Programación Orientada a Objetos

Tema 3:
Sintaxis del lenguaje POO Java

Tema 3-1: Sintaxis OO de Java

Universidad de Alcalá
Departamento Ciencias de la Computación

Contenidos

- Tema 3-1: Sintaxis OO de Java

• La clase es la unidad fundamental de programación en Java.
• La clase define el interfaz y la implementación de los objetos de esa clase.
• Todo objeto es instancia de alguna clase.

• La declaración de una clase tiene la forma básica:

[***public***] [***final***] [***abstract***] ***class*** NombreClase {

...

}

Cuerpo de la clase
Aquí declaramos los **miembros de la clase: atributos** y **métodos**

public es visible desde otros paquetes
final esta clase no puede tener subclases
abstract clase abstracta

Universidad de Alcalá
CLASES
Departamento Ciencias de la Computación

- **Modificadores de clase:**

Los modificadores de clase son palabras reservadas que se anteponen al nombre de las clases con el fin de definirles un comportamiento concreto. Los tipos de modificadores de clase que podemos definir son:

abstract	Una clase abstracta tiene al menos un método abstracto. Una clase abstracta no se instancia, sino que se utiliza como clase base para la herencia.
final	Una clase final se declara como la clase que termina una cadena de herencia. No se puede heredar de una clase final.
public	Las clases public son accesibles desde otras clases, bien sea directamente o por herencia. Son accesibles dentro del mismo paquete en el que se han declarado. Para acceder desde otros paquetes, primero tienen que ser importadas.
-	Si no se especifica ningún tipo de modificador, se entiende que cualquier objeto que se encuentre en el mismo paquete que esta clase podrá hacer uso de ella.

ATRIBUTOS Y MÉTODOS

• Una clase en Java puede contener atributos (variables) y métodos. Los atributos pueden ser tipos primitivos como int, char, etc. u otros objetos. Los métodos son funciones. Tanto los métodos como los atributos se definen dentro del bloque de la clase. Java no soporta métodos o variables globales. Ejemplo:

```
public class MiClase {
        int i;                          //atributo i
        public MiClase() {              //constructor
           i = 10;
           }
        public void suma_a_i( int j ) {          //método
           i = i + j;
           }
        }
...
        MiClase mc1 = new MiClase();
        mc1.i++; // incrementa la instancia de i de mc1
        mc1.suma_a_i( 10 );
        MiClase mc2 = new MiClase();
        mc2.i++; // incrementa la instancia de i de mc2
        mc2.suma_a_i( 5 );
```

i = 21
mc1

i = 16
mc2

Universidad de Alcalá
Departamento Ciencias de la Computación

ATRIBUTOS Y MÉTODOS

• Atributos:

La sintaxis básica para declarar atributos es:

[final] [static] acceso tipo lista-de-identificadores

acceso: ***public*** | ***private*** | ***protected*** | -
tipo: es cualquier tipo primitivo o clase o array de ellos

final: significa que el valor de ese campo no puede ser modificado (constante)
static: significa que todas las instancias comparten la misma variable (atributo de clase)

Ejemplos:

```
private double minimo, maximo;
final static int maxInstancias = 10;
static float[] valores;
```

ATRIBUTOS Y MÉTODOS

• El ocultamiento de información tiene varios grados en Java.
• Todos los atributos y métodos de una clase son accesibles desde el código de la propia clase. Para controlar el acceso desde otras clases, se utilizan **modificadores de acceso**:

public	Accesible desde cualquier lugar en que sea accesible la clase
protected	Accesible por las subclases y clases del mismo paquete
private	Sólo accesible por la propia clase
- (friendly)	Accesible por las clases del mismo paquete

Universidad de Alcalá
Departamento Ciencias de la Computación

ATRIBUTOS Y MÉTODOS

- **Inicialización de los atributos:**

Java asigna un valor por defecto a los atributos que no se inicializan que depende del tipo de éstos:

Tipo del campo	Valor Inicial
boolean	0
char	'\u0000'
byte, **short**, **int**, **long**	0
float, **double**	+0.0f ó +0.0d
referencia a objeto	null

Sin embargo, no asigna ningún valor por defecto inicial a las variables locales de métodos o constructores.

Debemos asignar algún valor a una variable local antes de usarla.

ATRIBUTOS Y MÉTODOS

• Inicialización de los atributos:

Los atributos pueden:

• **No inicializarse**, con lo que tendrán los valores por defecto vistos anteriormente, según el tipo del campo.

• **Inicializarse en la declaración**, asignándoles una expresión. No se pueden invocar métodos que pueden lanzar excepciones.

```
private String miNombre = “Juan ” + “Carlos”;
private int edad = obtenerMiEdad();
```


Universidad de Alcalá
Departamento Ciencias de la Computación

ATRIBUTOS Y MÉTODOS

• Métodos:

La sintaxis básica para declarar métodos es:

[final] [static] [abstract] acceso tipo identificador(args) {...}

final:	Significa que el método no puede ser sobrecargado por las clases que hereden de esta.
static:	Significa que es un método de clase.
abstract:	Sólo se proporciona la signatura del método. Sólo es posible en clases abstractas y no puede coincidir con los modificadores ***final***, ***static***, ***private.***

acceso: es igual que para los atributos

tipo:	el tipo de retorno del método. Puede ser cualquier tipo primitivo, clase, array o **void**
args:	lista de 0 o más argumentos separados por comas, donde cada argumento tiene la forma: ***tipo identificador***

ATRIBUTOS Y MÉTODOS

- **Métodos:**

Ejemplos:

```
public static void main(String args[]) {
        ...
}

public long[] getSueldos() {
        ...
}

public double potencia(double base, double exponente) {
        ...
}

abstract public void hacerAlgo();
```


Universidad de Alcalá
Departamento Ciencias de la Computación

ATRIBUTOS Y MÉTODOS

- **Métodos:**

• Todo método debe finalizar con una sentencia **return** (excepto los métodos de tipo **void**) seguida de una expresión del mismo tipo que el valor de retorno.

• Los métodos estáticos sólo pueden acceder a miembros estáticos.

• Podemos tener métodos con el mismo nombre pero diferente número y/o tipo de los argumentos. En ese caso el compilador escoge el método a invocar en función del número y tipo de los argumentos suministrados. Esto se conoce como **sobrecarga** de métodos.

```
X x = new X();
x.producto(2, 3);
x.producto(2.0, 3.5);
```

```
class X {
        public int producto(int a, int b) {
                return a * b;
        }

        public double producto(double a, double b) {
                return a * b;
        }...
```

EJEMPLO CLASE

```
public class Persona {

    //Atributos (privados)
    private String dni;
    private String nombre;
    private int edad;
    private String estado;

    //Constructor
    public Persona(String dni, String nombre,
                   int edad, String estado) {
        this.dni = dni;
        this.nombre = nombre;
        this.edad = edad;
        this.estado = estado;
    }
```

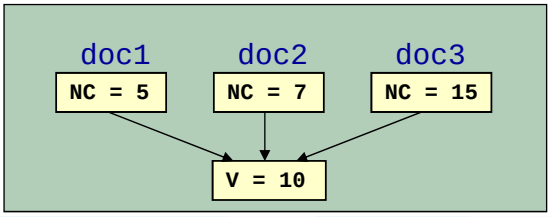
```
//Métodos (públicos)
public String getDni() {
    return dni;
}
public String getNombre() {
    return nombre;
}
public int getEdad() {
    return edad;
}
public void setEdad(int edad) {
    this.edad = edad;
}
public void cumpleaños() {
    edad++;
}
public String getEstado() {
    return estado;
}
public void setEstado(String estado) {
    this.estado = estado;
}
@Override
public String toString() {
    return "Persona{" + "dni=" + dni
 + ", nombre=" + nombre + ", edad=" + edad
 + ", estado=" + estado + '}';
} }
```


ATRIBUTOS Y MÉTODOS ESTÁTICOS

- Si queremos crear una clase en la que el valor de un atributo sea el mismo para todos los objetos instanciados a partir de esa clase debemos utilizar la palabra clave **static**. Para manejar este tipo de atributos necesitamos métodos estáticos.

```
public class Documento {
    private static int version;
    private int numero_de_capitulos;

    public Documento(int num_cap) {
        numero_de_capitulos = num_cap;
    }

    public void añade_un_capitulo() {
        numero_de_capitulos++;
    }

    public static void modifica_version(int i) {
        version = i;
    }

    public static int getVersion() {
        return Documento.version;
    }

    public int getNumCaps() {
        return this.numero_de_capitulos;
    }
}
```

```
public class PruebaDocumentos {

    public static void main(String args[]) {
        //Creamos documentos
        Documento doc1 = new Documento(5);
        Documento doc2 = new Documento(7);
        Documento doc3 = new Documento(15);

        //Aplicamos método de objeto
        doc1.añade_un_capitulo();
        System.out.println("Numero capítulos: "
                        + doc1.getNumCaps());
        //Aplicamos método de clase
        Documento.modifica_version(10);
        System.out.println("Versión: "
                        + Documento.getVersion());
    }
}
```

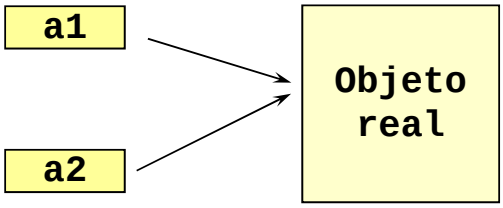


CREACIÓN DE OBJETOS

• Las referencias a objetos tienen valor **null** por defecto. A diferencia de otros lenguajes, la declaración de una variable (referencia) de una cierta clase, **no implica la creación de un objeto asociado**. La vinculación entre una referencia y un objeto se hace después mediante el operador **new** o asignando el valor de una referencia a otra del mismo tipo.

Cualquier objeto ha de ser creado explícitamente mediante el operador **new**.

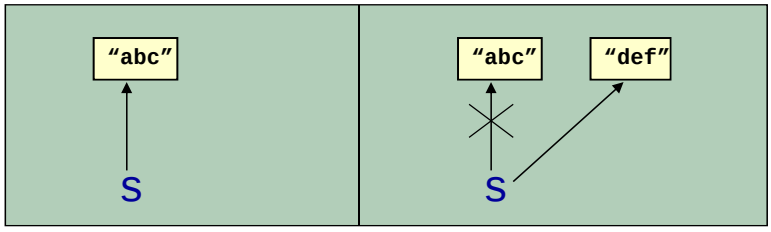
```
…
A a1, a2;
…
a1 = new A();
a2 = a1;
```



CREACIÓN DE OBJETOS

- Los objetos tienen un tiempo de vida y consumen recursos durante el mismo. Cuando un objeto no se va a utilizar más, debería liberar el espacio que ocupaba en la memoria de forma que las aplicaciones no la agoten.
- En Java, la recolección y liberación de memoria es responsabilidad de un thread llamado automatic garbage collector (recolector automático de basura). Este thread monitoriza el alcance de los objetos y marca los objetos que se han salido de alcance. Veamos un ejemplo:

```
String s;                  // no se ha asignado memoria todavía
s = new String( "abc" );   // memoria asignada
s = "def";                 // se ha asignado nueva memoria
                           // (nuevo objeto)
```



• La **Herencia** es el mecanismo por el que se crean nuevos objetos definidos en término de objetos ya existentes.

• Para heredar de una clase utilizamos la palabra clave **extends**:

```
class Empleado extends Persona {
        . . .
}
```

La clase Empleado hereda todos los atributos y métodos (la interfaz y la implementación) de Persona. A partir de aquí, podemos:

- **Extender** la clase Persona añadiendo nuevos métodos a la clase Empleado. Aumentamos la interfaz.
- **Anular** métodos de Persona con la propia implementación de Empleado.

• En Java sólo es posible extender de una clase (***herencia simple***).


```
public class MiClase {
        int i;
        public MiClase() {
            i = 10;
        }
        public void suma_a_i( int j ) {
            i = i + j;
        }
}
```

```
public class MiNuevaClase extends MiClase {
        public void suma_a_i( int j ) {
            i = i + ( j/2 );
            }
}
```

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.suma_a_i( 10 );
```

- Ahora cuando se crea una instancia de MiNuevaClase, el valor de i también se inicializa a 10 porque los constructores también se heredan.
- Pero la llamada al método suma_a_i() produce un resultado diferente.

- **This:**

- La referencia **this** se usa para referirse al propio objeto y se usa explícitamente para referirse tanto a las variables de instancia como a los métodos de un objeto.
- Dentro de los métodos no estáticos, podemos utilizar el identificador especial **this** para referirnos a la instancia que está recibiendo el mensaje.
- Algunos usos habituales son:
  - Evitar conflictos de nombres con los argumentos de métodos.

```
class Persona {
        String nombre;
        public void setNombre(String nombre) {
                this.nombre = nombre;
        …
```

  - Pasar una referencia de ese objeto a otro método.

```
Button incrementar = new Button("Incrementar");
incrementar.addActionListener(this);
```


- **Super:**

- Super se usa para referirse a métodos de la clase padre.

```
import MiClase;
  public class MiNuevaClase extends MiClase {
        public void suma_a_i( int j ) {
         i = i + ( j/2 );
        }
        public void suma_a_i_padre( int j ) {
         super.suma_a_i( j )
        }
  }
```

- Cuando extendemos una clase, los constructores de la subclase deben invocar algún constructor de la superclase como primera sentencia.

```
public Circulo(double radio) {
        super();
        this.radio = radio;
        }
```

- Si no se invoca explícitamente un constructor de la superclase con super, debe existir un constructor sin parámetros en la superclase o no debe haber ningún constructor (el compilador genera uno por defecto sin parámetros).

ASOCIACIÓN

- Una asociación es una relación estructural que describe una conexión entre objetos. La idea es que los objetos se unen para trabajar juntos.
- En Java se implementan las asociaciones mediante la inserción en una clase de un atributo que representa a otra clase (la clase asociada).
- A continuación veremos un ejemplo práctico donde usaremos herencia y asociación. Tenemos una clase Empleado que hereda de la clase Persona definida anteriormente y se asocia con la clase Departamento donde trabaja.

EJEMPLO HERENCIA Y ASOCIACIÓN


```
public class Departamento {
    // Atributos
    // Identificador del departamento
    private String id;
    // Nombre del departamento
    private String nombre;
    // Localización del departamento
    private String localizacion;

    //Constructor
    public Departamento(String id,
          String nombre,
          String localizacion) {
        this.id = id;
        this.nombre = nombre;
        this.localizacion = localizacion;
    }

    //Métodos
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getLocalizacion() {
        return localizacion;
    }
    public void setLocalizacion(String localizacion) {
        this.localizacion = localizacion;
    }
    @Override
    public String toString() {
        return "Departamento{" + "id=" + id
          + ", nombre=" + nombre
          + ", localizacion=" + localizacion + '}';
    }

}
```

EJEMPLO HERENCIA Y ASOCIACIÓN

```
public class Empleado extends Persona {

    //Atributos
    private String categoria;
    private double sueldo;
    //asociación
    private Departamento departamento;

    //Constructor
    public Empleado(String dni,
          String nombre,
          int edad,
          String estado,
          String categoria,
          double sueldo,
          Departamento departamento) {
        super(dni, nombre, edad, estado);
        this.categoria = categoria;
        this.sueldo = sueldo;
        this. departamento = departamento;
    }
```

```
    //Métodos
    public String getCategoria() {
        return categoria;
    }
    public void setCategoria(String categoria) {
        this.categoria = categoria;
    }
    public double getSueldo() {
        return sueldo;
    }
    public void setSueldo(double sueldo) {
        this.sueldo = sueldo;
    }
    public void subeSueldo(int cantidad) {
        sueldo += cantidad;
    }
    public Departamento getDepartamento() {
        return departamento;
    }
    public void setDepartamento(Departamento departamento) {
        this.departamento = departamento;
    }
    @Override
    public String toString() {
        super.toString() + "\nEmpleado{"
    + "categoria=" + categoria + ", sueldo=" + sueldo
    + ", departamento=" + departamento + '}';
    }
}
```


EJEMPLO HERENCIA Y ASOCIACIÓN

```
public class PruebaEmpleados {
    public static void main(String args[]) {
        //Creamos un departamento
        Departamento informatica = new Departamento("1Inf", "Informática", "Madrid");
        //Creamos empleados
        Empleado emp1 = new Empleado("153647458S", "Pepe", 35, "casado", "Analista", 1500, informatica);
        Empleado emp2 = new Empleado("452697419Z", "Maria", 25, "soltera", "Programadora", 1000, informatica);

        //Aplicamos métodos
        System.out.println(emp2.toString());
        emp2.subeSueldo(100);
        System.out.println("Sueldo: " + emp2.getSueldo());
    }
}
```

Ejecución:

Persona{dni=452697419Z, nombre=Maria, edad=25, estado=soltera}
Empleado{categoria=Programadora, sueldo=1000.0, departamento=Departamento{id=1Inf, nombre=Informática, localizacion=Madrid}}
Sueldo: 1100.0

CLASES ABSTRACTAS

- Cuando se organizan las clases en una jerarquía lo normal es que las clases que representan los conceptos más abstractos ocupen un lugar más alto en la misma.

- Los lenguajes orientados a objetos dan la posibilidad de declarar clases que definen como se utiliza pero sin tener que implementar los métodos que posee.

- En Java se realiza mediante clases abstractas que poseen métodos abstractos que implemantarán las clases hijas.

CLASES ABSTRACTAS

```
public class Punto {
        private int x;
        private int y;
        ...
}

public abstract class FiguraGeometrica {
        private Punto posicion;
        private String color;
        ...
        public void mover(Punto nuevaPos) {
                posicion = nuevaPos;
                dibujar();
        }
        abstract public void dibujar();
}

public final class Circulo extends FiguraGeometrica {

        public void dibujar() {
                . . .
        }
}
```

No se pueden crear instancias de las clases abstractas. Tienen la función de encapsular un concepto abstracto que compartirán sus subclases.
Parte del interfaz que definen, no posee una implementación.

No se puede extender (final). Suelen utilizarse cuando pueda ser "peligroso" permitir que las subclases dieran otra implementación de ciertos métodos.

- Constituyen un mecanismo para separar la interfaz de la implementación.
- Un interface contiene métodos y atributos constantes. Las clases que implementen el interface están obligadas a codificar todos los métodos definidos en el interface.

- Se declaran de forma similar a las clases:

```
interface Nombre {
   atributos
   métodos
}
```

Cuerpo del interfaz.
Sólo puede contener signaturas de métodos (no la implementación o cuerpo) y atributos

- Los atributos de un interfaz son **implícitamente public, static y final**, y deben ser inicializados en la declaración.

- Los métodos de un interfaz son **implícitamente public, abstract y no pueden ser static**.

- **Ejemplo:**

```
public interface VideoClip {
        int version = 1; //atributo constante
        void play();  //comienza la reproducción del video
        void bucle(); //reproduce el video en un bucle
        void stop();  //detiene la reproducción
}

public class DivX implements VideoClip {
        public void play() {
             <código>
        }
        public void bucle() {
             <código>
        }
        public void stop() {
             <código>
        }
}
```

PAQUETES

• Un **package** es un mecanismo para organizar clases e interfaces relacionadas. La propia librería estándar de Java está organizada en forma de paquetes. Además, todas las clases de un paquete tienen acceso ***friendly, por defecto*** unas a otras.

• La sentencia package, si existe, debe ir al principio del fichero fuente y debe ser la primera sentencia de éste:

package ***identificador[.identificador] ...***;

• Todas las clases declaradas en esa unidad de compilación pertenecerán al mismo paquete.

Ejemplo:

package Matematicas.Estadistica;

• Por defecto todas las clases que se encuentren en un mismo directorio pertenecen al mismo paquete.

PAQUETES

• El nombre completo de una clase es la suma del nombre del paquete más el nombre de la clase. Así, para declarar una referencia de la clase ***Varianza*** del paquete ***Matematicas.Estadistica*** utilizaríamos la sentencia:

```
Matematicas.Estadistica.Varianza var;
```

• Para evitar tener que utilizar nombres tan largos, podemos utilizar la sentencia **import**:

```
import Matematicas.Estadistica.Varianza;
```

o bien

```
// todas las clases del paquete
import Matematicas.Estadistica.*;
```

Nota: Cuando se utiliza este tipo de importación solo se importan las clases del paquete Estadística no los subpaquetes.

• De ese modo podemos declarar:

```
Varianza var;
```