LO MÁS IMPORTANTE DE ESTE BOLETÍN: clases internas anónimas y su relación con la lambda.

CLASES ANIDADAS

NO ES UNA CARACTERÍSTICA DE PROGRAMACIÓN FUNCIONAL, pero lo vemos en este bloque porque tienen relación con las expresiones lambda.

Dentro de las llaves de una clase, podemos definir como miembro un atributo, un método y también … ¡otra clase!. Esa clase se llama clase anidada (Nested class)

```
class OuterClass {
   ...
   class NestedClass {
      ...
   }
}
```

Se pueden distinguir varios tipos de **clases anidadas**:

- **Clases anidada estáticas** , declaradas con el modificador **static**.
- **Clases anidadas no static:**
    - **internas miembro**, conocidas habitualmente como **clases internas**. Declaradas al máximo nivel de la clase contenedora y no estáticas.
    - **Clases internas locales**, que se declaran en el interior de un bloque de código (normalmente dentro de un método).
    - **Clases anónimas**, similares a las internas locales, pero sin nombre (sólo existirá un objeto de ellas y, al no tener nombre, no tendrán constructores). Se suelen usar en la **gestión de eventos** en los **interfaces gráficos**.

Es por tanto un tema más amplio de lo que parece. Nos centramos en los tipos en rojo.

¿Porqué usar clases anidadas?

En el tutorial de oracle da 3 razones por las que son útiles, muy resumidamente podemos concluir que en algunos contextos es una forma conveniente de estructurar lógicamente el código.

Clase anidada interna miembro "Clase interna".

```
class Contenedora {
   ...
   class Interna{
      ...
   }
}
```

Algunas características de la clase "interna" :

- Clase interna definida como miembro (no static) de otra clase contenedora
- No puede tener miembros estáticos
- No pueden tener nombres comunes con la clase contenedora
- No es posible crear un objeto de la clase interna sin tener un objeto de la clase contenedora de forma que cada instancia de una clase miembro se asocia internamente a una instancia de la clase contenedora
- Pueden declararse como public, default, privadas o protegidas.
- Esto es importante y aplicable a otros tipos de anidadas=>Entre Contenedora y contenida no hay restricciones de acceso:
    - Aunque una clase interna sea privada la clase contenedora tiene libre acceso a ella.
    - Aunque la clase contenedora tenga miembros miembros private, la clase interna tiene libre acceso a ellos

Veamos algunos ejemplos

Una clase interna es un miembro más de la clase contenedora

Ejemplo: Para observar con claridad este tipo de código las indentaciones son de la mayor importancia.

```
class ClaseContenedora {
    public int intCC=1;
    public class ClaseInterna {
        public int intCI=2;
    }
}
public class App{
    public static void main(String[] args) {
        ClaseContenedora cc = new ClaseContenedora();
        //Los objetos de una clase interna se suelen manejar desde dentro de la contenedora
        //No Obstante, en este caso, poco habitual, lo vamos a hacer desde fuera de la contenedora, concretamente desde App
        ClaseContenedora.ClaseInterna ci = cc.new ClaseInterna();
        System.out.println("El entero de la clase interna vale: "+ ci.intCI);
        //ClaseContenedora.ClaseInterna ci2 = new ClaseInterna();
    }
}
```

Observa y prueba:

- ClaseContenedora sólo acepta el modificador public (o ninguno)
- ClaseInterna puede escribirse con modificador public/protected/private o ninguno. En el ejemplo es public, esto es "raro". Las clases internas no suelen ser accesibles desde el exterior.
- Creamos una instancia de ClaseInterna desde App. Esto también es raro, como veremos después, lo habitual es que sea sólo ClaseContenedora quien cree instancias de ClaseInterna

- que accedemos a la clase interna *"más o menos"* como a un atributo cualquiera con el operador punto.
- La última instrucción comentada daría un error ya que intentamos crear un objeto de clase interna sin crear su correspondiente objeto de clase contenedora. Una instancia de una clase interna siempre tiene que estar conectada a una instancia de su clase contenedora.

Esto es la base, a partir de aquí surgen un buen número de singularidades del uso de clases internas que se salen ahora de nuestros objetivos. Hay muchas novedades sintácticas que no vamos a usar, así que nos limitamos a los aspectos más comunes.

Las clases internas suelen ser private

Igual que un atributo o un método las clases internas también pueden tener acceso private y protected y pueden ser static

Lo más habitual es declarar una clase interna como private de forma que ya no estaría accesible por clases externas a la relación Contenedora/Interna como por ejemplo a la clase App en el ejemplo anterior. Observa los errores de compilación al declararla como private.

```
class ClaseContenedora {
    public int intCC=1;
    private class ClaseInterna {
        public int intCI=2;
    }
}
public class App{
    public static void main(String[] args) {
        ClaseContenedora cc = new ClaseContenedora();
        //error porque ClaseInterna es private
        ClaseContenedora.ClaseInterna ci = cc.new ClaseInterna();
        System.out.println("El entero de la clase interna vale: "+ ci.intCI);

    }
}
```

Lo habitual es que la creación de objetos de la clase interna y su manipulación se haga desde la clase contenedora, no desde otras clases externas

Esto es lógico ya que si hacemos una clase interna private, por descontado su manejo se hará desde la clase contenedora que tiene libre acceso a ella.

```
class ClaseContenedora {
    public int intCC = 1;
    private class ClaseInterna {
        public int intCI = 2;
    }
    public String unMetodoDeClaseContenedora() {
        //esto es lo habitual, objetos de la interna se crean y manipulan dentro del código de la contenedora
        ClaseInterna ci = new ClaseInterna();
        String s = "el entero clase Interna es: " + ci.intCI;
        return s;
    }
}

public class App {
    public static void main(String[] args) {
        ClaseContenedora cc = new ClaseContenedora();
        System.out.println(cc.unMetodoDeClaseContenedora());
    }
}
```

Entre Contenedora e Interna no hay restricciones de acceso.

Observa en este ejemplo, cómo la contenedora puede usar cosas private de la interna y la interna cosas private de la contenedora. Los private en este ejemplo sólo afectarían a clases externas a esta relación de anidamiento como App.

```
class ClaseContenedora {
    private int intCC=1;
    private class ClaseInterna {
        private int intCI=2;
        private void unMetodoDeClaseInterna(){
            //interna puede acceder a private de contenedora
            intCC=intCC*1000;
        }
    }
    public  String unMetodoDeClaseContenedora(){
        //el constructor y el método de interna son private pero Contenedora tiene acceso libre
        ClaseInterna ci= new ClaseInterna();
        ci.unMetodoDeClaseInterna();
        String s="el entero de la clase Interna es "+ ci.intCI+" y el de la contenedora "+intCC;
        return s;
    }
}
public class App{
    public static void main(String[] args) {
    ClaseContenedora cc = new ClaseContenedora();
    System.out.println(cc.unMetodoDeClaseContenedora());
    }
}
```

Ejercicio U8_B3_E1: Recuerda la siguiente implementación de pila

```
 interface Pila{

    //inserta un elemento en la cabeza de la pila
    void push(int dato);

    //saca un elemento de la cabeza de la pila.
    int pop();
    public boolean esVacia() ;
}
class Nodo {
    private Nodo sig;
    private int dato;

    public Nodo(int dato) {
        this.dato = dato;
        this.sig = null;
    }

    public Nodo(int dato, Nodo sig) {
        this.dato = dato;
        this.sig = sig;
    }

    public void setSiguiente(Nodo sig) {
        this.sig = sig;
    }

    public Nodo getSiguiente() {
        return sig;
    }

    public int getDato() {
        return dato;
    }
}

class MiPila implements Pila{

    private Nodo cabeza = null;
    public void push(int dato) {
        if (cabeza == null) {
            cabeza = new Nodo(dato);
```

```
        } else {
            Nodo temp = new Nodo(dato, cabeza);
            cabeza = temp;
        }

    }

    public int pop() {
        int dato = cabeza.getDato();
        cabeza = cabeza.getSiguiente();
        return dato;
    }

    public boolean esVacia() {
        return cabeza == null;
    }

}

class App {
    public static void main(String[] args) {
        MiPila mipila = new MiPila();
        mipila.push(1);
        mipila.push(2);
        mipila.push(3);
        mipila.push(4);
        mipila.push(5);
        while (!mipila.esVacia()) {
            System.out.println(mipila.pop());
        }
    }
}
```

Queremos ocultar la clase Nodo a App que no le interesa en absoluto. Realmente es un detalle técnico de la clase Pila.

Se pide: convierte la clase Nodo en un miembro interno de la clase Pila. Podemos hacer además esa clase private y recordando que entre contenedora e interna no hay restricciones de acceso podemos eliminar set() y get().

CLASES INTERNAS ANÓNIMAS

Son las más utilizadas y por tanto las más importantes. Muchas de las cuestiones vistas en clases internas se aplican directamente a anónimas. La primera gran diferencia es que tienen una sintaxis retorcida, pero una vez asimilada esta sintaxis, resultan muy prácticas.

Una clase anónima siempre debe ser una subclase (extends) de una super clase o bien debe implementar (implements) alguna interfaz. La sintaxis para la definición de una clase de este tipo será:

```
Superclase var = new Superclase() { // Definicion anónima }
```

Superclase se refiere a una clase (abstracta normalmente) o a un interface y *var* será la variable que referencia a la instancia de la clase anónima, que será una subclase de *Superclase.*

Realmente lo que es la definición de la clase anónima es lo que pusimos en rojo de forma que podemos incluir dicha definición en un return, en un parámetro de un método o la derecha de una expresión de asignación como arriba.

El código en rojo, es una sintaxis especial para las clases anonimas que realmente hace simultáneamente dos cosas: definir una clase anónima y crear una instancia de la clase anónima.

Como se puede observar, la definición de la clase anónima y la creación de una instancia de la misma representan acciones inseparables que se realizan en la misma linea de código. Este doble efecto debe recordarte a lo que hace una expresión lambda, de hecho como veremos, una expresiones lambda tiene su equivalente clase interna anónima.

Ejemplo:

Fíjate ahora mucho en la sintaxis sin preocuparte para qué vale ya que eso lo demostraremos más adelante. Las clases internas anónimas se suelen emplear sobre todo para definir una clase que implementa un Interface, por eso en este ejemplo inicial tenemos un Interface

```
interface Saludo{
    String getSaludo();
}
class SaludoHola implements Saludo{
    @Override
    public String getSaludo() {
        return "Hola";
    }
}

class SaludoAdios implements Saludo{
    @Override
    public String getSaludo() {
        return "Adios";
    }
}

public class App{
    public static void main(String[] args) {
        Saludo hola= new SaludoHola();
        System.out.println(hola.getSaludo());

        Saludo adios= new SaludoAdios();
        System.out.println(adios.getSaludo());
    }
}
```

Ahora el mismo efecto usando internas anónimas con las sintaxis:

```
Superclase var = new Superclase() {  Definicion de métodos de clase anónima}
```

Entre las llaves irá la definición de una clase con atributos y métodos. Como mínimo tendrá que incluir los métodos abstractos de la superclase. Por ejemplo:

```
interface Saludo {

    String getSaludo();
}

public class App {

    public static void main(String[] args) {
        Saludo hola = new Saludo() {
            int a;

            int métodoA() {
                return a;
            }

            @Override
            public String getSaludo() {
                return "HOLA";
            }
        };

        System.out.println(hola.getSaludo());
```

```
    }
}
```

Recuerda que no se puede hacer un new de un interface, no se puede crear directamente un objeto de tipo interface, pero en esta sintaxis el new tiene un significado especial y realmente el objeto creado sería de *la clase concreta sin nombre* cuyo código se indica entre llaves.

En la práctica las clases internas anónimas suelen simplemente incluir el código de los métodos que se ve obligada a implementar. Es raro que incluyan métodos y atributos extra, de tal forma que el ejemplo inicial de saludos ahora escrito con clases anónimas quedaría

```
interface Saludo {

    String getSaludo();
}

class SaludoHola implements Saludo {

    @Override
    public String getSaludo() {
        return "Hola";
    }
}

class SaludoAdios implements Saludo {

    @Override
    public String getSaludo() {
        return "Adios";
    }
}

public class App {

    public static void main(String[] args) {
        Saludo hola = new Saludo() {public String getSaludo() {return "Hola";}};
        System.out.println(hola.getSaludo());

        Saludo adios = new Saludo() {public String getSaludo() {return "Adios";}};
        System.out.println(adios.getSaludo());
    }
}
```

Ejercicio: Observa cuestiones de estilo al escribir clases anónimas.
Por motivos didácticos deliberadamente en el ejemplo anterior escribimos las definiciones en una única línea. Si la clase anónima tiene mucho texto quizá no quepa en una línea. No debes usar líneas a lo loco: para ver la forma "standard" de escribirlas usa el "format" del IDE y aprenderás rápido el standard de estilo.

CLASES INTERNAS ANÓNIMAS Y EXPRESIONES LAMBDA

Recuerda que las expresiones lambda implementan interfaces funcionales, es decir, interfaces con un único método abstracto. También podemos usar para los interfaces funcionales clases internas anónimas. Actualmente hay una tendencia clara a que las expresiones lambda desplacen a las clases internas anónimas en los casos en que se pueden usar expresiones lambda. No obstante, recuerda que las clases internas anónimas se pueden aplicar a interfaces no funcionales, por lo tanto, siempre les queda un uso.

Ejemplo: Como Saludo es un interface funcional (sólo tiene un método) puedo implementar la interface con una lambda

```
interface Saludo{
   String getSaludo();
}

public class App{
   public static void main(String[] args) {
      Saludo talogo= ()->"ata logo amigo";
      System.out.println(talogo.getSaludo());
   }
}
```

¡que fácil con expresiones lambda!

Ejemplo: a un ejercicio realizado anteriormente con lambda le añadimos su equivalente con interna anónima

```
 import java.util.function.IntPredicate;

class MiIntPredicate implements IntPredicate {

    @Override
    public boolean test(int i) {
        return i % 2 == 0;
    }

}

public class App {

    public static void main(String args[]) {
        int[] lista = {1, 2, 3, 4, 5, 6, 7, 8, 9};

        System.out.println("Imprimir números int pares:");
        //1. con expresionλ
        eval(lista, n -> n % 2 == 0);

        //2. creando clase concreta que implementa el interface
        eval(lista, new MiIntPredicate());

        //3. con clase interna anónima
        eval(lista, new IntPredicate() {
            public boolean test(int i) {
                return i % 2 == 0;
            }
        });

    }

    public static void eval(int[] list, IntPredicate predicate) {
         System.out.println();
        for (int n : list) {

            if (predicate.test(n)) {
                System.out.print(n + " ");
            }
        }
    }
}
```

Como vés meter la clase Interna anónima como argumento es "farragoso", la concisión de la lambda se ve bien en este ejemplo

Ejercicio U8_B3_E2: Escribe como clase anidada de App el código del comparador. Pues definirla como static, interna miembro o interna local (o de las tres formas). Cómo anónima

lo dejamos para el siguiente ejercicio. Como el main es static salen unas combinaciones un poco retorcidas pero las intentamos sólo a título de curiosidad ya que son bastante inhabituales.

```
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

class Articulo {
    String codArticulo;
    String descripcion;
    int cantidad;
    Articulo(String codArticulo, String descripcion, int cantidad) {
        this.codArticulo = codArticulo;
        this.descripcion = descripcion;
        this.cantidad = cantidad;
    }


}

class ComparadorArticulos implements Comparator<Articulo>{
    @Override
    public int compare( Articulo o1, Articulo o2) { return o1.codArticulo.compareTo(o2.codArticulo); }
}

class App {

    public static void main(String[] args) {
        LinkedList<Articulo> articulos = new LinkedList<Articulo>();
        articulos.add(new Articulo("34","cuchillo",5));
        articulos.add(new Articulo("12","tenedor",7));
        articulos.add(new Articulo("41","cuchara",4));
        articulos.add(new Articulo("11","plato",6));

        Collections.sort(articulos, new ComparadorArticulos());
        for(Articulo a:articulos)
            System.out.println(a.codArticulo+", "+a.descripcion+", "+a.cantidad);
    }
}
```

LOS EJERCICIOS 3 Y 4 SON LO MÁS IMPORTANTE DE ESTE BOLETÍN

Ejercicio U8_B3_E3: Escribe como clase interna **anónima** de App el código del comparador. Esta forma es más habitual que las posibilidades vistas en el ejercicio anterior.

Ejercicio U8_B3_E4: El interface Comparator es un un interface funcional, así que puedo escribir el comparador anterior con expresionλ

De los ejercicios anteriores debes sacar las siguientes conclusiones:

- Las expresiones lambda generan código más limpio y sencillo. Muchas líneas de una clase interna anónima se convierten en una única línea. Se dice que la forma de escribir una clase interna anónima es "vertical" (muchas líneas) y la de la expresión lambda "horizontal" (1 línea). Lo "Vertical" siempre es más difícil de leer a "un golpe de vista".
- Cuando escribes una clase interna anónima piensas en un objeto que recibe un mensaje … y ciertamente es así, pero cuando escribes una lambda expressión piensas

a un mayor nivel de abstracción, te dedicas simplemente a decir "haz esto", es decir, declaras (programación declarativa) lo que quieres no como lo quieres.

- Java no es ni mucho menos un lenguaje de programación funcional puro, así que para desenvolverse en java cómodamente con las expresiones lambda no podemos perder de vista la relación expresión lambda/interface funcional
- Aunque se prefieren las expresiones lambda, las clases internas anónimas pueden hacer más cosas ya que:
    - Pueden implementar interfaces con más de un método abstracto
    - Pueden tener estado, es decir, podemos añadirle atributos si nos interesa que la clase tenga estado.

Un ejemplo, inútil, que sólo pretende observar estas capacidades de las clases internas anónimas que no tienen las lambda

```
interface MiInterface{
   void hacerA();
   void hacerB();
}


public class App{
   public static void main(String[] args) {
      MiInterface mi= new MiInterface(){
         public int x=0;
         public void hacerA() {
            throw new UnsupportedOperationException("Not supported yet.");
         }

         @Override
         public void hacerB() {
            throw new UnsupportedOperationException("Not supported yet.");
         }

      };

   }

 }
```

LAS FUNCIONES LAMBDA PUEDEN ACCEDER A LA CLASE CONTENEDORA(poco importante)

Las expresiones lambda no tienen estado pero sí pueden acceder al estado de la clase que las contienen. Recuerda que una clase anidada puede acceder a los atributos de su contenedora y ya que una lambda realmente es una clase interna anónima …

En el siguiente ejemplo tanto la interna anónima como la lambda accede a la variable local x del método main(). Similarmente, también hay combinaciones en las que podrían acceder a atributos de App

```
interface MiInterface{
   int  miMetodoA(int i);
}
public class App{
   public static void main(String[] args) {
      int x=7;
      MiInterface mi1= new MiInterface(){
         @Override
         public int miMetodoA(int i) {
```

```
            return i+x;
        }

    };
    System.out.println(mi1.miMetodoA(3));
    MiInterface mi2=i-> i+x;
    System.out.println(mi2.miMetodoA(3));
  }

}
```