Clases anidadas

Tecnología de la Programación

Curso 2019-2020

Jesús Correas – jcorreas@ucm.es

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(Basado en material creado por Samir Genaim, utilizando los apuntes de Marco Antonio Gómez y Jorge Gómez)

Introducción

- Las clases internas permiten realizar una agrupación lógica de clases que están relacionadas.
- Por ejemplo, se puede crear una clase auxiliar para evitar que una clase sea excesivamente compleja, pero no se desea que esa nueva clase esté disponible para su uso en otras clases diferentes.
- En Java es posible **definir clases dentro de otras clases.** Por ejemplo:

```
public class A {
  public void metodoA() { ... }

  public class B {
    public void metodoB() { ... }
  }
}
```

- Este concepto es distinto de la relación de herencia: la clase B no hereda los atributos y métodos de A (aunque puede acceder a ellos)

Ventajas de las clases anidadas

- En determinadas circunstancias es conveniente utilizar clases anidadas, en particular cuando se programan interfaces gráficos de usuario.
- Algunas ventajas del uso de clases anidadas son:
  - Permite agrupar clases que **sólo se utilizan en un lugar.** Son habituales en el caso de "clases auxiliares" (*helper classes*).
  - Incrementa la **encapsulación:**
    - ⋆ Si una A tiene atributos o métodos que **es conveniente que sean privados** y otra clase B necesita acceder a ellos, se puede hacer que B se defina dentro de A.
    - ⋆ Además, B **puede declararse ella misma privada**, ocultándola al resto del programa.
  - El código se puede hacer **más legible y mantenible:** situando clases pequeñas dentro de otras clases permite colocarlas cerca de donde se usan.

Tipos de clases anidadas

- Hay distintos tipos de clases anidadas:
  - **Clases anidadas estáticas** (se suele utilizar el término *static nested classes*).
  - **Clases anidadas no estáticas** (para estas clases se utiliza el término *inner classes*).
  - **Clases locales.**
  - **Clases anónimas.**
- Las clases anidadas pueden tener modificadores de visibilidad (`public`, `protected`, `private`).
- Puede haber varios niveles de anidamiento.
- Cuando se compila un fichero Java con clases anidadas, se generan ficheros .class para cada clase. Al fichero de la clase interna se le da el nombre `ClaseExterna$ClaseInterna.class`.
- También se pueden definir **interfaces anidados**.

Clases anidadas estáticas

- Una **clase es anidada estática** si **no hace referencia a ningún método o atributo no estático de la clase externa.**
- Si la clase externa es a su vez una clase anidada en otra, entonces obligatoriamente **también debe ser estática.**

```
public class Externa {
  private static int at1 = 3;
  private int at2 = 4;
  private int m1() { return at1*at2; }
  private int m2() { return Interna.at3*Interna.at4; } // error
  public static class Interna {
    private static int at3 = 5;
    private int at4 = 6;
    private int m3() { return at1*at1; }
    private int m4() { return m1()*at2; } // error
  }}
```

- Desde `Interna` se puede acceder a **todos los miembros (atributos y métodos) estáticos** de `Externa`, **aunque sean privados.**
- Igualmente, desde `Externa` se puede acceder a **todos los miembros** de `Interna`, aunque sean privados.

Acceso a clases anidadas estáticas

- Se puede acceder a métodos y atributos de la clase anidada desde la clase externa como si no fuera una clase anidada.
- La única diferencia es que los atributos y métodos privados **también son accesibles desde la clase externa.**
- Las clases anidadas **se pueden utilizar desde otras clases** además de la clase externa.
  - En este caso, deben ser visibles (con los modificadores de visibilidad adecuados: `public`, `protected`), tanto la clase externa como la anidada.
  - Para crear objetos o llamar a métodos estáticos se debe utilizar **el nombre de la clase externa y el de la clase anidada.** Por ejemplo: Externa.Interna.m1 o new Externa.Interna()
- Recuerda que puede haber **varios niveles de anidamiento.**

Acceso a clases anidadas estáticas – Ejemplo 1

- **¿El siguiente código es correcto?**

```
package tests;

public class C1 {
  static public class C2 {
    static public class C3 {
      static public class C4 {
        static public int f;
      }
    }
  }
}
```

```
package tests;

public class A {
  void m() {
    C1.C2.C3.C4.f = 1;
  }
}
```

Acceso a clases anidadas estáticas – Más ejemplos 2

- **¿El siguiente código es correcto?**

```
package tests;

public class C1 {
  static public class C2 {
    static private class C3 {
      static public class C4 {
        static public int f;
      }
    }
  }
}
```

```
package tests;

public class A {
  void m() {
    C1.C2.C3.C4.f = 1;
  }
}
```

Acceso a clases anidadas estáticas – Más ejemplos 3

- **¿El siguiente código es correcto?**

```
package tests;

public class C1 {
  static public class C2 {
    static protected class C3 {
      static public class C4 {
        static public int f;
      }
    }
  }
}
```

```
package tests;

public class A {
  void m() {
    C1.C2.C3.C4.f = 1;
  }
}
```

Acceso a clases anidadas estáticas – Más ejemplos 4

- **¿El siguiente código es correcto?**

```
package tests;

public class C1 {
  static public class C2 {
    static protected class C3 {
      static class C4 {
        static public int f;
      }
    }
  }
}
```

```
package tests;

public class A {
  void m() {
    C1.C2.C3.C4.f = 1;
  }
}
```

Acceso a clases anidadas estáticas – Más ejemplos 5

- **¿El siguiente código es correcto?**

```
package tests;

public class C1 {
  static public class C2 {
    static public class C3 {
      static public class C4 {
        static public int f;
      }
    }
  }
}
```

```
package otrosTests;
import tests;

public class A {
  void m() {
    C1.C2.C3.C4.f = 1;
  }
}
```

Acceso a clases anidadas estáticas – Más ejemplos 6

- **¿El siguiente código es correcto?**

```
package tests;

public class C1 {
  static public class C2 {
    static private class C3 {
      static public class C4 {
        static public int f;
      }
    }
  }
}
```

```
package otrosTests;
import tests;

public class A {
  void m() {
    C1.C2.C3.C4.f = 1;
  }
}
```

Acceso a clases anidadas estáticas – Más ejemplos 7

- **¿El siguiente código es correcto?**

```
package tests;

public class C1 {
  static public class C2 {
    static protected class C3 {
      static public class C4 {
        static public int f;
      }
    }
  }
}
```

```
package otrosTests;
import tests;

public class A {
  void m() {
    C1.C2.C3.C4.f = 1;
  }
}
```

Acceso a clases anidadas estáticas – Más ejemplos 8

- **¿El siguiente código es correcto?**

```
package tests;

public class C1 {
  static public class C2 {
    static protected class C3 {
      static class C4 {
        static public int f;
      }
    }
  }
}
```

```
package otrosTests;
import tests;

public class A {
  void m() {
    C1.C2.C3.C4.f = 1;
  }
}
```

Clases anidadas no estáticas (*Inner classes*)

- Una **clase anidada no estática o clase interna (inner class)** es una clase anidada que utiliza atributos o métodos no estáticos de la clase externa.
- Todos los miembros de la clase externa **son accesibles** desde la clase interna y viceversa, aunque sean privados.
- Algunas **limitaciones** de las clases internas son:
- **No pueden definir atributos o métodos estáticos:** para eso están las clases anidadas estáticas.
- Sólo se pueden crear objetos de una clase interna **asociados a una instancia de la clase que la contiene.**
  - Se puede crear **desde un método no estático de la clase externa.** En este caso la instancia a la que se asocia es el objeto actual (`this`) de la clase externa.
  - Se puede crear desde otra clase, **haciendo referencia al objeto de la clase externa** sobre el que se crea, utilizando la sintaxis

```
x.new Interna()
```

Clases anidadas no estáticas – Ejemplo

```
class Externa {
  private int a1 = 1;
  public Externa(int i) { a1 = i; }
  public Interna m1() { return new Interna(a1*a1); }
  public String toString() { return "a1=" + a1; }

  public class Interna {
    private int a2 = 6;
    public Interna(int i) { a2 = i; }
    public String toString() {
      return "a1=" + a1 + " a2=" + a2;//accedo a a1 directamente
} } }

public class Ejemplo2 {
  public static void main(String[] s) {
    Externa ex1 = new Externa(2);
    Externa.Interna in1 = ex1.m1();
    Externa.Interna in2 = ex1.new Interna(4);
    System.out.println(in1);
    System.out.println(in2);
} }
```

Acceso a miembros de la clase externa

- Cuando la clase externa y la interna tienen atributos con el mismo nombre:

```
public class Externa {
  private int atr = 1;

  public class Interna {
    private int atr = 2;
    public String toString() {
      return "this.atr: " + this.atr +
             "Externa.this.atr: " + Externa.this.atr;
} } }
```

- **Desde la clase interna** se puede acceder a dos objetos que corresponden a la instancia actual:
  - La instancia actual de la clase interna se accede con **`this`**.
  - La instancia de la clase externa asociada a `this` se accede con **`Externa.this`**.
- El uso del prefijo `this.` y `Externa.this.` **es opcional**; solo es obligatorio si el atributo o método de la clase externa **es ocultado por otro de la clase interna con el mismo nombre.**

Acceso a miembros de la clase externa – Ejemplo

```
class OuterClass {
  private int f = 1;
  public class InnerClass1 {
    private int f = 2;
    public class InnerClass2 {
      private int f = 3;
      public void test() {
        System.out.println(this.f + " " + InnerClass1.this.f);
      }
    }
    public void test() {
      System.out.println(this.f + " " + OuterClass.this.f);
} } }

public class Ejemplo3 {
  public static void main(String[] s) {
    OuterClass x = new OuterClass();
    OuterClass.InnerClass1 a = x.new InnerClass1();
    OuterClass.InnerClass1.InnerClass2 b = a.new InnerClass2();
    a.test();
    b.test();
} }
```

Acceso a miembros de la clase externa – Más ejemplos

- **¿El siguiente código es correcto?**

```
package first;
public class OuterClass {

  private class InnerClass {
    ...
  }
  ...
}

//-----------------------------------------------

package first;
public class SomeClass {
  ...
  OuterClass x = new OuterClass();
  OuterClass.InnerClass a = x.new InnerClass();
  ....
}
```

Acceso a miembros de la clase externa – Más ejemplos

- **¿El siguiente código es correcto?**

```
package first;
public class OuterClass {

  protected class InnerClass {
    ...
  }
  ...
}

//-----------------------------------------------

package first;
public class SomeClass {
  ...
  OuterClass x = new OuterClass();
  OuterClass.InnerClass a = x.new InnerClass();
  ....
}
```

Acceso a miembros de la clase externa – Más ejemplos

- **¿El siguiente código es correcto?**

```
package first;
public class OuterClass {

  class InnerClass {
    ...
  }
  ...
}

//-----------------------------------------------

package first;
public class SomeClass {
  ...
  OuterClass x = new OuterClass();
  OuterClass.InnerClass a = x.new InnerClass();
  ....
}
```

Acceso a miembros de la clase externa – Más ejemplos

- **¿El siguiente código es correcto?**

```
package first;
public class OuterClass {

  class InnerClass {
    ...
  }
  ...
}

//-----------------------------------------------

package second;
import first.*;
public class SomeClass {
  ...
  OuterClass x = new OuterClass();
  OuterClass.InnerClass a = x.new InnerClass();
  ....
}
```

Iteradores como clases internas

- **Recordatorio:** un iterador es un interfaz para permitir recorrer de forma sencilla una estructura de datos. Tiene tres métodos fundamentales:

```
boolean hasNext();
E next();
void remove();
```

- Vamos a implementar el interfaz iterador en una pila. Queremos permitir el siguiente funcionamiento en la clase `Stack`:

```
Stack<Integer> s = new Stack<Integer>();
s.push(1);
s.push(4);
...
Iterator<Integer> it = s.iterator();
while (it.hasNext()) {
  System.out.println(it.next());
}
```

Iteradores como clases internas

- La clase `Stack` sin iterador es la siguiente:

```
public class Stack<T> {
  private static final int initSize = 10;
  private Object[] elem;
  private int last;

  Stack() {
    elem = new Object[initSize];
    last = -1;
  }
  T pop() throws StackError {
    if ( last >= 0) return (T)elem[last--];
    else throw new StackError();
  }
  void push(T x) {
    if ( last == elem.length-1 ) resize();
    elem[++last]=x;
  }
  private void resize() { ... }
}
```

Iteradores como clases internas

- Primero vamos a hacer que la clase `Stack` implemente el interfaz **`Iterable<T>`** (es el que usa `Collection`), que está definido de la siguiente forma:

```
public interface Iterable<T> {
    Iterator<T> iterator();
}
```

- Por tanto, `Stack` debe contener un método `iterator()`.
- Este método debe devolver un objeto que implemente el interfaz `Iterator<T>` de forma que este iterador recorra los elementos de la pila.
- Además, **el recorrido del iterador no debe modificar el estado de la pila.** No es adecuado extraer los elementos de la pila.
- Para poder acceder a los **atributos privados** de la clase `Stack` **vamos a utilizar una clase interna** que implemente el interfaz `Iterator<T>`.

Iteradores como clases internas

```
public class Stack<T> implements Iterable<T> {
  private static final int initSize = 10;
  private Object[] elem;
  private int last;

  Stack() {
    elem = new Object[initSize];
    last = -1;
  }
  T pop() throws StackError {
    if ( last >= 0)! return (T)elem[last--];
    else throw new StackError();
  }
  void push(T x) {
    if ( last == elem.length-1 ) resize();
    elem[++last]=x;
  }
  private void resize() { ... }

  public Iterator<T> iterator() {
    return new Iter();
  }
  // Continúa la clase interna Iter en la siguiente transparencia
```

Iteradores como clases internas

```
  // Clase interna Iter, definida dentro de Stack.
  private class Iter implements Iterator<T> {
    int curr = 0; // posición de recorrido del iterador.

    @Override
    public boolean hasNext() {
      return curr <= last;
    }

    @Override
    public T next() {
      curr++;
      return elem[curr-1];
    }

    @Override
    public void remove() {}
} }
```

- **Importante:** Aunque `Iter` es una clase privada, el método `iterator()` **devuelve objetos de esa clase, de los que sólo se pueden utilizar los métodos del interfaz `Iterator<T>`.**

Clases locales

- Hemos visto que las clases internas (*inner classes*) se declaran dentro de otra clase, **pero fuera de cualquier método.**
- Las **clases locales** son un tipo particular de clases internas que se declaran **dentro de un bloque de código.**
- La clase es **privada al bloque donde está declarada.** Por tanto, no es necesario indicar un modificador de visibilidad.
- Las clases locales, como las clases internas, no pueden contener métodos o atributos estáticos. Solo pueden utilizar los métodos o atributos estáticos de la clase que la contiene (u otra clase accesible).

Clases locales

- Por ejemplo, se podría declarar la clase `Iter` del ejemplo anterior **dentro del método `iterator()`:**

```
public class Stack<T> implements Iterable<T> {
  ...
  public Iterator<T> iterator() {
    //clase local definida dentro del método iterator().
    class Iter implements Iterator<T> {
      ...
    }

    return new Iter();
  }
}
```

Clases locales

- Las clases locales se utilizan para implementar un interfaz o extender una clase **que sí es accesible desde fuera del bloque**, y el bloque devuelve un objeto de la clase padre o interfaz.
- Las clases locales tienen una característica que las diferencia de las clases internas: **pueden acceder a las variables locales y parámetros del bloque donde están declaradas.**
- Para ello, las variables locales y parámetros que se utilicen deben ser **`final`** (no modificables).

```
class OuterClass<T> implements Iterable<T> {
  ...
  Iterator<T> metodo(int i, final int j) {
    final int k = 5; //variables locales de metodo()
    int h;
    class Iter implements Iterator<T> {
      ...
        y = j*k; // accede a vars.locales y params. finales
    }
    return new Iter();
} }
```

Clases anónimas

- Las **clases anónimas** son como las clases locales, pero sin nombre.
- Se utilizan en aquellos casos en los que la clase local se va a utilizar **en un solo punto del bloque donde está declarada**.
- Se utilizan para crear una instancia de una clase que implementa un interfaz o hereda de otra clase.
- Las clases anónimas no pueden tener constructoras propias.
- La sintaxis es diferente, porque **mezcla la llamada a la constructora** de la superclase/interfaz de la que hereda **con la declaración de la clase anónima:**

```
public class Stack<T> implements Iterable<T> {
  ...
  public Iterator<T> iterator() {
    return new Iterator<T>() {
                  public boolean hasNext() {...}
                  public T next() {...}
                  public void remove() {...}
                };
  }
}
```

Clases anónimas

- La llamada a la constructora de la superclase puede contener parámetros.
- Una clase anónima se puede crear **en cualquier lugar donde se pueda utilizar una expresión** (argumento de una llamada a un método, etc.).
- Se utilizan en *Swing* para hacer *listeners* de componentes visuales.

```
public class HelloWorldSwing {
  private static void createAndShowGUI() {
    JFrame frame = new JFrame("HelloWorldSwing");
    JButton btn = new JButton("Hola, púlsame!");
    btn.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        // Acción al pulsar el botón.
        System.out.println("Has pulsado el botón.");
      }
    });
    frame.getContentPane().add(btn);
    frame.pack();
    frame.setVisible(true);
  } ... }
```