Programación

Tema 4

Programación Orientada a Objetos

Tema 4
POO - Avanzado

1.- Clases Abstractas
2.- Interface
3.- final
4.- static
5.- Clases embebidas

1.- Clases Abstractas

- Hay ocasiones, cuando se desarrolla una jerarquía de clases en que algún comportamiento está presente en todas ellas, pero se materializa de forma distinta para cada una.
  - Ejemplo : Estructuras de clases para manipular figuras geométricas
  - Tener una clase genérica **Figura** y una serie de clases derivadas circulo, cuadrado, triangulo etc…
  - Cada una de las clases derivadas con un método **dibujar**, que conlleva operaciones concretas dependiendo de cada figura.
  - En la clase figura no tiene sentido tener el método dibujar
  - Por tanto figura representa una **abstracción** de las posibles figuras
  - Tenemos que **imponer** que las clases derivadas implementen el método **dibujar**

1.- Clases Abstractas

- Para resolver esta problemática Java proporciona las clases y métodos abstractos.
- Un **método abstracto** es un método declarado en una clase para el cual esa clase no proporciona la implementación (el código).
- Una **clase abstracta** es una clase que tiene al menos un método abstracto.
- Una clase que extiende a una clase abstracta puede :
  - Implementar los métodos abstractos (escribir el código)
  - Volverlos a declarar como abstractos, con lo que ella misma se convierte también en clase abstracta.

1.- Clases Abstractas

Declaración de una clase abstracta

- La clase abstracta se declara simplemente con el modificador **abstract** en su declaración.
- Los métodos abstractos se declaran también con el mismo modificador, declarando el método pero sin implementarlo
  - sin el bloque de código encerrado entre {}

```
Public abstract class FiguraGeometrica {
   . . .
   abstract void dibujar();
   . . .
}

Public class Circulo extends FiguraGeometrica {
   . . .
   void dibujar() {
      // codigo para dibujar Circulo
      . . .
   }
}
```

1.- Clases Abstractas

- La clase derivada se declara e implementa de forma normal, como cualquier otra.
- Sin embargo si no declara e implementa los métodos abstractos de la clase base (en el ejemplo el método dibujar) el compilador genera un error indicando que no se han implementado todos los métodos abstractos. Opciones
  - Se implementan los métodos abstractos
  - Se declara la clase abstracta

```
Public abstract class FiguraGeometrica {
   . . .
   abstract void dibujar();
   . . .
}

Public class Circulo extends FiguraGeometrica {
   . . .
   void dibujar() {
      // codigo para dibujar Circulo
      . . .
   }
}
```

1.- Clases abstractas

Referencias y objetos abstracto

- Se pueden crear referencias a clases abstractas como cualquier otra. No hay ningún problema en poner
  - FiguraGeometrica figura;
- Una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de una clase abstracta.
  - El compilador producirá un error si se intenta:

    FiguraGeometrica figura = new FiguraGeometrica();

- Esto es coherente dado que una clase abstracta no tiene completa su implementación y encaja bien con la idea de que algo abstracto no puede materializarse.

1.- Clases abstractas

Referencias y objetos abstracto

- Sin embargo, utilizando el up-casting visto en el capítulo dedicado a la Herencia si se puede escribir:

```
FiguraGeometrica figura = new Circulo(. . .);
figura.dibujar();
```

- La invocación al método dibujarse resolverá en tiempo de ejecución y la JVM llamará al método de la clase adecuada. En nuestro ejemplo se llamará al método dibujarde la clase Circulo.

2.- Interface

- El concepto de Interface lleva un paso más adelante la idea de las clases abstractas.
- En Java un interface es una **clase abstracta pura**.
  - Una clase donde **todos** los métodos son **abstractos**
  - No se implementa ninguno.
  - Permite al diseñador de clases establecer la forma de una clase
    - nombres de métodos
    - listas de argumentos
    - tipos de retorno
    - Pero no bloques de código.
- Un interface puede también contener datos miembro, pero estos son siempre static y final.
- Una interface sirve para establecer un '**protocolo**' entre clases.

2.- Interface

- Para crear una interface, se utiliza la palabra clave **interface** en lugar de class.
- La interface puede definirse public o sin modificador de acceso, y tiene el mismo significado que para las clases.
- **Todos** los **métodos** que declara una interface son siempre **public**.
- Para indicar que una clase implementa los métodos de una interface se utiliza la palabra clave **implements**.
- El compilador se encargará de verificar que la clase efectivamente declare e implemente todos los métodos de la interface.
- Una clase puede implementar **más** de una interface.

2.- Interface

- Declaración y uso

```
interface nombre_interface {
    tipo_retorno nombre_metodo ( lista_argumentos ) ;
    . . .
}
Por ejemplo:
interface InstrumentoMusical {
    void tocar();
    void afinar();
    String tipoInstrumento();
}
```

2.- Interface

- Declaración y uso

```
class InstrumentoViento extends Object implements InstrumentoMusical
{
   void tocar() { . . . };
   void afinar() { . . .};
   String tipoInstrumento() {}
}
class Guitarra extends InstrumentoViento {
   String tipoInstrumento() {
      return "Guitarra";
   }
}
```

2.- Interface

Referencia a Interfaces

- Es posible crear referencias a interfaces, pero las interfaces no pueden ser instanciadas.
- Una referencia a una interface puede ser asignada a cualquier objeto que implemente la interface

```
I_InstrumentoMusical instrumento = new Guitarra();
instrumento.play();
System.out.prinln(instrumento.tipoInstrumento());

I_InstrumentoMusical i2 = new I_InstrumentoMusical(); //error.No se
puede instanciar
```

2.- Interfaces

Extensión de interfaces

- Las interfaces pueden extender otras interfaces y, a diferencia de las clases, una interface puede extender más de una interface. La sintaxis es:

```
interface nombre_interface_extends nombre_interface1, nombre_interface2,  {
   tipo_retorno nombre_metodo ( lista_argumentos ) ;
   . . .
}
```

2.- Interfaces

Agrupaciones de constantes

- Todos los datos miembros que se definen en una interface son **static** y **final.**
- **D**ado que las interfaces no pueden instanciarse resultan una buena herramienta para implantar grupos de constantes

```
public interface Meses {
   int ENERO = 1 , FEBRERO = 2 . . . ;
   String [] NOMBRES_MESES = { " " , "Enero" , "Febrero" , . . . };
}
```

Esto puede usarse simplemente:

```
System.out.println(Meses.NOMBRES_MESES[ENERO]);
```

3.- Palabra reservada final

- En ocasiones es conveniente que un método no sea redefinido en una clase derivada o incluso que una clase completa no pueda ser extendida.
- Para esto está la cláusula **final**, que tiene significados levemente distintos según se aplique a un dato miembro, a un método o a una clase.
- **Para una clase**, final significa que la clase no puede extenderse. Es, por tanto el punto final de la cadena de clases derivadas.
  - Por ejemplo si se quisiera impedir la extensión de la clase Ejecutivo, se pondría:

```
final class Ejecutivo {
. . .
}
```

3.- Palabra reservada final

- Para un **método**, final significa que no puede redefinirse en una clase derivada. Por ejemplo si declaramos:

```
class Empleado {
   . . .
   public final void aumentarSueldo(int porcentaje) {
      . . .
   }
   . . .
}
```

- La clase Ejecutivo, clase derivada de Empleado no podría reescribir el método aumentarSueldo, y por tanto cambiar su comportamiento

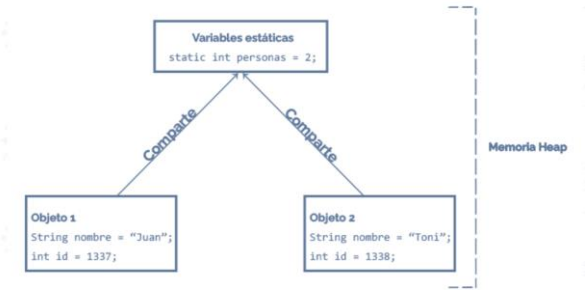
3.- Palabra reservada final

- Para un **dato** miembro, final significa también que **no** puede ser **redefinido** en una clase derivada, como para los métodos, pero además significa que **su valor no puede ser cambiado** en ningún sitio; es decir el modificador final sirve también para definir valores constantes

```
class Circulo {
   . . .
   public final static float PI = 3.141592;
   . . .
}
```

4.- Palabra reservada static

- La palabra clave **static** se usa en atributos, métodos , clases embebidas y bloques
- Utilizamos **static** cuando no queremos crear nuevas instancias cada vez
- Lo usamos cuando queremos una copia única compartida dentro de la clase.
- Los atributos static se almacenan en la memoria dinámica (heap) que es permanente



4.- Palabra reservada static

Atributos static

- Cuando declaramos un campo como **static**, se crea exactamente una copia de ese campo y se comparte entre todas las instancias de esa clase.
- No importa cuántas veces instanciemos una clase. Siempre habrá una sola copia del campo **static** perteneciente a ella.
- El valor de este campo **static** se comparte entre todos los objetos de la misma clase.
- Desde la perspectiva de la memoria, las variables estáticas se almacenan en la memoria **heap**.

4.- Palabra reservada static

```java
public class Coche {
    private String nombre;
    private String motor;

    public static int cantidadDeCoches;

    public Coche(String nombre, String motor) {
        this.nombre = nombre;
        this.motor = motor;
        cantidadDeCoches++;
    }

    // getters y setters
}
```

```java
@Test
public void cuandoSeInicializanObjetosDeCoche_LaCuentaEstaticaAumenta() {
    new Coche("Seat", "Panda");
    new Coche("Porsche", "Taycan");

    assertEquals(2, Coche.cantidadDeCoches);
}
```

4.- Palabra reservada static

Métodos static

- Los métodos static también pertenecen a una clase en lugar de a un objeto.
- Por lo tanto, podemos llamarlos sin crear un objeto de la clase en la que residen.
- Usamos métodos static para realizar una operación que no depende de la creación de una instancia.
- También comúnmente usamos métodos static para crear clases de utilidad o ayudantes para que podamos obtenerlos sin crear un nuevo objeto de esas clases.

4.- Palabra reservada static

```
static void establecerCantidadDeCoches(int cantidadDeCoches) {
    Coche.cantidadDeCoches = cantidadDeCoches;
}
```

```
public String getNombre() {
    return nombre;
}

public String getMotor() {
    return motor;
}

public static String getInformacionDeCoches(Coche coche) {
    return coche.getNombre() + "-" + coche.getMotor();
}
```

4.- Palabra reservada static

Bloques static

4.- Palabra reservada static

- Usamos un bloque static para inicializar variables static.
- Aunque podemos inicializar variables static directamente durante la declaración, hay situaciones en las que necesitamos realizar un procesamiento multilineal.
- En tales casos, los bloques static son útiles

```
public class DemoBloqueEstatico {
    public static List<String> rangos = new LinkedList<>();

    static {
        rangos.add("Oro");
        rangos.add("Platino");
        rangos.add("Diamante");
    }

    static {
        rangos.add("Master");
        rangos.add("Grand Master");
    }
}
```

5.- Clases Embebidas

- Una **clase embebida** es una clase que se define dentro de otra.
- Es una característica de Java que permite agrupar clases lógicamente relacionadas y controlar la 'visibilidad' de una clase.
- El uso de las clases embebidas no es obvio y contienen detallas algo más complejos

```
class Externa {
   . . .
   class Interna {
      . . .
   }
}
```

5.- Clases Embebidas

- La clase Externa puede instanciar y usar la clase Interna como cualquier otra, sin limitación ni cambio en la sintaxis de acceso:

```
class Externa {
   . . .
   class Interna {
      . . .
   }
   void metodo() {
      Interna i = new Interna(. . .);
      . . .
   }
}
```

5.- Clases Embebidas

- Una diferencia importante es que un objeto de la clase embebida está relacionado siempre con un objeto de la clase que la envuelve.
- Las instancias de la clase embebida deben ser creadas por una instancia de la clase que la envuelve.
- Desde el exterior estas referencias pueden manejarse, pero **calificandolas** completamente, es decir nombrando la clase externa y luego la interna.
- Además una instancia de la clase embebida tiene acceso a todos los datos miembros de la clase que la envuelve sin usar ningún calificador de acceso especial, como si le pertenecieran.

5.- Clases Embebidas

```
public class Pizza {

    private static String cantidadCocinada;
    private boolean esMasaDelgada;

    public static class ContadorVentasDePizza {

        private static String cantidadOrdenada;
        public static String cantidadEntregada;

        ContadorVentasDePizza() {
            System.out.println("Campo estático de la clase contenedora es "
              + Pizza.cantidadCocinada);
            System.out.println("Campo no estático de la clase contenedora e
              + new Pizza().esMasaDelgada);
        }
    }

    Pizza() {
        System.out.println("Campo no estático público de la clase estática
          + ContadorVentasDePizza.cantidadEntregada);
        System.out.println("Campo no estático privado de la clase estática
          + ContadorVentasDePizza.cantidadOrdenada);
    }

    public static void main(String[] a) {
        new Pizza.ContadorVentasDePizza();
    }
}
```

5.- Clases Embebidas

Razones para usar clases anidadas static

- Agrupar clases que solo se usarán en un lugar aumenta la encapsulación.
- Acercamos el código al único lugar que lo utilizará. Esto aumenta la legibilidad y facilita el mantenimiento.
- Si una clase anidada no requiere acceso a los miembros de instancia de su clase contenedora, es mejor declararla como static. De esta manera, no estará acoplada a la clase externa y, por lo tanto, será más óptima, ya que no requerirá memoria heap o memoria stack.