

# Clases Anidadas y clases Internas

# Clases Anidadas y clases Internas

Las clases que trabajamos hasta ahora son **clases de nivel superior**: son miembros directos de **paquetes** y se definen en forma **independiente de otras clases**.

- Los **clases anidadas y las internas** son clases definidas **dentro de otras clases**.
- Existen sólo para **servir a la clase que la anida**. Si son útiles en otros contextos, entonces deben definirse como clases de nivel superior.
- Las **clases anidadas NO definen una relación de composición** entre objetos.
- Pueden declararse **private** o **protected** a diferencia de las de nivel superior que sólo pueden ser **public** o tener accesibilidad de **default** o **package**.
- Hay de **4 tipos**: clases miembro estáticas, no-estáticas, anónimas y locales.
- A las **clases miembro estáticas** también se las conoce como **clases internas**.

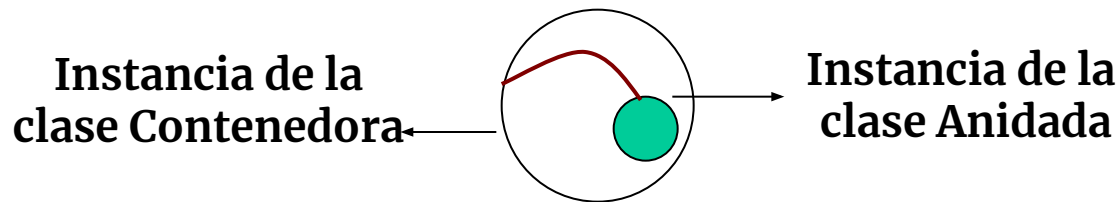
```
class Contenedora{  
    private int x=1;  
    static class Interna {  
        //TODO  
    }  
    class Anidada {  
        //TODO  
    }  
}
```

Una **clase interna** tiene **acceso** a todos los miembros declarados **static** de la clase que la contiene, aún aquellos que son privados. **No está ligada a ninguna instancia particular**. Sólo existe para la clase que la contiene.

Una **clase anidada** tiene **acceso** a la **implementación** de la clase que la contiene (variables de instancia, de clase y métodos) como si fuesen propios. **Es una relación entre objetos**.

# Clases Anidadas y clases Internas

- Las **clases anidadas** del tipo **miembro no-estático** son similares a los métodos de instancia o a las variables de instancia. **Sus instancias se asocian a cada instancia de la clase que la contiene.**
- Un objeto de una **clase anidada** tiene **acceso ilimitado a la implementación** del objeto que lo anida, inclusive aquellos declarados **private**.
- Un objeto de una **clase anidada** tiene una **referencia implícita al objeto de la clase que lo instanció** (clase contenedora). A través de dicha referencia tiene acceso al estado completo del objeto contenedor, inclusive a sus datos privados. Por lo tanto las **clases anidadas** tienen **más privilegios de acceso que las de nivel superior.**
- El compilador agrega la **referencia implícita en el constructor** de la **clase anidada**. Es invisible en la definición de la clase anidada.



Una instancia de una clase anidada está siempre asociada con una instancia de la clase contenedora

- Las **clases internas son miembros estáticos**, no tienen acceso a esta referencia implícita, sólo pueden acceder a los miembros declarados estáticos (inclusive los privados).
- Sintácticamente las clases miembro no-estáticas y estáticas son similares, difieren en que las estáticas tienen el modificador **static** en su declaración.

# Ejemplo de clase anidada

```
public class Paquete {  
    class Contenido {  
        private int i = 11;  
        public int valor() {  
            return i;  
        }  
    }  
    class Destino {  
        private String etiqueta;  
        Destino(String donde) {  
            etiqueta = donde;  
        }  
        String leerEtiqueta() {  
            return etiqueta;  
        }  
    }  
    public void vender(String dest) {  
        Contenido c = new Contenido();  
        Destino d = new Destino(dest);  
        System.out.println(d.leerEtiqueta());  
    }  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        p.vender("Roma");  
    }  
}
```

## Clases Anidadas

En los métodos de instancia, las clases anidadas se usan de la misma manera que en las clases de nivel superior

Declaración de objetos de la clase anidada

```
public class Paquete {  
    class Contenido {  
        //código  
    }  
    class Destino {  
        //código  
    }  
    public Destino hacia(String s) {  
        return new Destino(s);  
    }  
    public Contenido cont() {  
        return new Contenido();  
    }  
    public void vender(String dest) {  
        Contenido c = cont();  
        Destino d = hacia(dest);  
        System.out.println(d.leerEtiqueta());  
    }  
} // Fin de la clase Paquete
```

La clase contenedora define métodos que devuelven referencias a objetos de la clase anidada

```
public class Viaje {  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        p.vender("Roma");  
        Paquete q = new Paquete();  
        Paquete.Contenido c = q.cont();  
        Paquete.Destino d = q.hacia("Buenos Aires");  
    }  
}
```

# Clases anidadas & ocultamiento de implementación

¿Puedo ocultar una clase sin usar clases anidadas? **SI!!!** ¿Cómo?

Definiendo a la clase con acceso de **default** o **package**: la clase es visible solamente dentro del paquete donde se declaró.

¿Para qué usamos clases anidadas?

```
public interface Contenido {
    int valor();
}
public interface Destino {
    String leerEtiqueta();
}
```

Clase anidada privada: es accesible solamente desde la clase Paquete

```
package turismo;
public class Viaje {
    public static void main(String[] args) {
        Paquete p = new Paquete();
        Contenido c = p.cont();
        Destino d = p.hacia("Buenos Aires");
        System.out.println(d.leerEtiqueta());
        System.out.println(c.valor());
    }
}
```

Para proveer **ocultamiento de detalles de implementación**

Se obtiene una referencia, solamente a través **Upcasting/Generalización** a una clase base o interface, públicas.

```
public class Paquete {
    private class PContenido implements Contenido{
        private int i=11;
        public int valor() {return i;}
    }
    private class PDestino implements Destino{
        private String etiqueta;
        private PDestino(String donde){
            etiqueta=donde;
        }
        public String leerEtiqueta() {
            return etiqueta;
        }
    }
    public Destino hacia(String s) {
        return new PDestino(s);
    }
    public Contenido cont() {
        return new PContenido();
    }
} // Fin de la clase Paquete
```

**Upcasting** al tipo de la interface

Las **clases anidadas privadas que implementan interfaces** son completamente invisibles e inaccesibles y de esta manera se **oculta la implementación**. Se evitan dependencias de tipos. Desde afuera de la clase **Paquete** se obtiene una **referencia al tipo de la interface pública**.

# Acceso a los miembros de la clase contenedora

En la clase anidada **SIterador** se hace referencia a la variable **objetos** que es un atributo privado de la clase contenedora **Secuencia**.

```
public interface Iterador
{
    boolean fin();
    Object actual();
    void siguiente();
}
```

La interface **Iterador** se usa en la clase **Secuencia** para recorrer secuencias de objetos

```
public class TestSecuencia{
    public static void main(String[] args) {
        Secuencia s = new Secuencia(10);
        for (int i=0; i<10 ; i++)
            s.agregar(Integer.toString(i));
        Iterador it = s.getIterador();
        while (!it.fin()) {
            System.out.println(it.actual());
            it.siguiente();
        }
    }
}
```

Recorreremos una **Secuencia**

```
public class Secuencia{
    private Object[] objetos;
    private int sig = 0;
    public Secuencia(int tamaño){
        objetos = new Object[tamaño];
    }
    public void agregar(Object x){
        if (sig < objetos.length)
            objetos[sig++] = x;
    }
```

La clase **SIterador** se declaró privada: es inaccesible para los usuarios de la clase **Secuencia**.

```
private class SIterador implements Iterador{
    private int i = 0;
    public boolean fin(){ return (i== objetos.length); }
    public Object actual(){ return objetos[i]; }
    public void siguiente(){
        if (i< objetos.length) i++; }
}
```

```
public Iterador getIterador(){return new SIterador(); }
} // Fin de la clase Secuencia
```

Se crea un objeto **Iterador** asociado a un objeto **Secuencia**

Las clases anidadas pueden acceder a métodos y atributos de la clase contenedora como si fuesen propios.

# Implementación de patrones con clases anidadas

Las clases anidadas se usan en el framework de colecciones para implementar:

- **iteradores** que acceden a los elementos de la colección secuencialmente sin exponer la representación interna. Usados en implementaciones de List y Set.
- **adaptadores** que permiten a las implementaciones de Map implementar sus vistas de colección, mediante algunos métodos de Map.

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable {
```

```
    public Set<K> keySet() {
        Set<K> ks = keySet;
        return (ks != null ? ks : (keySet = new KeySet()));
    }
```

```
    private final class KeySet extends AbstractSet<K> {
        // código de la clase anidada
    }
```

**Adapter**

**keySet()** devuelve un **Set** y así un **HashMap** puede tratarse como un **Set**. El Set está soportado por el Map.

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess,
Cloneable, java.io.Serializable {
```

```
    public Iterator<E> iterator() {
        return new Itr();
    }
```

```
    private class Itr implements Iterator<E> {
        // código de la clase anidada
    }
```

**Iterator**

**Iterator** es una interface pública del framework de colecciones. **ArrayList** delega en el iterador el recorrido de la lista, mediante la implementación de **Itr**.

# Clases Locales

Las **clases locales** se definen dentro de un método o dentro de un bloque de código. Se declaran sin especificador de acceso dado que su alcance está restringido al bloque de código donde se definió. **NO pueden declararse *public*, *protected*, *private* ni *static***. Las interfaces y los tipos enumerativos **NO** pueden definirse localmente. Una **clase local** es similar a una **variable local**: es visible solamente dentro del bloque de código donde se definió

```
public class Paquete {
    public Destino hacia(String s) {
        class PDestino implements Destino {
            private String etiqueta;
            private PDestino(String donde){
                etiqueta=donde;
            }
            public String leerEtiqueta(){
                return etiqueta;
            }
        } // Fin de la clase PDestino

        return new PDestino(s);
    } // Fin del método hacia()
} // Fin de la clase Paquete
```

```
package turismo;
public class Viaje {
    public static void main(String[] args) {
        Paquete p = new Paquete();
        Destino d = p.hacia("Buenos Aires");
        System.out.println(d.leerEtiqueta());
    }
}
```

Las instancias de **clases locales** están asociadas con instancias de la clase que la contiene, por lo tanto pueden **acceder a TODOS** sus miembros incluyendo los privados.

-**PDestino** es una **clase interna local**.

-**Solamente adentro del método **hacia()** se pueden crear objetos PDestino.**

-**PDestino** forma parte del método **hacia()** en vez de ser parte de la clase Paquete. La clase **PDestino** **NO** puede ser accedida afuera del método **hacia()**, excepto a través de una referencia a la interface **Destino**

Lo único que sale del método **hacia()** es una referencia a **Destino**.  
Upcasting

-Una vez que el método **hacia()** terminó de ejecutarse, el objeto **PDestino** (*upcasteado* a Destino) es un objeto válido, es accesible.

-La clase **PDestino** a pesar de estar definida localmente en el método **hacia()** se compila con el resto de la clase (es un .class separado).


-La clase **PDestino** no está disponible afuera del método, está fuera de alcance (no se pueden crear objetos **PDestino** afuera del método **hacia()**), es un nombre inválido.



# Clases Locales

Anidar una clase **en un alcance arbitrario**:

```
public class PaqueteCondicional {  
    private void tramoInterno(boolean b) {  
        if(b) {  
  
            class UnPaseo {  
                private String id;  
                UnPaseo(String s) {  
                    id = s;  
                }  
                String getPaseo() { return id; }  
            } //Fin clase UnPaseo  
  
            UnPaseo up = new UnPaseo("Villa Traful");  
            String s = up.getPaseo();  
        }  
    }  
    public void tramo() { tramoInterno(true); }  
}
```



**UnPaseo** es una **clase local** definida dentro del **bloque if**. No implica que la clase se cree condicionalmente: la clase se compila con el resto de la clase, sin embargo no está disponible fuera del alcance donde se definió.

La clase **UnPaseo** está fuera de alcance. No se reconoce el nombre.

```
package turismo;  
public class Viaje {  
    public static void main(String[] args) {  
        PaqueteCondicional p = new PaqueteCondicional();  
        p.tramo();  
    }  
}
```

# Clases Anónimas

Las **clases anónimas** son **clases locales** sin nombre. Se crean extendiendo una clase o implementando una interface. **Combinan** la sintaxis de **definición de clases** con la de **instanciación de objetos**. Las interfaces y los tipos enumerativos NO pueden definirse anónimamente.

El método `cont()` combina la creación del valor de retorno con la definición de la clase que representa el valor retornado.

- Crea un objeto de una clase anónima que implementa la interface **Contenido**.
- La referencia que devuelve el método `cont()` es automáticamente *upcasteada* a una referencia a **Contenido**.

```
public class Paquete {
    public Contenido cont() {
        return new Contenido(){
            private int i=11;
            public int valor(){
                return i;
            }
        };
    }
} // Fin de la clase Paquete
```

Es una **abreviatura** de la declaración de una clase que implementa la interface **Contenido**

```
public interface Contenido {
    int valor();
}
```

```
public Contenido cont() {

    class PContenido implements Contenido {
        private int i=11;
        public int valor(){ return i;}
    }

    return new PContenido();
}
```

```
package turismo;
public class Viaje {
    public static void main(String[] args) {
        Paquete p = new Paquete();
        Contenido c = p.cont();
    }
}
```

# Clases Anónimas

Las clases anónimas son simultáneamente declaradas e instanciadas en el punto en que se van a usar.

```
public class Datos{  
    private int i;  
    public Datos(int x){i=x;}  
    public int valor(){return i;}  
}
```

La clase base, **Datos**, requiere un constructor con un argumento.

```
public class Paquete {  
    public Datos info(int x) {  
        return new Datos(x){  
            public int valor(){  
                return super.valor()*50;  
            }  
        };  
    }  
} // Fin de la clase Paquete
```

El método **info()** crea un objeto de una clase anónima que es subclase de **Datos** usando el constructor con un argumento de la superclase.

Es una abreviatura de la declaración de una clase que extiende la clase **Datos**

```
public Datos info(int x) {  
    class MisDatos extends Datos {  
        public MisDatos(int y){super(y);}  
        public int valor(){  
            return super.valor()*50;  
        } //Fin clase MisDatos  
    }  
    return new MisDatos(x);  
}
```

```
package turismo;  
public class Viaje {  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        Datos d = p.info(10);  
        System.out.println(d.valor());  
    }  
}
```

# Clases Anónimas

- Las **clases locales** y las **anónimas** tienen acceso a las **variables locales** del bloque de código donde están declaradas.

En este ejemplo, la clase anónima accede al parámetro String del método **hacia()**: inicializa la variable de instancia **etiqueta** con el valor del parámetro **donde**.

- Las **variables locales**, **parámetros de métodos** y **parámetros de manejadores de excepciones** que se usan en las clases locales y anónimas son **efectivamente final**.  
**¿Por qué?** El tiempo de vida de una instancia de una clase local o anónima es mayor que el tiempo de vida del método en el que se declaran y por ello es necesario preservar el estado de las variables locales a las que accede. Para asegurar esto, el compilador crea copias privadas de todas las variables locales que se usan y se reemplazan todas las referencias a las variables locales por referencias a las copias. La única manera de garantizar que las variables locales y sus copias contengan el mismo valor es obligando a las variables locales a ser **final**. **Java 8 introduce el concepto de efectivamente final**.
- El compilador se encarga de agregar un parámetro extra al constructor de la clase anónima (del tipo de la variable local) y una variable de instancia, en la que se mantendrá la copia.

```
public interface Destino {  
    String leerEtiqueta();  
}
```

efectivamente final

```
public class Paquete {  
    public Destino hacia(String donde) {  
        return new Destino() {  
            private String etiqueta=donde;  
            public String leerEtiqueta() {  
                return etiqueta;  
            }  
        };  
    }  
} // Fin de la clase Paquete
```

```
package turismo;  
public class Viaje {  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        Destino d = p.hacia("Buenos Aires");  
        System.out.println(d.leerEtiqueta());  
    }  
}
```

# Clases Anónimas

¿Puede definirse un constructor en una clase anónima?

**NO!!!**


No es posible pues la clase no tiene nombre y el constructor debe tener el mismo nombre que la clase.

¿Cómo podemos realizar una inicialización similar a la de un constructor?

Bloques de inicialización (de instancia)

```
public class Paquete {  
    public Destino hacia(String donde, float precio) {  
        return new Destino() {  
            private int costo;  
            private String etiqueta=donde;  
            {costo=Math.round(precio);  
                if (costo>100)  
                    System.out.println("Muy caro!!!");  
            }  
            public String leerEtiqueta(){return etiqueta;}  
        };  
    }  
} // Fin de la clase Paquete
```

**Bloque de Inicialización**



- El bloque de inicialización funciona como un constructor para la clase anónima. Se ejecuta cada vez que se crea una instancia.
- El uso del bloque de inicialización para definir constructores es limitado dado que NO es posible definir constructores sobrecargados.

Las clases anónimas son limitadas dado que sólo pueden extender una clase o implementar una interface, no pueden hacer ambas cosas a la vez ni tampoco pueden implementar más de una interface.

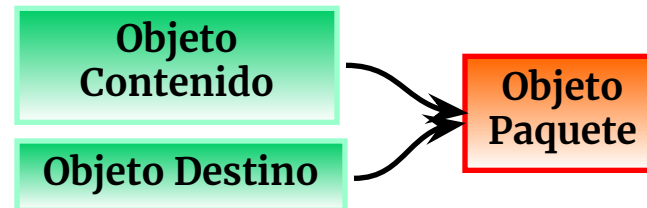
# ¿Cómo pueden las clases anidadas acceder a los miembros de la clase contenedora?

Cada **objeto de la clase anidada** mantiene una **referencia al objeto de la clase contenedora** que lo creó. De esta manera cuando nos referimos a un miembro de la clase contenedora (atributo o método) esta referencia “oculta” es usada.

El compilador se encarga de todos los detalles: agrega en el constructor de la clase anidada una referencia al objeto de la clase contenedora.

**SIEMPRE** un objeto de una clase anidada está asociado con un objeto de la clase contenedora: la construcción de un objeto de la clase anidada requiere de la referencia al objeto de la clase contenedora.

```
public class Paquete {  
    class Contenido {  
        private int i = 11;  
        public int valor() {return i;}  
    }  
    class Destino {  
        private String etiqueta;  
        Destino(String donde) {etiqueta = donde;}  
        String leerEtiqueta() {return etiqueta;}  
    }  
}
```



```
public class Viaje {  
    public static void main(String[] args) {  
        Paquete p = new Paquete();  
        Paquete.Contenido c = p.new Contenido();  
        Paquete.Destino d = p.new Destino("Buenos Aires");  
    }  
}
```

Para crear un objeto de la clase anidada es necesario usar un objeto de la clase contenedora (asociación manual)

# ¿Cómo nombrar al objeto de la clase contenedora?

```
private class SIterador implements Iterador{  
    private int i = 0;  
  
    public boolean fin(){return ( this.i == Secuencia.this.objetos.length );}  
  
    public Object actual(){return Secuencia.this.objetos[i];}  
  
    public void siguiente(){ if ( this.i < Secuencia.this.objetos.length) this.i++ ; }  
}
```

Es necesario usar esta sintaxis sólo si el nombre del atributo de la clase contenedora está **ocultado** por un atributo con el mismo nombre en la clase anidada

La sintaxis para nombrar al **objeto de la clase contenedora** es: **NombreDeLaClaseContenedora.this**

## ¿Cómo crear instancias de una clase anidada?

```
public Iterador getIterador(){return new SIterador();  
public Iterador getIterador(){return this.new SIterador();}
```

Al invocar al constructor de la clase anidada, **automáticamente** la instancia de la **clase anidada** se asocia con el **objeto this** de la **clase contenedora**.

Es posible **explicitar la instancia contenedora** cuando se crea el objeto de la clase anidada.

```
Paquete p=new Paquete();  
Paquete.Destino d=p.new Destino("Roma");  
Paquete.Contenido c=p.new Contenido();
```

Dependiendo de la visibilidad de las clases anidadas, es posible crear instancias fuera de la clase contenedora. En el ejemplo, es posible crear instancias de las clases Destino y Contenido en clases ubicadas en el paquete por *default*.

La sintaxis **.new** produce el alcance correcto, no es necesario calificar el nombre de la clase contenedora en la invocación al constructor

# Identificadores de clases anidadas

Las clases JAVA de alto nivel generan archivos **.class** que almacenan toda la información necesaria para crear objetos. Esta información produce una “meta-clase” llamada **objeto Class**.

Las **clases internas** también producen archivos **.class** que contienen la información de sus **objetos Class**. Los nombres de estos archivos cumplen la siguiente regla:

```
interface Contador {  
    int siguiente();  
}
```



Contador.class

ClaseAltoNivel.class

ClaseAltoNivel\$1ContadorLocal.class

ClaseAltoNivel\$2.class



```
public class ClaseAltoNivel{  
    Contador getContador(final String nom){  
        class ContadorLocal{  
            public ContadorLocal(){...}  
            public int siguiente(){....}  
        }  
        return new ContadorLocal();  
    }  
    Contador getContador2(final String nom){  
        return new Contador(){  
            public int siguiente(){....}  
        };  
    }  
}
```



# Clases anidadas y objetos-función

Java no ofrece funciones o referencias a funciones.

```
class StringLengthComparator {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

Un objeto  
**StringLengthComparator**  
expresa un  
**Objeto-Función**

Una instancia de una clase JAVA que exporta métodos que realizan operaciones sobre otros objetos pasados como parámetros, es un **objeto-función**.

Un objeto **StringLengthComparator** exporta un único método que toma 2 strings y devuelve un número negativo si el primer string es más corto que el segundo, cero si ambos strings tienen la misma longitud y un número positivo si el primer string es más largo que el segundo.

El método **compare()** permite ordenar strings de acuerdo a su longitud.

Un objeto **StringLengthComparator** es un “objeto función” o una “referencia” a un comparador, pudiendo ser invocado con un par de strings arbitrarios.

```
Lista.ordenar(stringArray[], new StringLengthComparator())
```

# Clases anidadas y objetos-función

Los objetos **StringLengthComparator** representan una **estrategia concreta** para comparar strings por longitud.

La clase **StringLengthComparator** **no tiene estado**: no tiene variables de instancia, todas las instancias son funcionalmente equivalentes. La **estrategia de comparación** debe definirse como un **singleton**.

```
class StringLengthComparator {  
    private StringLengthComparator() { }  
    public static final StringLengthComparator  
        INSTANCE = new StringLengthComparator();  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

**Estrategia Concreta**

`Lista.ordenar(args, StringLengthComparator.INSTANCE);`

Para pasar la instancia de **StringLengthComparator** a un método se necesita contar con un **tipo** apropiado como parámetro. No es recomendable usar el tipo **StringLengthComparator** porque no permite intercambiar estrategias de comparación. **¿Cómo intercambiar estrategias?**

Definimos una **interface genérica** para la estrategia: **Estrategia Abstracta**

```
public interface Comparator<T> {  
    public int compare(T t1, T t2);  
}
```

# Clases anidadas y objetos-función

```
class StringLengthComparator implements Comparator<String> {  
    private StringLengthComparator() { }  
    public static final Comparator<String>  
        INSTANCE = new StringLengthComparator();  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

**Estrategia Concreta**

Definimos las **estrategias abstractas** como interfaces y las **concretas** como implementaciones de dichas interfaces

## ¿Cómo usamos la estrategia de comparación de strings?

```
public class Lista {  
    public static void ordenar(String[] stringArray, Comparator<String> comparador)  
    {  
        for (int i = 0; i < stringArray.length-1; i++) {  
  
            for (int j = i+1; j < stringArray.length; j++)  
                if (comparador.compare(stringArray[i], stringArray[j]) > 0) {  
                    String aux = stringArray[i];  
                    stringArray[i] = stringArray[j];  
                    stringArray[j] = aux;  
                }  
        }  
    }  
}
```

# Clases anidadas y objetos-función

## ¿Cómo usamos la estrategia de comparación de strings?

Objeto Comparator

```
Listas.ordenar(listaDeStrings, StringLengthComparator.INSTANCE);
```

Las clases que representan **estrategias concretas** frecuentemente son definidas anónimas:

```
Listas.ordenar(listaDeStrings, new Comparator<String> () {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

En este último caso se crea una nueva instancia cada vez que se invoca al `ordenar()`

# Clases anidadas y objetos-función

Redefinimos el ejemplo usando la interface `java.util.Comparator` que es genérica, entonces es aplicable a cualquier tipo de comparadores:

```
package java.util;  
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object obj);  
}
```

**Estrategia Abstracta**

```
package anidadas;  
import java.util.Comparator;  
class StringLengthComparator implements Comparator<String>{  
    private StringLengthComparator() { }  
    public static final Comparator<String>  
        INSTANCE = new StringLengthComparator();  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
}
```

**Estrategia Concreta**

Método `sort()` de la clase `Arrays`:

```
public static <T> void sort(T[] a, Comparator<? super T> c)  
Arrays.sort(stringArray, StringLengthComparator.INSTANCE);
```

# Clases anónimas y objetos-función

```
package java.util;  
public interface Comparator<T> {  
    int compare(T o1, T o2);  
    boolean equals(Object obj);  
}
```

**La estrategia concreta  
definida como una  
Clase Anónima**

```
Arrays.sort(stringArray, new java.util.Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

# Clases anónimas y objetos-función

```
package anidadas;
import java.util.Arrays;
public class TestAnonimas {
    public static void main(String[] args) {
        String[] stringArray= {"hola", "chau", "hi", "goodbye"};
        Arrays.sort(stringArray, new java.util.Comparator<String>() {
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        });
        for (String s: stringArray)
            System.out.println(s);
    }
}
```

¿Cuál es la salida?

**hi**  
**hola**  
**chau**  
**goodbye**

# Clases anidadas y objetos-función

Usar **clases anónimas** en algunas circunstancias creará un objeto nuevo cada vez, por ejemplo si se ejecuta repetitivamente. Una solución más eficiente consiste en guardar la referencia al objeto función en una constante de clase y reusarla cada vez que se necesita.

La **interface que representa la estrategia** sirve como **tipo para todas las instancias de estrategias concretas**, por ello las clases que implementan estrategias concretas no necesitan ser públicas y, esto permite intercambiarlas. La clase **Host** **exporta constantes de clase del tipo de la interface de la estrategia** y las clases que implementan las estrategias pueden ser **clases anidadas privadas** de dicha clase.

```
package anidadas;
```

## Patrón Strategy

```
public class Host {  
    private static class StrLenCmp implements java.util.Comparator<String> {  
        public int compare(String s1, String s2) {  
            return s1.length() - s2.length();  
        }  
    }  
  
    public static final java.util.Comparator<String>  
        STRING_LENGTH_COMPARATOR = new StrLenCmp();  
}
```



# Clases anidadas y objetos-función

```
package anidadas;
import java.util.Arrays;

public class TestAnidadas2 {
    public static void main(String[] args) {
        String[] stringArray= {"hola", "chau", "hi", "goodbye"};

        Arrays.sort(stringArray, Host.STRING_LENGTH_COMPARATOR) ;
        for (String s: stringArray)
            System.out.println(s);
    }
}
```

**Síntesis:** los objetos-función permiten implementar el **patrón Strategy**. En JAVA este patrón se implementa declarando una **interface** que **representa la estrategia** y diferentes **clases** que implementan dicha interface, las **estrategias concretas**. Si la **estrategia concreta** se **usa sólo una vez**, entonces se declara e instancia como una **clase anónima**. Si una **estrategia concreta** se usa **repetitivamente** es conveniente definirla como una **clase interna privada** y **exportar la estrategia** mediante una **constante pública de clase** del tipo de la interface

De esta manera es posible **intercambiar** en ejecución las estrategias.

# Java 8, lambdas y clases anónimas

A partir de JAVA 8 se formalizó la noción que las interfaces con un único método que no requieren del estado de un objeto son especiales y merecen un tratamiento especial.

Estas interfaces se conocen como **interfaces funcionales** y JAVA permite crear implementaciones de estas interfaces usando **expresiones lambda o lambdas**.

Las lambdas son similares a las clases anónimas, en cuanto a su función, pero son más concisas.

```
Collections.sort(palabras,  
    (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

# Clases internas y el patrón Builder

Builder es un patrón de diseño creacional que permite construir objetos complejos paso a paso.

```
public class Car {  
    // Campos requeridos  
    private final String engine;  
    private final String transmission;  
  
    // Campos opcionales  
    private final boolean airbags;  
    private final boolean sunroof;  
    private final boolean gps;  
  
    // Constructor privado  
    private Car(Builder builder) {  
        this.engine = builder.engine;  
        this.transmission = builder.transmission;  
        this.airbags = builder.airbags;  
        this.sunroof = builder.sunroof;  
        this.gps = builder.gps;  
    }  
}
```

```
@Override  
public String toString() {  
    return "Car{" +  
        "engine=" + engine + " " +  
        ", transmission=" + transmission + " " +  
        ", airbags=" + airbags +  
        ", sunroof=" + sunroof +  
        ", gps=" + gps +  
        '}';  
}
```

```
public static void main(String[] args) {  
    // Ejemplo de uso del patrón Builder  
    Car car = new Car.Builder("V8", "Manual")  
        .withAirbags(true)  
        .withSunroof(true)  
        .withGPS(true)  
        .build();  
  
    System.out.println(car);  
}
```

El **Builder** se usa para construir una instancia de **Car** con la configuración deseada

# Clases internas y el patrón Builder

```
// Clase interna estática Builder
public static class Builder {
    // Campos requeridos
    private final String engine;
    private final String transmission;

    // Campos opcionales con valores predeterminados
    private boolean airbags = false;
    private boolean sunroof = false;
    private boolean gps = false;

    // Constructor para los campos requeridos
    public Builder(String engine, String transmission) {
        this.engine = engine;
        this.transmission = transmission;
    }
}
```

```
// Métodos para configurar los campos opcionales
public Builder withAirbags(boolean airbags) {
    this.airbags = airbags;
    return this;
}

public Builder withSunroof(boolean sunroof) {
    this.sunroof = sunroof;
    return this;
}

public Builder withGPS(boolean gps) {
    this.gps = gps;
    return this;
}

// Método para construir el objeto Car
public Car build() {
    return new Car(this);
}
} // Fin clase interna Builder
} // Fin clase Car
```