



Clases Anidadas

Clases Anidadas

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

- Los **clases anidadas** son clases definidas adentro de otras clases.
- Las **clases anidadas** deben existir 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 establecen una relación de composición entre objetos.
- Las **clases anidadas** 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 4 tipos de **clases anidadas**: clases miembro no-estáticas, clases miembro estáticas, clases anónimas y clases locales.
- A las **clases miembro estáticas** también se las conocen como **clases internas**.

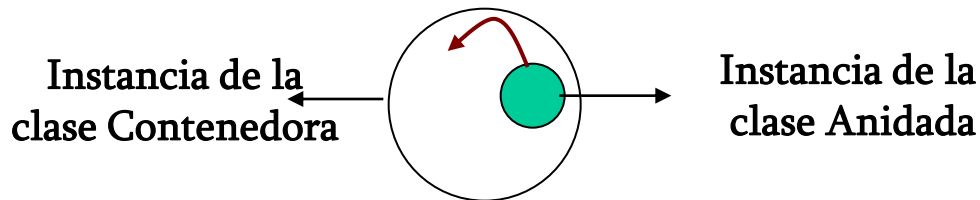
```
class Contenedora{  
    private int x=1;  
    static class Interna {  
        .....  
    }  
    class Anidada {  
        .....  
    }  
}
```

La clase Interna tiene acceso a todos los miembros declarados “static” de la clase que la anida, aún aquellos que son privados. No está ligada a ninguna instancia particular. Sólo existe para la clase que la contiene.

La 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

- 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**.
- Las **clases anidadas** son usadas extensamente para escribir programas que manejan eventos de GUI (Graphic User Interface).
- Un objeto de una **clase anidada** tiene una referencia implícita al objeto de la clase que lo instanció (clase contenedora). A través de esta 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 es el que agrega esta referencia implícita en el constructor de la **clase anidada**. Es invisible en la definición de la clase interna.



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 aquellos que son 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 interna

```
public class Paquete {  
    class Contenido {  
        //código  
    }  
    class Destino {  
        //código  
    }  
    public Destino hacia(String s) { anidada  
        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

```
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 adentro del paquete donde se declaró.

¿Para qué usamos clases anidadas?

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

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

Clase 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**

Solamente a través **Upcasting/Generalización** a una clase base o interface pública, se obtiene una referencia

```
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.

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.

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

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

Uso 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 JAVA para implementar **iteradores** que acceden a los elementos de una colección secuencialmente sin exponer la representación interna.

Las clases anidadas también implementan **adapters** que permiten a una instancia de una clase de nivel superior ser "vista" como instancia de alguna clase no relacionada.

```
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable {
    // métodos de la clase HashMap
    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

```
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
    }
}
```

Iterador

Devuelve un Set y de esta manera un HashMap puede tratarse como un Set. El Set está soportado por el Map.

Clases Locales

Las **clases locales** se definen dentro de un método o dentro de un bloque de código JAVA. 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;  
            }  
        }  
  
        return new PDestino(s);  
    }  
} // 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; }  
            }  
  
            UnPaseo up = new UnPaseo("Villa Traful");  
            String s = up.getPaseo();  
        }  
    }  
    public void tramo() { tramoInterno(true); }  
}
```



UnPaseo es una clase local definida adentro 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 afuera 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.tramoInterno();  
    }  
}
```

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.

```
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**

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

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

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

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**

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

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

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 el atributo 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 deben declararse **final**. El tiempo de vida de una instancia de una clase local o anónima es mayor que el tiempo de vida de la ejecución 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 se crean copias privadas de todas las variables locales que se usan (lo hace automáticamente el compilador) 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 los mismos valores es obligando a las variables locales a ser **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();
}
```

```
public class Paquete {
    public Destino hacia(final 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 al estilo de un constructor?

Bloques de inicialización (de instancia)

```
public class Paquete {  
    public Destino hacia(final String donde, final 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.

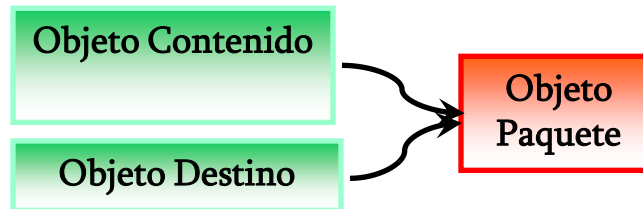
¿Cómo pueden las clases anidadas acceder a los miembros de la clase contenedora?

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();}
```

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.

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

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 afuera 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

Clases Internas

Las **clases internas** o clases miembro estáticas deben declararse **static**. Las **clases internas** NO contienen una referencia al objeto de la clase contenedora que lo creó.

- 1 No se necesita un objeto de la clase contenedora para crear un objeto de una clase interna.
- 2 No se puede acceder a objetos no-estáticos de la clase contenedora desde una clase interna.
- 3 Las **clases anidadas (no estáticas)** NO pueden tener atributos **static**, métodos **static**, ni **clases internas**.

```
public class Paquete10 {  
    private static class PaqueteContenido implements Contenido {  
        private int i=11;  
        public int valor(){return i;}  
    }  
    protected static class PaqueteDestino implements Destino {  
        private String etiqueta;  
        static int x=20;  
        public PaqueteDestino(String donde){etiqueta=donde;}  
        public String leerEtiqueta(){ return etiqueta;}  
        public static void f(){.....}  
        static class Nivel2{  
            static int x=10;  
            public void f(){}  
        }  
    }  
    public static Contenido cont(){ }  
    return new PaqueteContenido();  
}
```

La clase interna no puede acceder a las variables ni métodos de instancia de la clase contenedora

```
public class Viaje{  
    public static void main(String[] args) {  
        Contenido c=Paquete10.cont();  
        Paquete10.PaqueteDestino d= new Paquete10.PaqueteDestino("Barcelona");  
        d.leerEtiqueta();  
        Paquete10.PaqueteDestino.Nivel2 n=new Paquete10.PaqueteDestino.Nivel2();  
        n.f();  
    }  
}
```

No se necesitan objetos de tipo Paquete10 para crear objetos de las clases anidadas.

Objetos Función

Java no provee objetos-función o referencias a funciones.

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

Un objeto
`StringLengthComparator`
puede expresar
Objetos-Función

Una instancia de una clase JAVA que exporta métodos que realizan operaciones sobre otros objetos pasados explícitamente como parámetros, expresa **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 un puntero a un comparador, pudiendo ser invocado con un par de strings arbitrarios.

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

Objetos Función

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

La clase **StringLengthComparator** no tiene estado: no tiene variables de instancia y por ende 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 un **tipo** apropiado como parámetro. No es recomendable usar el tipo **StringLengthComparator** porque no permite intercambiar estrategias de comparación.

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

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

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 estas 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;  
                }  
        }  
    }  
}
```

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()`

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);
```

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();  
    }  
});
```

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

Objetos Función

Usar una clase anónima 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 ser intercambiables. Una “**host class**” puede exportar una constante de clase del tipo de la interface de la estrategia y la clase que implementa la estrategia puede ser una clase anidada privada de la “host class”.

```
package anidadas;
```

Clase Miembro Estática Privada

```
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();
```

```
}
```


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);
    }
}
```

Resumen: el principal uso de objetos-función es 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.

Identificadores de Clases Anidadas

Las clases JAVA de alto nivel generan archivos **.class** en donde se guarda toda la información necesaria para crear objetos de ese tipo. 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

- El nombre de clase de alto nivel, seguido del signo \$, seguido del nombre de la clase anidada.
- Si la clase es anónima, el compilador genera un número como identificador de la clase.
- Si la clase anidada está anidada adentro de otra anidada, sus nombres se agregan después de un \$ y del nombre de la clase externa

```
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(){....}  
        };  
    }  
}
```

Resumen

- Hay 4 diferentes tipos de clases anidadas.
- Si las instancias de la clase miembro requieren una referencia a la instancia de la clase que la contiene, entonces debe definirse como una clase anidada (miembro de instancia). En otro caso, debemos declararla como una clase interna (miembro estática).
- Si una clase necesita ser visible afuera de un método o es demasiado extensa (muchas líneas de código) como para definirse adentro del alcance de un método, debe declararse como una clase miembro.
- Si la clase sólo tiene sentido adentro de un método y las instancias se crean en un único punto y además la clase está caracterizada por un tipo pre-existente, se debe declarar anónima, en otro caso, local.