA:
            txtImportancia = "ALERTA";
            break;
        default:
            txtImportancia = "INFO";
            break;
    }
    return "[" + txtImportancia + "] " + getFechaComoString() + " " + getHoraComoString() + " " +
    mensaje;
}
```

PREGUNTA 6 (1 PUNTO) Implementad, en la clase `Anotador`, el siguiente método:

```
public void creaYEnviaAnotacionASalida(String idSalida, int importancia, String mensaje) throws ImportanciaInvalidaException, NoSalidaException;
```

Este método genera un objeto instancia de `Anotacion` con el nivel de importancia pasado en el argumento `importancia`, con el mensaje pasado en el argumento `mensaje` y con la hora y la fecha obtenidas a través de los métodos estáticos de la clase `Reloj`. Una vez generada la anotación la envía al objeto instancia de `Salida` cuyo identificador se pasa en el argumento `idSalida`. Si el valor del argumento `importancia` no es válido (no equivale al valor de ninguna de las importancias definidas en la clase `Constant`), el método crea y lanza una excepción `ImportanciaInvalidaException`. Si el anotador no está conectado con ninguna salida con ese identificador, el método crea y lanza una excepción `NoSalidaException`.

```
public void creaYEnviaAnotacionASalida(String idSalida, int importancia, String mensaje) throws
ImportanciaInvalidaException, NoSalidaException {
    if (Constant.esValida(importancia) == false) throw new ImportanciaInvalidaException();
    Salida s = salidas.get(idSalida);
    if (s == null) throw new NoSalidaException(idSalida);
    s.anota(new Anotacion(mensaje, Reloj.getFecha(), Reloj.getHora(), importancia));
}
```

PREGUNTA 7 (1 PUNTO) Implementad, en la clase `Anotador`, el siguiente método:

```
public void creaYEnviaAnotacionASalidas(int importancia, String mensaje) throws
ImportanciaInvalidaException;
```

Este método genera un objeto instancia de `Anotacion` con el nivel de importancia pasado en el argumento `importancia`, con el mensaje pasado en el argumento `mensaje` y con la hora y la fecha obtenidas a través de los métodos estáticos de la clase `Reloj`. Una vez generada la anotación la envía a todos los objetos instancia de `Salida`. Si el valor del argumento `importancia` no es válido, el método crea y lanza una excepción `ImportanciaInvalidaException`.

```
public void creaYEnviaAnotacionASalidas(int importancia, String mensaje) throws
ImportanciaInvalidaException {
    if (Constant.esValida(importancia) == false) throw new ImportanciaInvalidaException();
    Anotacion a = new Anotacion(mensaje, Reloj.getFecha(), Reloj.getHora(), importancia);
    for (Salida s : salidas.values()) s.anota(a);
}
```

PREGUNTA 8 (1 PUNTO) Implementad, en la clase `Anotador`, el siguiente método:

```
public void elimina(List<Anotacion> anotacionesAEliminar);
```

Este método elimina del contenedor de anotaciones para ser enviadas por lotes todas aquellas anotaciones que estén contenidas en `anotacionesAEliminar`.

Tened en cuenta que las listas tienen un método `remove` al que se le puede pasar como argumento una referencia al objeto a eliminar.

```
public void elimina(List<Anotacion> anotacionesAEliminar) {
    lotes.removeAll(anotacionesAEliminar);
}
```

PREGUNTA 9 (1 PUNTO) Implementad, en la clase `Anotador`, el siguiente método:

```
public void enviaLoteDeAnotacionesASalida(int importancia, String idSalida)
throws ImportanciaInvalidaException, NoSalidaException;
```

Este método envía aquellas anotaciones presentes en el contenedor de anotaciones para ser enviadas por lotes y cuya importancia es igual a la indicada en el argumento `importancia`, a la salida cuyo identificador es igual al indicado por el argumento `idSalida` (y las elimina de la cola de anotaciones utilizando el método implementado en el ejercicio anterior). Si el valor del argumento `importancia` no es válido, el método crea y lanza una excepción `ImportanciaInvalidaException`. Si el anotador no está conectado con ninguna salida con ese identificador, el método crea y lanza una excepción `NoSalidaException`.

```
public void enviaLoteDeAnotacionesASalida(int importancia, String idSalida) throws
ImportanciaInvalidaException, NoSalidaException {
    if (Constant.esValida(importancia) == false) throw new ImportanciaInvalidaException();
    Salida s = salidas.get(idSalida);
    if (s == null) throw new NoSalidaException(idSalida);
    List<Anotacion> anotacionesAEliminar = new ArrayList<Anotacion>();
    for(Anotacion a : lotes) {
        if(a.getImportancia() == importancia) {
            s.anota(a);
            anotacionesAEliminar.add(a);
        }
    }
    elimina(anotacionesAEliminar);
}
```

PREGUNTA 10 (1 PUNTO) Implementad el método:

```
public void anota(Anotación a);
```

Para las clases `SalidaConsola` y `SalidaArchivo`. Este método envía una anotación como texto a la salida correspondiente de cada clase.

```
public class SalidaConsola extends Salida {
    @Override
    public void anota(Anotacion a) {
        System.out.println(a);
    }
}
public class SalidaArchivo extends Salida {
    @Override
    public void anota(Anotacion a) {
        try{
            salida.println(a);
        }catch(IOException e){
            System.out.println(ex.getMessage());
        }
    }
}
```

}