Introducción a la Programación Orientada a Objetos
DCIC – UNS - 2024
Actividades prácticas: Objetos y Clases

Objetivos

- Implementar una clase en Java con atributos de instancia de tipo elemental o tipo String, a partir de una especificación que incluye el diagrama de clases y comentarios o un texto que describe funcionalidades y responsabilidades.
- Verificar los servicios a través de una clase tester con valores leídos por consola o establecidos en el diseño.
- Dibujar el diagrama de objetos de acuerdo a una consigna.

EJERCICIO 1.

- Describa el modelo computacional de la programación orientada a objetos.
- Defina los conceptos de objeto, clase, comando y consulta.
- Explique qué es un constructor, cómo se define y usa en Java y para qué sirve.
- Explique por qué la programación orientada a objetos favorece la productividad del software.
- ¿Qué es la programación basada en eventos?

EJERCICIO 2. Dado el siguiente diagrama:

Poliza
<<*Atributos de instancia*>> nroPoliza: entero incendio: real robo: real activa: boolean
<<*Constructor*>> Poliza(np: entero) Poliza(np: entero, i: real, r: real) <<*Comandos*>> establecerIncendio(m: real) establecerRobo(m: real) actualizarPorcentaje(p: entero) activar() desactivar() <<*Consultas*>> obtenerNroPoliza(): entero obtenerIncendio(): real obtenerRobo(): real obtenerCostoPoliza(): real estaActiva(): boolean
Responsabilidades: Todos los servicios requieren que los parámetros tengan valores positivos.

actualizarPorcentaje(p: entero)
Si la póliza está activa modifica los valores de incendio y robo incrementándolos de acuerdo al porcentaje p.

obtenerCostoPoliza(): real
retorna la suma del monto por incendio y el monto por robo.

a) Implemente en Java la clase *Poliza*. Una póliza siempre está activa cuando se crea.

b) Implemente una clase *TesterPoliza* con un método main() que solicite al usuario los datos de una póliza, controle que los tres valores sean positivos, cree un objeto de la clase *Poliza* usando el constructor con tres parámetros y a continuación, actualice con un porcentaje de 20%, desactive, actualice en 10%, active y muestre por pantalla el número de póliza, el costo y el estado (activa o no).

c) Modifique el método main de la clase *TesterPoliza* agregando instrucciones para crear un objeto de la clase *Poliza* usando el constructor con un parámetro con valor fijo 111, luego establezca valores

1000 y 1200 para robo e incendio, actualícelos en un 15% y a continuación muestre en consola el número de póliza y el costo.

EJERCICIO 3. Una estación de servicio cuenta con surtidores de combustible capaces de proveer Gasoil, Nafta Super y Nafta Premium 2.000. Todos los surtidores tienen capacidad para almacenar un máximo de 20.000 litros de cada combustible. En cada surtidor se mantiene registro de la cantidad de litros disponibles en depósito de cada tipo de combustible (esta cantidad se inicializa en el momento de crearse un surtidor con la cantidad máxima de carga). En cada surtidor es posible llenar el depósito o extraer combustible.

Surtidor
<<*Atributos de clase*>> maximaCarga: entero <<*Atributos de instancia*>> cantGasoil: entero cantSuper: entero cantPremium: entero
<<*Constructor*>> Surtidor() <<Comandos>> llenarDepositoGasoil() llenarDepositoSuper() llenarDepositoPremium() extraerGasoil(litros: entero) extraerSuper(litros: entero) extraerPremium(litros: entero) <<*Consultas*>> obtenerMaximaCarga(): entero obtenerLitrosGasoil(): entero obtenerLitrosSuper(): entero obtenerLitrosPremium(): entero depositosLlenos(): boolean

- *llenarDepostioGasoil(), llenarDepositoSuper(), llenarDepositoPremium()* se completa el depósito de acuerdo a la máxima carga.
- *extraerGasoil(litros: entero), extraerSuper(litros: entero), extraerPremium(litros: entero)*. Si la cantidad de combustible disponible es mayor a litros, decrementa la cantidad en litros, sino le asigna 0. Requiere litros > 0.
- *depositosLlenos(): boolean*. Retorna true si los tres depósitos tienen la máxima carga.

a) Implemente en Java la clase *Surtidor* de acuerdo al diagrama y las funcionalidades descriptas.
b) Implemente en Java la clase *SimulacionSurtidor* que permita verificar los servicios provistos por la clase *Surtidor*, de acuerdo al algoritmo:

```
Algoritmo SimuladorSurtidor
    crea un surtidor
    n = leer cantidad de iteraciones
    repetir n veces
      mostrar la cantidad actual en el depósito de cada combustible
      opción = leer una opción entre 1 y 6
      según opción sea:
        1: leer litros a cargar y cargar Gasoil
        2: leer litros a cargar y cargar Super
        3: leer litros a cargar y cargar Premium
        4: llenar Deposito Gasoil
        5: llenar Deposito Super
        6: llenar Deposito Premium
```

EJERCICIO 4. En un videojuego las criaturas habitan en refugios que disponen de cierta cantidad de alimentos, bebidas y camas. Un refugio es habitable si tiene alimentos o bebidas o camas disponibles. Los atributos se inicializan al crearse el refugio, con los valores de los tres parámetros. El atributo camas indica cuántas camas están ocupadas, alimentos indica cuántos alimentos hay en la alacena y bebidas cuántas bebidas hay en la alacena.

- *Refugio(a, b, c: entero)*. Si la suma de los valores de los parámetros a y b es mayor a la capacidad de la alacena, asigna la mitad de esta capacidad a alimentos y la mitad a bebidas, sino asigna el valor de a al atributo alimentos, y el valor de b al atributo bebidas. Si el parámetro c es mayor a la cantidad de camas, asigna esta constante al atributo camas, sino le asigna c.
- *consumirAlimento()* y *consumirBebida()*. Decrementan el valor de los atributos en 1. Requieren que la clase cliente haya controlado que hay alimento o bebida, según corresponda.
- *reponerAlimentos(n: entero): boolean* y *reponerBebidas(n: entero): boolean.* Incrementan los valores de los atributos alimentos o bebidas, siempre y cuando n sea positivo y la capacidad de la alacena lo permita; en ese caso el comando retorna true. Si n no es positivo o al sumar n a la cantidad de alimentos y bebidas el valor supera a la capacidad de la alacena, el atributo no se modifica y el comando retorna false.
- *desocuparCama(): boolean*. Si hay camas ocupadas decrementa en 1 camas y retorna true, si no retorna false.
- *ocuparCama(): boolean*. Si hay camas disponibles incrementa camas en 1 y retorna true, si no retorna false.
- *disponibilidad(): entero*. Retorna la cantidad de camas disponibles.
- *diasSupervivencia(): entero*. Retorna el menor valor entre alimentos y bebidas.
- *mayorAlimentos(r: Refugio): boolean*. Retorna true si r está ligado y el refugio que recibe el mensaje tiene más alimentos que el refugio r, en caso contrario retorna false.
- *equals(r: Refugio): boolean*. Retorna true si r está ligado y tiene el mismo estado interno que el refugio que recibe el mensaje.

Implemente la clase *Refugio* en Java y verifique con la clase *TesterRefugio* publicada. Luego complete el tester de manera de probar los métodos que faltan.

Refugio
<<*Atributos de clase*>> capacidadAlacena = 20, cantidadCamas = 10 <<*Atributos de instancia*>> alimentos, bebidas, camas: entero
<<Constructor>> Refugio(a, b, c: entero) <<*Comandos*>> consumirAlimento() consumirBebida() ocuparCama(): boolean desocuparCama(): boolean reponerAlimentos(n: entero): boolean reponerBebidas(n: entero): boolean <<*Consultas*>> obtenerAlimentos(): entero obtenerBebidas(): entero obtenerCamas(): entero obtenerCapacidadAlacena(): entero esHabitable(): boolean disponibilidad(): entero diasSupervivencia(): entero mayorAlimentos(r: Refugio): boolean equals(r: Refugio): boolean clone(): Refugio toString(): String

EJERCICIO 5. Dada la siguiente especificación:

Sensor	
<<*Atributos de clase*>> max = 0,01 <<*Atributos de instancia*>> p1: real p2: real	
<<*Constructor*>> Sensor(p1, p2: real) <<*Comandos*>> establacerP1(p: real) establacerP2(p: real) copy(s: Sensor) <<*Consultas*>> obtenerP1(): real obtenerP2(): real riesgo(): boolean emergencia(): boolean equals(s: Sensor): boolean clone(): Sensor	copy(s: Sensor) si el parámetro s está ligado modifica el estado interno del sensor que recibe el mensaje; sino no provoca ningún cambio. riesgo() retorna true si p2 es mayor que p1 emergencia() retorna true si p1 es menor a max.

a) Implemente la clase *Sensor* modelada en el diagrama de clase. Tenga en cuenta que si no se especifica quién es responsable de controlar que el parámetro esté ligado, es el método *equals* quien debe hacerlo y retornar false si no lo está. Observe también que, en el constructor, coinciden los nombres de los parámetros con los de los atributos de instancia. ¿Cómo se procede en casos como éstos?

b) Elabore un diagrama de objetos al finalizar el siguiente segmento de código:

```
Sensor s1, s2, s3;
s1 = new Sensor(1.5, 1.6);
s2 = s1.clone();
s3 = new Sensor(1.5, 1.5);
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

c) Dibuje un nuevo diagrama de objetos luego de continuar la ejecución con el siguiente segmento de código:

```
s3.copy(s1);
s1 = s2;
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

EJERCICIO 6. Analice si las siguientes afirmaciones son correctas:

a. Un constructor es un método que se invoca cuando se crea un objeto.
b. Un comando es un método que no retorna un resultado.
c. Una consulta es un método que no modifica el estado interno del objeto.
d. En Java, el pasaje de parámetros es por valor.
e. Desde el punto de vista del diseño de un sistema orientado a objetos, una clase es un patrón que establece los atributos y el comportamiento de un objeto.
f. En la implementación de un sistema orientado a objetos, una clase es un módulo de software que puede desarrollarse con cierta independencia del resto de los módulos.
g. Si las variables a y b se declaran de clase Refugio y a == b, entonces a.equals(b) es true.
h. Si las variables a y b se declaran de clase Refugio, a y b están ligadas y a == b, entonces a.equals(b) es true.

EJERCICIO 7. Dada la siguiente implementación para la clase *CuentaBancaria*

```
class CuentaBancaria{
//Atributos de clase, monto máximo para extraer en descubierto
  private static final int maxDescubierto = 1000;
//Atributos de instancia
  private int codigo;
  private float saldo;
// Constructores
//El código se establece al crear la cuenta y no cambia
  public CuentaBancaria(int cod){
       codigo = cod; saldo = 0;    }
  public CuentaBancaria(int cod, float sal){
       codigo = cod; saldo = sal;  }
```

```
// Comandos
  public void depositar(float mto){
//Requiere mto > 0
           saldo = saldo + mto;        }
  public void extraer(float mto){
//Requiere mto > 0
      if (puedeExtraer(mto))
        saldo = saldo - mto;  }
// Consultas
  public int obtenerCodigo(){
        return codigo;  }
  public float obtenerSaldo(){
        return saldo;}
  public String toString(){
        return codigo + " " + saldo;}
  public boolean puedeExtraer(float mto){
//Requiere mto > 0
           boolean puede = false;
        if ((-1)*(saldo-mto)<=maxDescubierto)
           puede = true;
        return puede;      }
}
```

a. Enumere las variables definidas por el programador que forman parte del ambiente de referenciamiento en el bloque condicional del método puedeExtraer.
b. Muestre el alcance de cada variable declarada en la clase.

EJERCICIO 8. Implemente la clase *Fecha* de acuerdo a la siguiente especificación e implemente un *Tester* para verificarla:

Fecha
<<*Atributos de instancia*>> dia, mes, anio: entero
<<*Constructor*>> Fecha(d, m, a: entero) <<*Comandos*>> establecerDia(d: entero) establecerMes(m: entero) establecerAnio(a: entero) <<*Consultas*>> obtenerDia(): entero obtenerMes(): entero obtenerAnio(): entero esBisiesto(): boolean esAnterior(f: Fecha): boolean mismoAnio(f: Fecha):boolean equals(f: Fecha): boolean toString(): String

La clase cliente es responsable de garantizar que los valores de los parámetros del constructor sean válidos.

- *esAnterior(f: Fecha): boolean*. Retorna verdadero si y solo si la fecha que recibe el mensaje es anterior a la fecha pasada por parámetro. Requiere f ligada.

- *equals(f: Fecha): boolean*. Requiere f ligada.

EJERCICIO 9. Dado el siguiente diagrama de clase:

Color
<<*Atributos de instancia*>> rojo: entero verde: entero azul: entero
<<*Constructores*>> Color() Color(r, a, v: entero) <<*Comandos*>> variar(val: entero) variarRojo(val :entero) variarAzul(val: entero) variarVerde(val: entero) establecerRojo(val: entero) establecerAzul(val: entero) establecerVerde(val: entero) copy(c: Color) <<*Consultas*>> obtenerRojo(): entero obtenerAzul(): entero obtenerVerde(): entero esRojo(): boolean esGris(): boolean esNegro(): boolean complemento(): Color equals(c:Color): boolean clone(): Color toString(): String

Cuando observamos una imagen en la computadora no percibimos cada color particular, pero si aumentamos la imagen, por ejemplo, un 600%, cada color se vuelve distinguible. La percepción humana de la luz también es muy limitada y depende de nuestros sensores del color. Nuestro cerebro determina qué color "ve" en base a sensar tres colores: azul, verde y rojo. De este modo, cada color puede codificarse mediante una terna de números: el primero representa la cantidad de color rojo, el segundo la cantidad de color verde y el tercero la cantidad de color azul. El rango para cada valor es 0...255. Combinando el máximo de los tres se obtiene el blanco (255, 255, 255). La ausencia de los tres produce el negro (0, 0, 0). El rojo es claramente (255, 0, 0). Cuando se mantiene el mismo valor para las tres componentes se obtiene gris. La terna (50, 50, 50) representa un gris oscuro. En cambio, (150, 150, 150) es un gris más claro. Esta representación se llama modelo RGB.

En la clase *Color*:

- *Color()*. Inicializa con la representación del blanco.
- *Color(r, a, v: entero)*. Si uno de los tres parámetros está fuera de rango, inicializa con la representación del blanco.
- *variar(val: entero).* Modifica cada componente de color sumándole si es posible, un valor dado. Si sumándole el valor dado a una o varias componentes se supera el valor 255, dicha componente

queda en 255. Si el argumento es negativo la operación es la misma, pero en ese caso el mínimo valor que puede tomar una componente es 0.

- *variarRojo(val: entero).* Modifica la componente de rojo sumándole un valor dado. Ídem para azul *variarAzul(val: entero)* y verde *variarVerde(val: entero)*.
- *esRojo(): boolean*. Retorna el valor verdadero si el objeto que recibe el mensaje representa el color rojo. Ídem para *grisesGris():boolean* y para negro *esNegro():boolean*
- *complemento():Color*. Retorna un nuevo objeto con el color complemento del color del objeto que recibe el mensaje para alcanzar el color blanco.

Atención: los métodos *equals* y *copy* deben controlar que el parámetro esté ligado ya que la especificación no indica que el control sea responsabilidad de la clase cliente. Observe que al no especificarse la funcionalidad para el caso de que el parámetro no esté ligado, puede implementarlos de modo que el comando *copy* no tendrá ningún efecto si c no está ligado, y que *equals* retorne false si c no está ligado.

a) A partir de la especificación anterior implemente y verifique la clase *Color* encapsulando atributos y comportamiento para modelar la situación planteada. Incluya todos los métodos auxiliares que considere oportunos.

b) Elabore un diagrama de objetos al finalizar el siguiente segmento de código:

```
Color s1, s2, s3, s4;
boolean b1, b2;
s1 = new Color(100, 90, 120);
s2 = new Color(55, 110, 100);
s3 = new Color(110, 110, 100);
s4 = s1;
```

Elabore un nuevo diagrama de objetos luego de continuar la ejecución con el siguiente segmento de código:

```
s1 = s3;
s2 = s1;
s1 = new Color(80, 80, 80);
```

Dibuje tres nuevos diagramas partiendo del anterior y ejecutando cada uno de los siguientes segmentos de código:

```
s2 = s1.clone();          s3.copy(s1);            s4 = s1;
b1 = s1 == s2;            b1 = s1 == s3;          b1 = s1 == s4;
b2 = s1.equals(s2);       b2 = s1.equals(s3);     b2 = s1.equals(s2);
```

d) Analice qué ocurrirá al ejecutarse el siguiente segmento de código:

```
s1 = null;
s2 = s1.clone();
```

EJERCICIO 10. Implemente y verifique la clase *Jugador* de acuerdo a la siguiente especificación:

Jugador
<<*Atributos de instancia*>> nombre: String nroCamiseta: entero posicion: entero golesConvertidos: entero partidosJugados: entero
<<*Constructor*>> Jugador(nom: String) <<*Comandos*>> establecerNroCamiseta(n: entero) establecerPosicion(n: entero) estableceGolesConvertidos(n: entero) establecerPartidosJugados(n: entero) aumentarGoles(n: entero) aumentarUnPartido() <<*Consultas*>> obtenerNombre(): String obtenerNroCamiseta(): entero obtenerPosicion(): entero obtenerGolesConvertidos(): entero obtenerPartidosJugados(): entero promedioGolesXPart(): entero masGoles(j:Jugador): boolean jugConMasGoles(j: Jugador): Jugador toString(): String clone(): Jugador equals(j: Jugador): boolean

- *masGoles(j: Jugador): boolean*. Devuelve true si el jugador que recibe el mensaje tiene más goles que el jugador que pasa como parámetro, en caso contrario retorna false.
- *jugConMasGoles(j: Jugador): Jugador*. Retorna la referencia al jugador que recibió el mensaje, si hizo más goles que el jugador j, sino retorna j. Requiere j ligado.
- *equals(j: Jugador): boolean*. Requiere j ligado.

Atención: En la implementación del método clone tenga en cuenta que no dispone de un constructor que inicialice todos los atributos de instancia de acuerdo a los valores de los parámetros.