Laboratorio de Software - 2025
Práctica n° 1

Temas
- Especificadores de acceso: private, public, protected, default.
- Constructores de Clases.
- Clases abstractas.

(1) Escriba una clase llamada Vacuna con 4 variables de instancia: marca, país de origen, enfermedad que previene y cantidad de dosis. Implemente los getters y setters para cada una de las variables de instancias anteriores.
a) Sobre-escriba el método toString() de Object, para ello declare una variable local de tipo StringBuffer y utilícela para concatenar cada uno de los datos de la vacuna y retorne un objeto String con los datos del mismo.
b) Escriba el método main() en la clase TestVacuna, donde se debe crear un arreglo con 5 objetos Vacuna inicializados, para luego recorrer el arreglo e imprimir en pantalla los objetos guardados en él.

c) Comente el método toString() escrito en la clase Vacuna y vuelva a ejecutar el programa. ¿Cuál es la diferencia entre b) y c)?
Al ejecutar el método toString() por defecto, la pantalla imprime algo similar a Vacuna@452b3a41.
Donde:
- Vacuna: es el nombre de la clase del objeto.
- 452b3a41: es el hashcode del objeto, expresado en hexadecimal.

Este hashcode está relacionado con la dirección en memoria, pero no es la dirección física real. Java no expone directamente las direcciones de memoria por razones de seguridad y portabilidad (especialmente en la JVM, que puede mover objetos en memoria).

d) Cree otro objeto de tipo Vacuna y compárelo con el anterior. ¿Qué método de Object es utilizado para la comparación por contenido?
Para la comparación por contenido se utiliza el método equals() modificándolo.
Por defecto, este método compara referencias (es decir, si dos variables apuntan al mismo objeto en memoria). Pero cuando se sobrescribe en una clase (como String, Integer, o una clase personalizada como Vacuna), se puede hacer que compare el contenido de los objetos.

e) Ejecute la aplicación fuera del entorno de desarrollo. ¿Para que se utiliza la variable de entorno CLASSPATH?
La variable de entorno CLASSPATH contiene las rutas en donde las herramientas de Java y la JVM deben mirar para localizar los archivos de código y los paquetes que se van a utilizar en el programa.

f) Construya un archivo jar con las clases anteriores, ejecútelo desde la línea de comandos. ¿Dónde se especifica en el archivo jar la clase que contiene el método main?
La clase que contiene el main se especifica en el manifest (META-INF/MANIFEST.MF). Tiene una línea que indica cuál es la clase principal.

(2) Analice las siguientes clases y responda cada uno de los incisos que figuran a continuación.

a) Considere la siguiente clase **Alpha**. ¿Es válido el acceso de la clase **Gamma**? Justifique.

```
package  griego;
class Alpha {
     protected int x;
     protected void otroMetodoA(){
         System.out.println("Un método protegido");
     }
}

package  griego;
class Gamma {
   void unMétodoG(){
         Alpha a = new Alpha();
         a.x=10;
         a.otroMetodoA();
     }
 }
```

El acceso de Gamma es válido, un miembro protected es accesible:

- Dentro de la misma clase.
- En las subclases, incluso si están en otro package.
- En todas las clases del mismo package, aunque no sean subclases.

b) Considere la siguiente modificación de la clase **Alpha**. ¿Son válidos los accesos en la clase **Beta**? Justifique.

```
package griego;

public class Alpha {
    public int x;
    public void unMetodoA(){
         System.out.println("Un Método Público");
    }
}


package romano;
import griego.*;

class Beta {
   void unMetodoB(){
         Alpha a=new Alpha();
         a.x=10;
         a.unMetodoA();
   }
}
```

Los accesos de Beta son válidos. Lo que esté declarado public está disponible para todos.
Un miembro public es accesible:

- Desde cualquier clase, esté o no en el mismo paquete.
- Siempre que la clase que lo contiene también sea public y visible.

c) Modifique la clase **Alpha** como se indica debajo. ¿Es válido el método de la clase **Beta**? Justifique.

```
package  griego;
public class Alpha {
    int x;
    void unMetodoA(){
         System.out.println("Un mét. paquete");
    }
}

package romano;
import griego.*;
class Beta {
   void unMetodoB(){
         Alpha a = new Alpha();
         a.x=10;
         a.unMetodoA();
    }
}
```

Los accesos no son válidos. La clase Alpha tiene x y unMetodoA sin especificador de control de acceso, por lo que toman el especificador package.

d) Considere el inciso c) ¿Es válido el acceso a la variable de instancia x y al método de instancia unMetodoA() desde una subclase de Alpha perteneciente al paquete romano? Justifique.
Los accesos no serían válidos. Las variables, métodos y constructores declarados privados del paquete son accesibles sólo desde clases pertenecientes al mismo paquete donde se declaran.

e) Analice el método de la clase Delta. ¿Es válido? Justifique analizando cómo influye el control de acceso protected en la herencia de clases.

```
package  griego;
public class Alpha {
     protected int x;
     protected void otroMetodoA(){
         System.out.println("Un método protegido");
     }
}

package romano;
import griego.*;
public class Delta extends Alpha {
   void unMetodoD(Alpha a, Delta d){
         a.x=10;
         d.x=10;
         a.otroMetodoA();
         d.otroMetodoA();
        }
}
```

El método no es válido.
Dentro del mismo paquete protected se comporta como package, cualquier clase puede acceder.
En otro paquete, las subclases pueden acceder a los miembros protected, pero solo a través de sí mismas o de referencias de su propio tipo. No pueden

acceder a los protected de otra instancia de la superclase que no sea del mismo tipo.

(3) Respecto de los **constructores**, analice los siguientes casos:

3.1.- Escriba 3 subclases de la clase Vacuna(definida en el punto 1) llamadas VacunaPatogenoIntegro, VacunaSubunidadAntigenica y VacunaGenetica con las siguientes variables de instancias:

- VacunaPatogenoIntegro: define una variable de instancia destinada para el nombre del virus patógeno inactivado o atenuado.
- VacunaSubunidadAntigenica: define 2 variables de instancia, una para guardar la cantidad de antígenos de la vacuna y la otra para mantener el tipo de proceso llevado a cabo.
- VacunaGenetica: define dos variables de instancia, una para la temperatura mínima y otra para la temperatura máxima de almacenamiento.

a) Implemente los getters y setters para cada una de las variables de instancias anteriores.
b) Implemente los constructores para las clases anteriores, todos ellos deben recibir los parámetros necesarios para inicializar las variables de instancia propias de la clase donde están definidos.
c) ¿Pudo compilar las clases? ¿Qué problemas surgieron y por qué? ¿Cómo los solucionó?

Si se crean los constructores únicamente con las variables de instancia indicadas al inicio, sin las que hereda de la clase padre, las clases no pueden compilarse. Es necesario el llamado super() con la lista de atributos necesarios para inicializar las variables heredadas.

3.2.- El siguiente código, define una subclase de java.io.File. Verifique si compila. Si no lo hace implemente una solución.

```
package laboratorio;
import java.io.File;
public class MiArchivo extends File {
    public MiArchivo() {
        System.out.println("Mi Archivo instanciado") ;
    }
}
```

Nota: Recuerde que en la url https://docs.oracle.com/en/java/javase/24/docs/api/ tiene disponible al documentación de la API de java.

La clase java.io.File no tiene constructor sin argumentos. Todos sus constructores esperan al menos un parámetro. Por lo tanto, al definir un

constructor vacío en MiArchivo, el compilador intenta invocar implícitamente al constructor por defecto de File, que no existe, y da error de compilación.

3.3.- Las clases definidas a continuación establecen una relación de herencia. La implementación dada, ¿es correcta?.

Constructores privados

```
package laboratorio;
public class SuperClase {
  private SuperClase() {
  }
}

package laboratorio;
public class SubClase extends SuperClase {
      public SubClase() {
  }
}
```

No es correcta. Como el constructor de la superclase está declarado private, no puede ser accedido por la subclase y no puede instanciarse por default.

Constructores protegidos

```
package laboratorio;
public class SuperClase{

 protected SuperClase(){
 }

}

package laboratorio1;

public class SubClase extends SuperClase {
  public SubClase() {
  }
}


package laboratorio1;

public class OtraClase {
  public OtraClase() {
  }
  public void getX() {
      new SuperClase();
  }
}
```

La implementación es incorrecta.

Para que SubClase sea válida, debe importarse a SuperClase.

Por su parte, OtraClase no será válida ni siquiera con la importación porque el constructor de SuperClase sólo brinda acceso a sus hijas y a las clases dentro de su paquete.

(4) Implemente una clase llamada Logger que siga el patrón Singleton. Esta clase debe garantizar que solo haya una instancia de Logger a lo largo de la ejecución de la aplicación. Proporcione un método estático getInstance() para

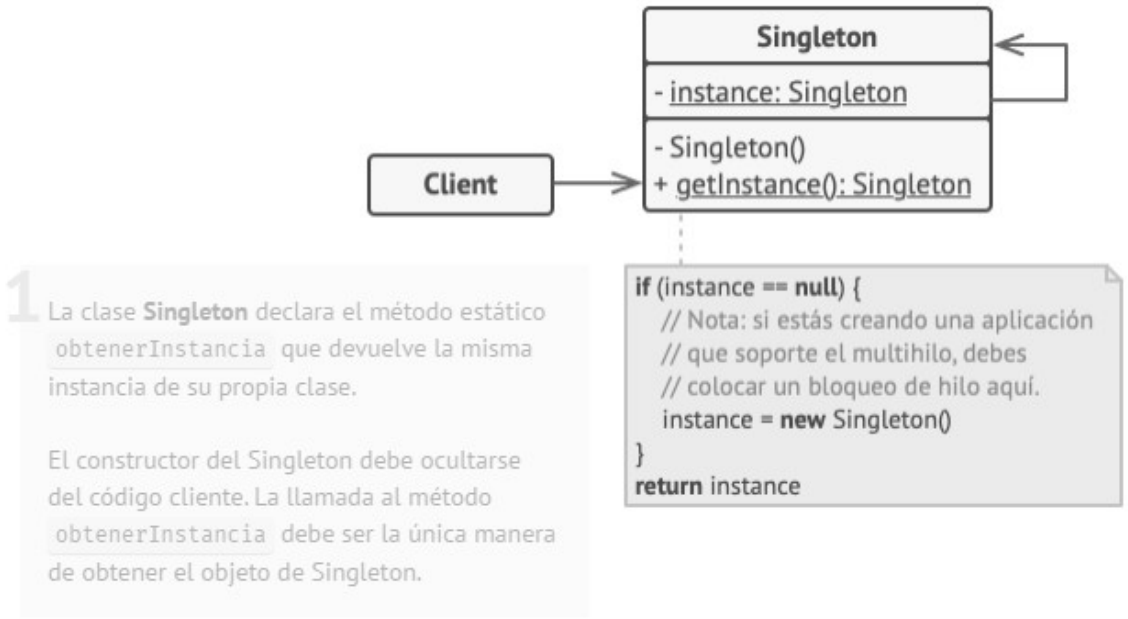
acceder a esta instancia. Además, debe incluir métodos para registrar mensajes de diferentes tipos, como logInfo(String mensaje), logWarning(String mensaje) y logError(String mensaje), que simulen el registro de mensajes mostrándolos en la consola. Piense en los modificadores de acceso del constructor, para buscar una solución.

Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.
El patrón Singleton resuelve dos problemas al mismo tiempo, vulnerando el Principio de responsabilidad única:

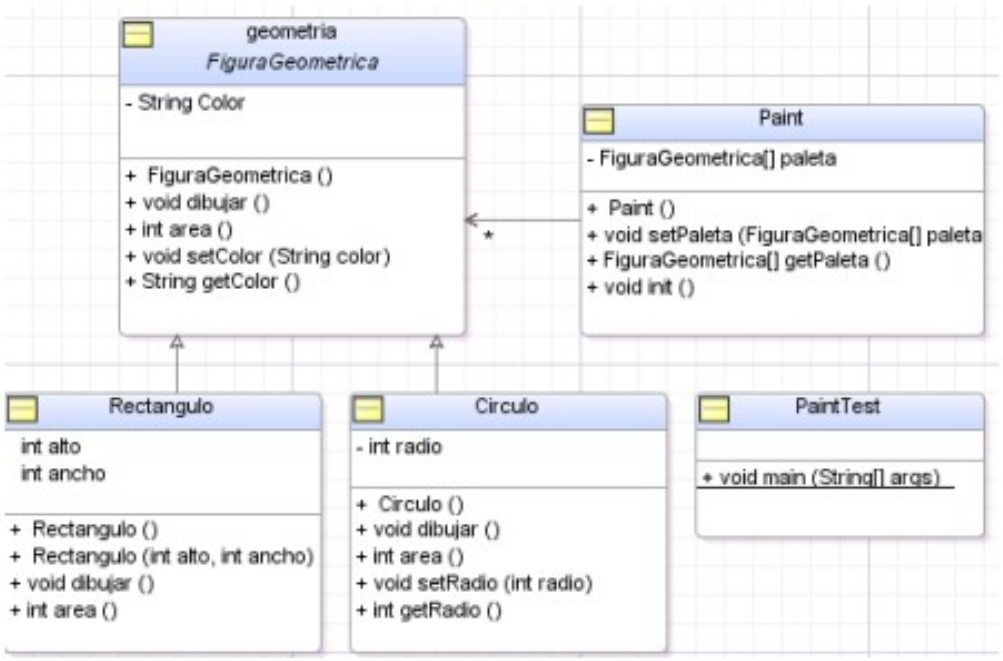
- Garantizar que una clase tenga una única instancia. La finalidad de esto es controlar el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo.
- Proporcionar un punto de acceso global a dicha instancia. Las variables globales son poco seguras, ya que cualquier código podría sobrescribir el contenido de esas variables y descomponer la aplicación.

Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.
- Crear un método de creación estático que actúe como constructor. Este método invoca al constructor privado para crear un objeto y lo guarda en un campo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.



(5) Escriba las siguientes clases java que figuran en el siguiente diagrama UML respetando cada una de las especificaciones para las clases y las relaciones entre ellas:



Tenga en cuenta lo siguiente:

- La clase `FiguraGeometrica` es una clase abstracta con 2 métodos abstractos `dibujar()` y `area()` y el resto de los métodos concretos.
- Las subclases `Rectangulo` y `Circulo` son clases concretas. Ambas deben implementar el método `dibujar()` simplemente imprimiendo un mensaje en la consola. Por ejemplo: "se dibuja un círculo de radio 2 y de color azul" (donde el `radio` y el `color` son variables de instancia). El método `area()` debe implementarse en cada subclase de `FiguraGeometrica`.
- En la clase `Paint`, el método `init()` debe crear las instancias de `Rectangulo` y `Circulo` y guardarlas en el arreglo `paleta`. Los valores para crear estas instancias son los siguientes:
  - Defina 2 objetos `Circulo` (radio 2 y color azul, radio 3 y color amarillo)
  - Defina 2 objetos `Rectangulo` (alto 2, ancho 3, color verde y alto 4 y ancho 10 y color rojo).
- La clase `PaintTest` debe crear una instancia de `Paint`, inicializarla y recorrerla. En cada iteración invoque el método `area()` sobre el elemento actual y `getRadio()`, sólo si se trata de un objeto de tipo `Circulo`.
- Construya un archivo jar ejecutable con las clases anteriores. El mismo debe poderse ejecutar como un programa haciendo doble click