Programación. 1º DAW

Unidad 4: Desarrollo de clases y utilización de objetos

Índice

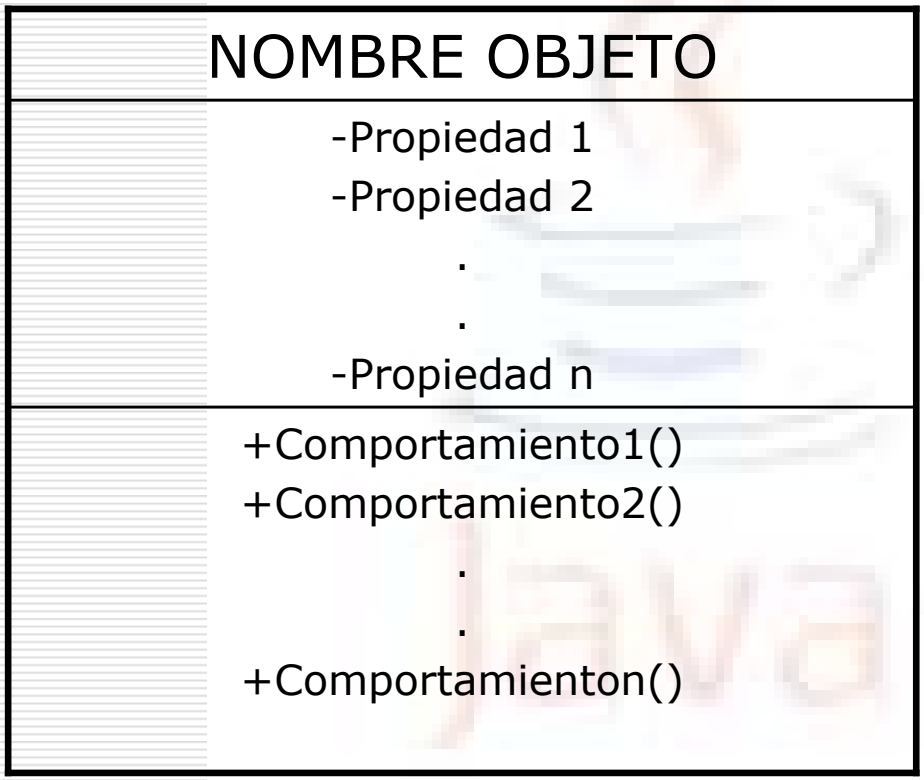
Introducción a la POO. Características Básicas

- 1. Todo es un Objeto
  - Según el DRAE:
    - Objeto es sinónimo de cosa "Todo lo que tiene entidad, ya sea corporal o espiritual, natural o artificial, real o abstracta"
  - Objetos: Silla, viento, luz, persona, boligrafo…
  - Todo Objeto o cosa se caracteriza por:
    - Sus propiedades: color, tamaño, forma, estado,…
    - Su comportamiento o posibles usos
      - Silla→Sirve para sentarse
      - Bolígrafo→ Sirve para escribir

Introducción a la POO. Características Básicas

1. Todo es un Objeto

- Notación UML



Datos: Int, char, float, objetos,…

Funciones: Instrucciones que representan el comportamiento del objeto y que pueden operar con los datos que representan las propiedades

Introducción a la POO. Características Básicas

- 2. Todo Objeto es de algún tipo
  - El comportamiento esperado de un objeto es lo que nos determina su tipo o clase.
  - Podemos afirmar que todo objeto es de algún tipo o alguna **CLASE**

CLASE	OBJETO (Instancia de una clase)
▪No tiene existencia. ▪Es algo abstracto	▪Existen Objetos concretos ▪Dos objetos de una misma clase pueden tener propiedades distintas ▪Dos objetos de una misma clase tienen los mismos comportamientos, que definen su clase ▪Todo objeto es de una clase

Introducción a la POO. Características Básicas

- 2. Todo Objeto es de algún tipo
  - Una **CLASE** es la abstracción de las propiedades y de los comportamientos de un conjunto de objetos.
  - Ejemplo de clase y objeto:

Clase Mechero
longitud: real > 0 (en cm) peso: real > 0 (en gr) gasRestante: real ≥ 0 (en ml) encendido: lógico
encender( ) apagar( ) rellenarGas (cantidad: real)

≠

Objeto Mechero Bic
longitud = 5.0 peso = 10.3 gasRestante = 3.4 encendido = verdadero
encender( ) apagar( ) rellenarGas (cantidad: real)

Introducción a la POO. Características Básicas

- 2. Todo Objeto es de algún tipo
  - **La clase es el PLANO o el MOLDE con el que se construyen los objetos. Es la "fábrica" de los objetos.**
  - El segundo principio decía: ***todo objeto es de algún tipo,*** es decir **debe ser creado a partir de una clase o tipo.**
  - A los objetos que se crean a partir de una clase se les llama **instancias de la clase.** En el ejemplo anterior: el objeto '*mechero bic*' **es una instancia de** la clase Mechero.

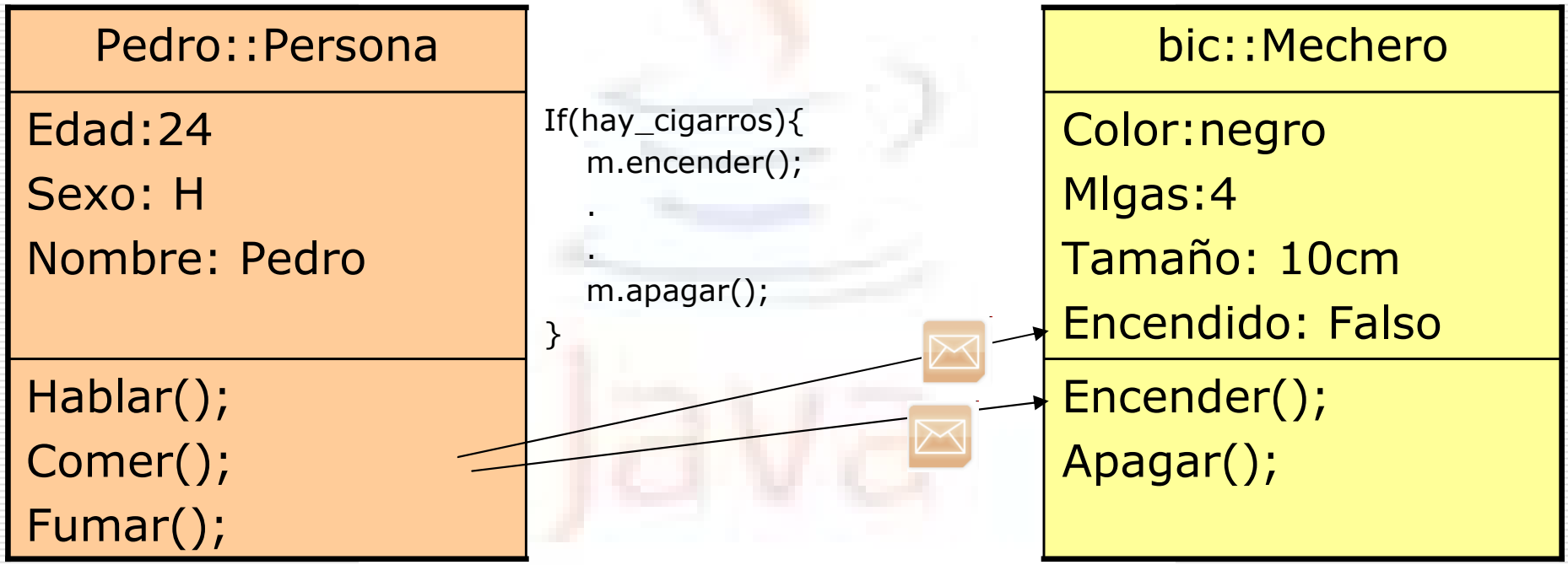
Introducción a la POO. Características Básicas

- 2. Todo Objeto es de algún tipo
  - Hay que distinguir claramente ambos conceptos:

Clases	Objetos
No tienen existencia real, son sólo el molde de creación de los objetos	**Tienen existencia** real y unas propiedades con valores concretos
Son elementos **estáticos**, no evolucionan en el tiempo	Son elementos **dinámicos**, su estado evoluciona durante la marcha del programa
De **una clase** se pueden crear **muchos objetos** (instancias)	**Un objeto** sólo puede ser creado a partir de **una clase**

Introducción a la POO. Características Básicas

- 3. Todo Objeto se comunica con otros objetos mediante mensajes
  - Mensaje: Petición que un objeto realiza a otro para que este ejecute alguna de sus operaciones que define su comportamiento.



Introducción a la POO. Características Básicas

- 4. Un Programa Orientado a Objetos es una **comunidad** de objetos que:
  - Nacen o se CREAN (Constructor)

```
MiClase C = new MiClase(arg1,arg2,…,arg3);
```

  - Se comunican entre ellos o se MANDAN MENSAJES

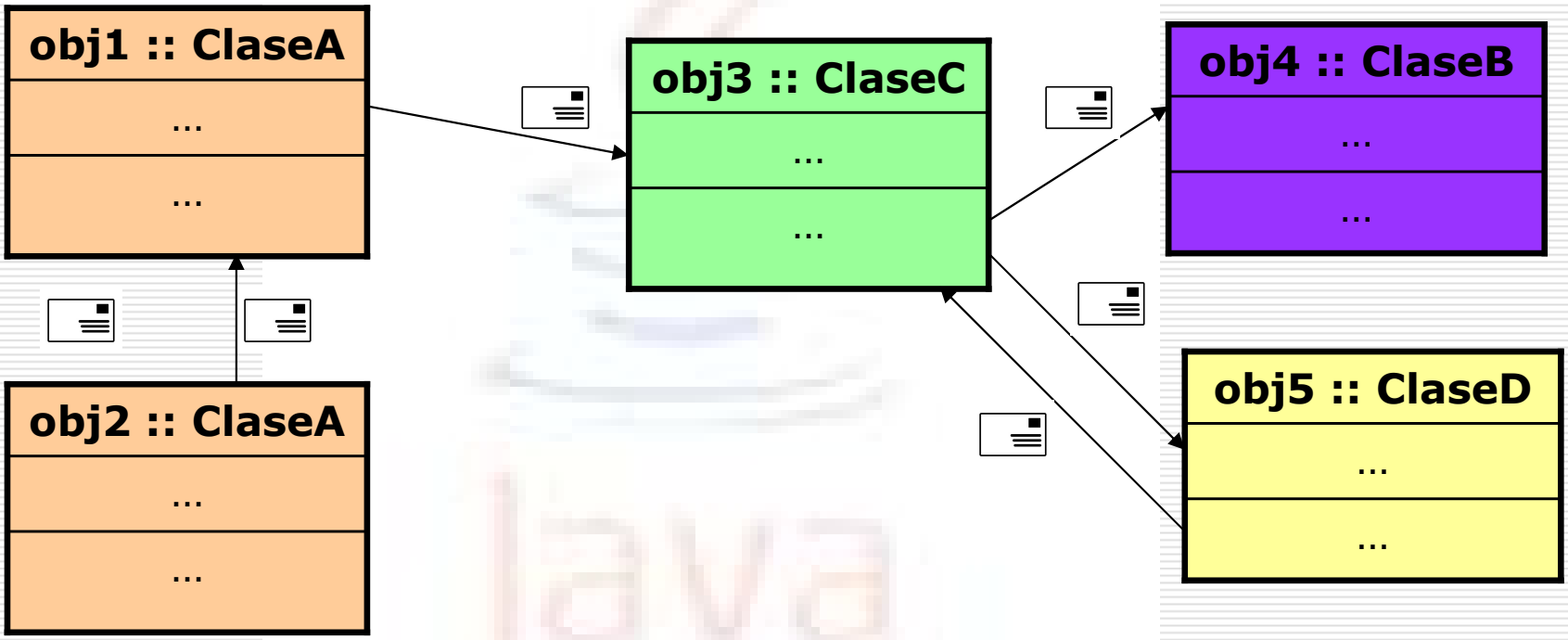
```
void setNombre(String nuevoNombre)
{
        nombre=nuevoNombre;
}
public String getNombre()
{
        return nombre;
}
```

  - Y Mueren o se DESTRUYEN (Finalize)

```
protected void finalize() {
        // Liberación del recurso no Java o recurso compartido
}
```

Introducción a la POO. Características Básicas

- 4. Un Programa Orientado a Objetos es una **comunidad** de objetos



Introducción a la POO. Principios de la POO

- Principio de Abstracción
  - Suprimir u ocultar algunos detalles de un proceso o de un elemento, para resaltar algunos aspectos, detalles o estructuras
- Principio de Encapsulamiento
  - Permite al programador de objetos que información dar a conocer y ocultar al resto de objetos.
- Principio de Polimorfismo
  - Capacidad que tienen objetos de diferentes clases de responder al mismo mensaje, es decir, que puede haber muchos mensajes con el mismo nombre, en diferente s clases

Introducción a la POO. Principios de la POO

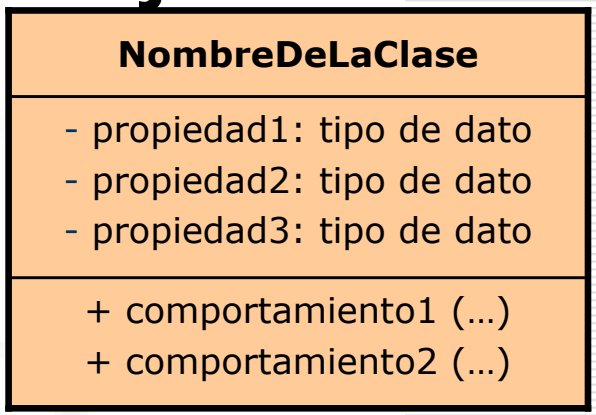
- Principio de la jerarquia
  - Las clases dentro de un programa también se presentan ordenadas en jerarquías, formándose los árboles de herencia
- Principio de modularidad
  - Permite dividir un programa complejo en varios módulos o partes diferentes. Cada uno resolverá una parte de un problema grande, y después interactuarán todos los módulos, como si se armara un rompecabezas. Se basa en la frase "divide y vencerás
- Principio de paso de mensajes
  - Son el medio a través del cuál interactúan los objetos
  - Con el paso de mensajes los objetos pueden solicitar a otros objetos que realicen alguna acción o que modifiquen sus atributos.
  - Junto con el paso de mensajes se implementan llamadas a los métodos de un objeto, que nos permite que ese objeto realice una acción que está dentro de su comportamiento.

Introducción a la POO. Notación

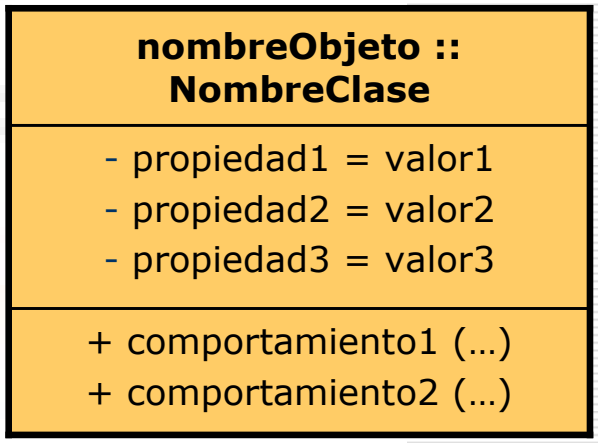
- **Diagrama de clases y de objetos en UML**
  - Para denotar nuestras clases, objetos y sus interacciones usaremos la notación UML.
    - UML (unified modeling language = lenguaje de modelado unificado) es el lenguaje diagramático más usado para representar sistemas orientados a objetos
  - Empezaremos con lo básico y ampliaremos la notación a medida que nos haga falta

Introducción a la POO. Notación

- **Diagrama de clases y de objetos en UML**
  - **Representación de una clase en UML:**



  - **Representación de un objeto en UML:**



Conceptos de Orientación a Objetos CLASE

- Una clase es una agrupación de ***datos*** (variables o campos) y de ***funciones*** (métodos) que operan sobre esos datos. La definición de una clase se realiza en la siguiente forma:

```
[public] class Classname {
    // Declaración de las propiedades
    // Declaración de los comportamientos
}
```

  - donde la palabra ***public*** es opcional: si no se pone, la clase es sólo es visible para las demás clases del ***package***.

Conceptos de Orientación a Objetos
Características de una CLASE

- En un fichero se pueden definir varias clases, pero **solo una** clase ***public***.
  - Este fichero se debe llamar como la clase ***public*** que contiene con extensión ****.java***.
  - Lo habitual es escribir una sola clase por fichero.
  - Si una clase contenida en un fichero no es ***public***, no es necesario que el fichero se llame como la clase.

- Los métodos de una clase pueden referirse de modo global al ***objeto*** de esa clase al que seaplican por medio de la referencia ***this***.

Conceptos de Orientación a Objetos
Tipos de CLASES

- **abstract**
  - Una clase *abstract* tiene al menos un método abstracto.
  - No es instanciable.
  - 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, 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.
- **synchronizable**
  - Todos los métodos definidos en la clase son sincronizados:
    - no se puede acceder al mismo tiempo a ellos desde distintos threads
    - el sistema se encarga de colocar los flags necesarios para evitarlo.
    - Este mecanismo hace que desde threads diferentes se puedan modificar las mismas variables sin que haya problemas de que se sobreescriban.

Conceptos de Orientación a Objetos
Definición de una CLASE

- Definición de una CLASE
  - Declaración de una clase

```
public class NombreClase {
    // Declaración de las propiedades
    // Declaración de los comportamientos
}
```

  - Por el momento asumiremos que todos los elementos que declaremos serán **públicos.**

  - **Estilo de nombrado:** el nombre de la clase empieza por mayúsculas, sigue en minúsculas y se capitaliza el comienzo de cada palabra.

    Ej: Persona, VentanaTipoA, ImpresoraLaser…

Conceptos de Orientación a Objetos
Creación de atributos

- Declaración de las propiedades
  - Es el conjunto de características atribuibles a la clase que se define.
  - Al igual que existen variables:
    - Definidas como parámetros a una subrutina
    - O definidas dentro de una subrutina, localmente.
  - Las propiedades se declaran como variables que pertenecen a la clase, que la constituyen, por eso reciben el nombre de **variables miembro** o **campos.**

Conceptos de Orientación a Objetos
Creación de atributos

- Ejemplo:

```
public class Empleado {
        // Declaración e inicialización de propiedades
        public String nombre = null;
        public String apellido1, apellido2, cargo;
        public double salario = 1000;
        public Date fechaAlta, fechaBaja, fechaNacim;
        public boolean estaDeBaja = false;

        // Comportamientos
        //…
}
```

Conceptos de Orientación a Objetos
Métodos

Declaración de los comportamientos

- Los **métodos** son los encargados de definir los comportamientos de una clase.
- Los métodos se declaran igual que las subrutinas en C, así que hay que concretar:
  - Un nombre
  - De 0 a N parámetros de entrada
  - De 0 a 1 parámetro de salida
- **Estilo de nombrado**: tanto las propiedades como los métodos se nombran en minúsculas y capitalizando el comienzo de cada palabra excepto la primera.
- Ejemplos *(relativos a la clase Empleado)*:
  - **public void** cambioDeCargo (String nuevoCargo) {…}
  - **public** String obtenerCargo ( ) {…}

Java

Conceptos de Orientación a Objetos
Métodos

- La diferencia entre los métodos y las subrutinas de C es que los **métodos pueden leer y escribir las propiedades** o variables miembros de la clase
- Ejemplos:
  - **public void** cambioDeCargo (String *nuevoCargo*) {
    cargo = *nuevoCargo*;
    }
  - **public** String obtenerCargo ( ) {
    return cargo;
    }

Si no hay parámetros de entrada **no se pone void**

Conceptos de Orientación a Objetos
Constructores

- Constructores
  - Un ***constructor*** es un método que se llama automáticamente cada vez que se crea un objeto de una clase.
  - El ***constructor*** reserva memoria e inicializa las variables miembro de la clase.
  - Los ***constructores*** no tienen valor de retorno (ni siquiera ***void***)
  - Su ***nombre*** es el mismo que el de la clase.
  - Su ***argumento implícito*** es el objeto que se está creando.
  - Una clase tiene ***varios constructores***, que se diferencian por el tipo y número de sus argumentos (son un ejemplo típico de métodos ***sobrecargados***).
  - Se llama ***constructor por defecto*** al constructor que no tiene argumentos.

Conceptos de Orientación a Objetos
Constructores

- Constructores
  - Un ***constructor*** de una clase puede llamar a ***otro constructor previamente definido*** en la misma clase por medio de la palabra ***this***.
    - La palabra ***this*** sólo puede aparecer en la ***primera sentencia*** de un ***constructor***.
  - El ***constructor*** de una ***sub-clase*** puede llamar al constructor de su ***super-clase*** por medio de la palabra ***super***, seguida de los argumentos apropiados entre paréntesis.
  - En caso de no existir el ***constructor*** de una clase, el ***compilador*** crea un ***constructor por defecto***, inicializando las variables de los tipos primitivos a su valor por defecto, y los ***Strings*** y las demás ***referencias*** a objetos a ***null***.
    - Si hace falta, se llama al ***constructor*** de la ***super-clase*** para que inicialice las variables heredadas.

Conceptos de Orientación a Objetos
Constructores

- Se declaran de la siguiente forma:
  - **Constructor definido**

    public NombreDeLaClase (lista parámetros) {…}
    - Pueden ser mas de uno
    - No puede ser declarado como static, final, abstract o synchronized
    - Por regla general se declaran públicos
  - **Constructor por defecto**:

    public NombreDeLaClase () {…}
    - Constructor público sin parámetros ni código, cuando no se especifica en el código
    - Se ejecuta siempre de manera automática e inicializa el objeto con los valores predeterminados
    - Es necesario incluirlo en el código si existe algún constructor definido.

Conceptos de Orientación a Objetos
Ejemplo de CLASE

- Un ejemplo sencillo, la clase Bombilla:

```
public class Bombilla {
      // Tres propiedades
      public boolean encendida = false;
      public int potencia = 100;
      public int numEncendidos = 0;
      //Constructor
      public Bombilla ( ) {
            encendida = true;
            potencia =50;
      }
      // Dos comportamientos
      public void encender ( ) {
            encendida = true;
            numEncendidos++;
            System.out.println ("Bombilla de "+potencia+" vatios encendida");
      }
      public void apagar ( ) {
            encendida = false;
            System.out.println ("Bombilla de "+potencia+" vatios apagada");
      }
}
```

Bombilla
+ potencia: int + numEncendidos: int + encendida: boolean
+ void encender ( ) + void apagar ( )

Conceptos de Orientación a Objetos
Objeto. Instanciación.

- OBJETO
  - Un ***objeto*** (en inglés, ***instance***) es un ejemplar concreto de una clase.
    - Las ***clases*** son como tipos de variables
    - Los ***objetos*** son como variables concretas de un tipo determinado (una clase).
  - Un objeto se instancia así

    Classname unObjeto;

    Classname otroObjeto;

Conceptos de Orientación a Objetos
Creación y destrucción de un Objeto

- Para crear un objeto nuevo en Java se antepone la palabra **new** a un *constructor* de la clase del objeto que se desea crear.
  - Por ejemplo, para crear un punto con coordenadas (2,3) se escribiria:

    punto p; // declara que p es una variable de tipo *punto*

    p=new punto(2,3); // crea el nuevo objeto p
  - Los objetos de una clase se llaman también **instancias** de la clase.
- Toda clase tiene al menos un constuctor.
  - Los constructores pueden tener ninguno, uno o varios parámetros.
  - Si en una clase no se define explícitamente un constructor, entonces tiene sólo un constructor sin parámetros.

Conceptos de Orientación a Objetos
Creación y destrucción de un Objeto

- Al crear un objeto se reserva espacio de memoria para contener las variables del objeto.

- Cuando un objeto deja de usarse en un programa porque ya no se hace referencia a él, entonces el espacio de memoria apartado para sus variables queda disponible para que el controlador de memoria RAM de Java lo utilice nuevamente.

- En Java el programador no tiene que preocuparse de destruir los objetos para liberar memoria

  System.gc() //libera la memoria.

Conceptos de Orientación a Objetos

- OBJETO. Propiedades y Métodos
  - **Propiedad o atributo**:
    - Características predeterminadas de un objeto o tipo de dato asociado a un objeto
    - Su valor puede ser alterado por la ejecución de algún método
  - **Método**:
    - Algoritmo asociado a un objeto (o a una clase de objetos)
    - Su ejecución se desencadena tras la recepción de un "mensaje".
    - Es lo que el objeto puede hacer.
      - Puede producir un cambio en las propiedades del objeto
      - Generar un "evento" con un nuevo mensaje para otro objeto del sistema.

Conceptos de Orientación a Objetos
La creación de Objetos

- **Los constructores de objetos**
  - Son unos métodos especiales que permiten la creación de objetos de la clase que los contiene. Deben garantizar la inicialización de las propiedades del objeto a valores coherentes.
  - Se declaran de la siguiente forma:
    - **public** NombreDeLaClase (lista parámetros) {…}
  - En cuanto a la sintaxis:
    - Se deben llamar exactamente igual que la clase
    - No hay que escribir ningún tipo de retorno

Conceptos de Orientación a Objetos
La creación de Objetos

Bombilla
+ potencia: int + numEncendidos: int + encendida: boolean
+ *Bombilla* (int potIni, int numIni) + void encender ( ) + void apagar ( )

- Ejemplo:

```
public class Bombilla {
    // Propiedades
    public int potencia;
    public int numEncendidos;
    public boolean encendida, fundida;

    // Método constructor
    public Bombilla (int potenciaInicial, int numEncendidosInicial ) {
        encendida = false;
        fundida = false;
        potencia = potenciaInicial;
        if (potencia <= 0)
            potencia = 20;
        numEncendidos = numEncendidosInicial;
        if (numEncendidos < 0)
            numEncendidos = 0;
    }
    // Comportamientos…
```

Conceptos de Orientación a Objetos
La creación de Objetos

- Cuando escribimos un constructor debemos saber que el compilador añade al código compilado un conjunto de instrucciones antes de las nuestras.
- Estas instrucciones generadas automáticamente se encargan de la inicialización de las propiedades del objeto de la siguiente forma:
  - Si la propiedad declarada no se inicializa:
    - Los números y char se ponen a **cero**
    - Los lógicos se inicializan a **false**
    - Las referencias a objetos apuntan a **null**
  - Si la propiedad declarada se inicializa entonces toma dicho valor inicial

Conceptos de Orientación a Objetos
La creación de Objetos

- Ejemplo de uso:

```
public class Bombilla {
    // Propiedades
    public int potencia = 20;
    public int numEncendidos;
    public boolean encendida, fundida;

    // Método constructor
    public Bombilla (int potenciaInicial, int numEncendidosInicial ) {
        potencia = 20;
        numEncendidos = 0;
        encendida = false;
        fundida = false;
        if (potenciaInicial > 0)
            potencia = potenciaInicial;
        if (numEncendidosInicial > 0)
            numEncendidos = numEncendidosInicial;
    }
    // Comportamientos…
```

Instrucciones que se añaden al código compilado de forma automática

Instrucciones escritas por el programador/a

Conceptos de Orientación a Objetos
La creación de Objetos

- Si no escribimos un constructor para una clase el compilador genera el **constructor por defecto** (lo añade al código compilado no al código fuente).

- En nuestro ejemplo anterior se generaría:
  - **public** Bombilla ( ) {}

  que realizaría las siguientes inicializaciones:

  potencia = 20; numEncendidos = 0; encendida = **false**; fundida = **false**;

Conceptos de Orientación a Objetos
La creación de Objetos

- **La instanciación y utilización de objetos:**
  - Para crear o **instanciar** un objeto a partir de una clase:
    - Necesitamos una **referencia** capaz de apuntar al objeto a crear.
    - Asignar a dicha referencia el resultado de la creación del objeto, que se realiza con el operador **new** y un **constructor** de la clase.

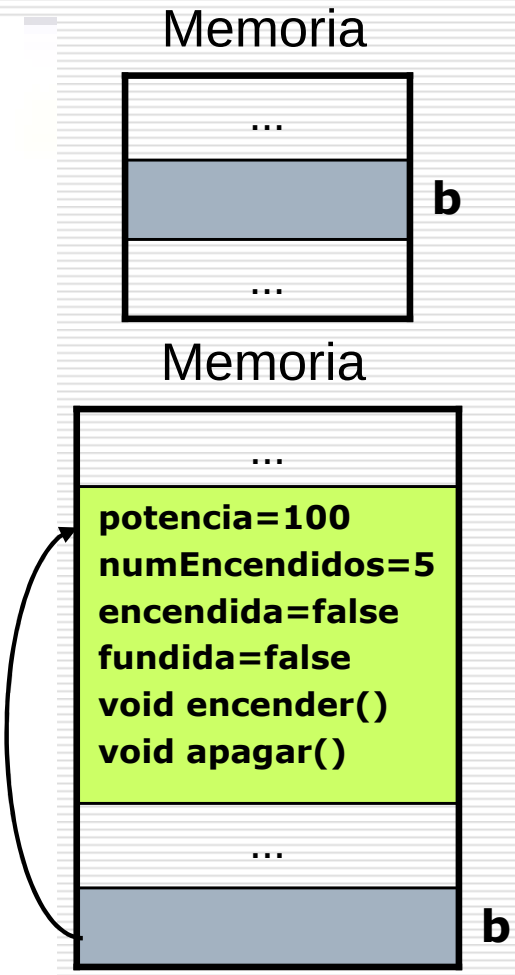
Conceptos de Orientación a Objetos
La creación de Objetos

- Ejemplo:
  - Bombilla b;

    /* b es una referencia a objetos de tipo Bombilla */
  - b = **new** Bombilla (100, 5);

    /* A la referencia b le asignamos el objeto resultante de llamar

    al constructor. Hacemos que apunte al objeto */
- También lo podemos hacer todo en una sola línea:
  - Bombilla b = **new** Bombilla (100, 5);



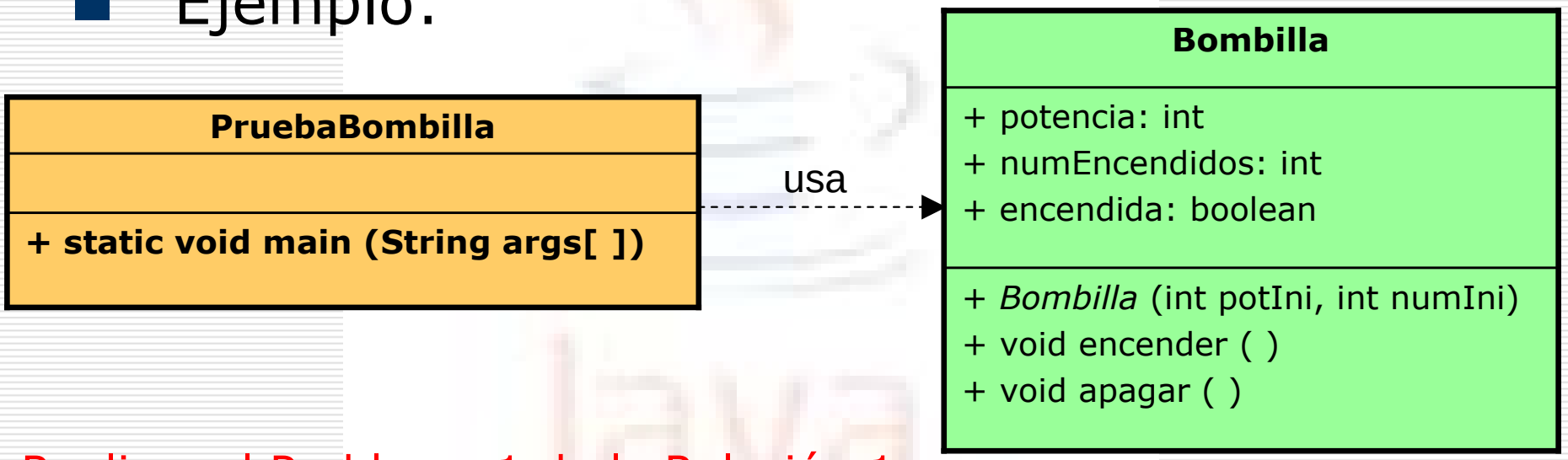
Conceptos de Orientación a Objetos
La creación de Objetos

- Para **utilizar** un objeto:
  - Necesitamos una **referencia** que apunte al objeto que queremos utilizar.
  - Utilizar el **operador punto** '**.**' para acceder a sus miembros (propiedades y métodos).
  - Ejemplo:

```
Bombilla b1 = new Bombilla (100, 5);
Bombilla b2 = new Bombilla (60, 0);
if (b1.fundida == false)
    b1.encender();
b2.encender();
b1.apagar();
```

Conceptos de Orientación a Objetos
La creación de Objetos

- Para probar nuestras clases y manejar objetos creados a partir de ellas crearemos una clase de pruebas que sólo contenga el método main.
- Ejemplo:



Realizar el Problema 1 de la Relación 1

Conceptos de Orientación a Objetos
La creación de Objetos

- Un posible código de *PruebaBombilla*:

```
public class PruebaBombilla {
    // No hay propiedades ni constructores en la clase

    public static void main(String args[]) {
        int i;
        Bombilla b = new Bombilla(); // Creamos un objeto bombilla

        // La encendemos y apagamos 1000 veces
        for (i=1; i<=1000; i++)
        {
            b.encender();
            b.apagar();
        }
        b.encender();  // Y otra vez más, ya debe estar fundida
        b.apagar();
    }
}
```

Conceptos de Orientación a Objetos
Características de los Objetos

- Tienen un **espacio de memoria propio.**
- Como consecuencia de esto, cada objeto tiene una **identidad propia** y **conserva su estado interno**.
- Independientemente de cómo sean manipulados, los objetos deben tener siempre en un **estado interno coherente**. Esto dependerá cómo se haya escrito la clase (ésta debe ser *responsable*).

Conceptos de Orientación a Objetos
Metodos de Objetos

- Los ***métodos*** son funciones definidas dentro de una clase.
- Salvo los métodos ***static*** o de clase, se aplican siempre a un objeto de la clase por medio del ***operador punto*** (.). Dicho objeto es su ***argumento implícito***.
- La primera línea de la definición de un método se llama ***declaración*** o ***header***
- El código comprendido entre las ***llaves*** {…} es el ***cuerpo*** o ***body*** del método.
  - EJEMPLO:

```
public Circulo elMayor(Circulo c) { // header y comienzo del método
        if (this.r>=c.r) // body
        return this; // body
        else // body
        return c; // body
    } // final del método
```

- El ***header*** consta de:
  - El cualificador de acceso
  - El tipo del valor de retorno (***void*** si no tiene)
  - El ***nombre de la función***
  - Una lista de ***argumentos explícitos*** entre paréntesis, separados por comas.
    - Si no hay argumentos explícitos se dejan los paréntesis vacíos.

Conceptos de Orientación a Objetos
Métodos de Objetos

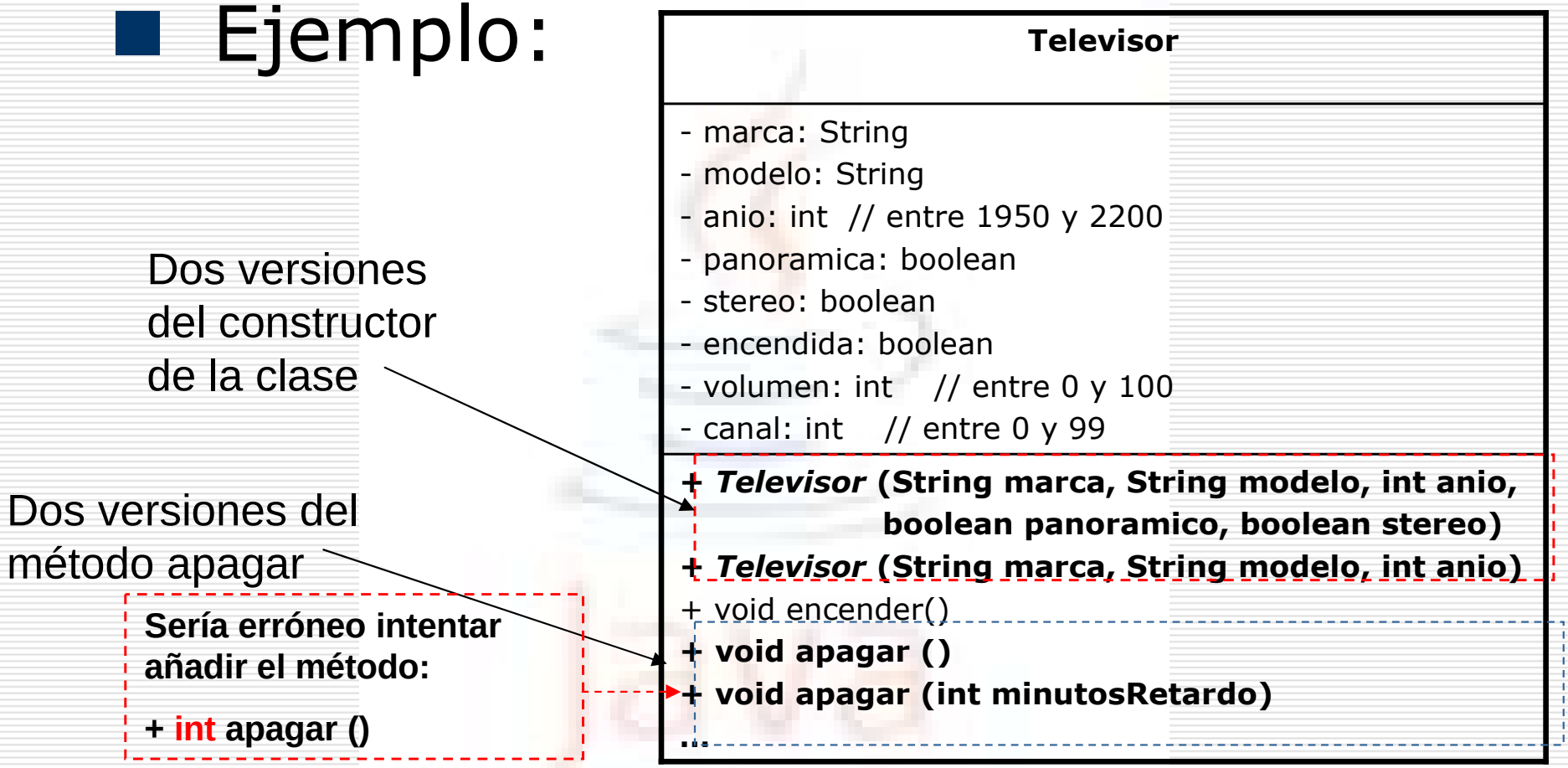
- Los métodos tienen ***visibilidad directa*** de las variables miembro del objeto
  - También se puede acceder a ellas mediante la referencia ***this***.
- El ***valor de retorno*** puede ser un valor de un ***tipo primitivo*** o una ***referencia a un objeto***.
  - No puede haber más que un único valor de retorno (que puede ser un objeto o un array).
- Se puede devolver como valor de retorno un objeto de la misma clase que el método o de una sub-clase, pero nunca de una super-clase.
- Los métodos pueden definir ***variables locales***.
  - Su visibilidad llega desde la definición al final del bloque en el que han sido definidas.
  - No hace falta inicializar las variables locales cuando se definen, pero el compilador no permite utilizarlas sin haberles dado un valor.

Objetos
Métodos sobrecargados (overloaded)

- Java permite que una clase tenga varias versiones de un mismo método o constructor.
- De esta forma evitamos "memorizar" demasiados nombres de métodos.
- Las **reglas** a seguir para **sobrecargar** métodos o constructores son las siguientes:
  - Los métodos tiene el mismo nombre y tipo de retorno.
  - Los métodos se distinguen por el número y tipo de los parámetros de entrada.
  - La visibilidad (private, public…) de los métodos no sirve para distinguir métodos entre sí.

Conceptos de Orientación a Objetos
Métodos sobrecargados

- Ejemplo:



Conceptos de Orientación a Objetos
Paso de argumentos a Métodos

- En ***Java*** los argumentos de los ***tipos primitivos*** se pasan siempre ***por valor***.
  - El método recibe una copia del argumento actual; si se modifica esta copia, el argumento original que se incluyó en la llamada no queda modificado.
- La forma de modificar dentro de un método una variable de un tipo primitivo es incluirla como variable miembro en una clase y pasar como argumento una referencia a un objeto de dicha clase. Las ***referencias*** se pasan también ***por valor.***
- En ***Java*** no se pueden pasar métodos como argumentos a otros métodos (en C/C++ se pueden pasar punteros a función como argumentos).
  - Lo que se puede hacer en ***Java*** es pasar una referencia a un objeto y dentro de la función utilizar los métodos de ese objeto.
- Dentro de un método se pueden crear ***variables locales*** de los tipos primitivos o referencias que dejan de existir al terminar la ejecución del método
- Los argumentos formales de un método (variables del ***header*** del método para recibir el valor de los argumentos actuales) tienen categoría de variables locales del método.

Conceptos de Orientación a Objetos
Paso de argumentos a Métodos

- Si un método devuelve ***this*** (es decir, un objeto de la clase), puede encadenarse con otra llamada a otro método de la misma o de diferente clase y así sucesivamente.
  - Los diferentes métodos aparecen en la misma sentencia unidos por el operador punto (.), por ejemplo,

    String numeroComoString = "8.978";

    float p = Float.valueOf(numeroComoString).floatValue();
    - Donde:
      - El método ***valueOf(String)*** de la clase ***java.lang.Float*** devuelve un objeto de la clase ***Float***
      - Sobre el anterior se aplica el método ***floatValue()***, que finalmente devuelve una variable primitiva de tipo ***float***.
- El ejemplo anterior es equivalente a:

  String numeroComoString = "8.978";

  Float f = Float.valueOf(numeroComoString);

  float p = f.floatValue();
  - El operador (.) en se ejecuta de izquierda a derecha

Conceptos de Orientación a Objetos
Métodos de Clase (static)

- Existen métodos que no actúan sobre objetos concretos a través del operador punto. A estos métodos se les llama ***métodos de clase*** o ***static***.
- Los métodos de clase pueden recibir objetos de su clase como argumentos explícitos, pero no tienen argumento implícito ni pueden utilizar la referencia ***this***.
- Un ejemplo típico de métodos ***static*** son los métodos matemáticos de la clase ***java.lang.Math*** (*sin()*, *cos()*, *exp()*, *pow()*, etc.).
- Los métodos y variables de clase se crean anteponiendo la palabra ***static***.
- Para llamarlos se suele utilizar el nombre de la clase, en vez del nombre de un objeto de la clase (por ejemplo, ***Math.sin(ang)***, para calcular el seno de un ángulo).

Conceptos de Orientación a Objetos
Métodos de Clase (static)

- El modificador **static aplicado a un método** (no constructores) hace que dicho método sea considerado como "de clase", pudiendo ser llamado sin crear ningún objeto de la clase.
- La sintaxis de la llamada sería:
  - NombreClase.nombreMétodoEstático (…);
- Si retocamos el ejemplo anterior añadiendo **static** al método *getConsumoTotalBombillas*:

```
public static int getConsumoTotalBombillas() {
    return consumoTotal;
}
```

Conceptos de Orientación a Objetos
Métodos de Clase (static)

- Como un método estático puede **ser llamado desde la clase**, sin que existan objetos, cuando escribamos el código de un método estático sólo podremos utilizar propiedades estáticas y llamar a métodos estáticos.
  - En nuestro ejemplo de las bombillas, el método estático *getConsumoTotalBombillas* no puede acceder a: encendida, numEncendidos, potencia…
- Aunque el modificador **static** parece poco útil sin embargo se utiliza mucho.

Conceptos de Orientación a Objetos

- Ahora podríamos ejecutar el siguiente código:

```
System.out.print ("Sin bombillas el consumo es ");
System.out.println ( Bombilla.getConsumoTotalBombillas()+"W" );
Bombilla b1 = new Bombilla (100, 5);
Bombilla b2 = new Bombilla (60, 1);
b1.encender();
b2.encender();
System.out.print ("Con dos bombillas el consumo es ");
System.out.println ( Bombilla.getConsumoTotalBombillas() +"W");
// Aunque también podríamos sustituir la última instrucción por
// System.out.println ( b1.getConsumoTotalBombillas()+"W");
```

Ocultación y Encapsulación

- Modificadores de Acceso
  - Permiten al diseñador de una clase determinar quien accede a los datos y métodos miembros de una clase.
  - Preceden a la declaración de un elemento de la clase (ya sea dato o método), de la siguiente forma:

    [*modificadores*] *tipo_variable nombre*;

    [*modificadores*] *tipo_devuelto nombre_Metodo (lista_Argumentos)*;
  - Existen los siguientes modificadores de acceso:
    - **public**: Todo el mundo puede acceder al elemento.
    - **private:** Sólo se puede acceder al elemento desde métodos de la clase, o sólo puede invocarse el método desde otro método de la clase.
    - **protected**: Esta involucradado con la herencia.
      - Un miembro designado como *protected* aparece como *public* para los miembros de clases derivadas de la clase y aparece como *private* para todas las demás.
    - sin modificador: Se puede acceder al elemento desde cualquier clase del package donde se define la clase.

Ocultación y Encapsulación

- Si hemos dicho que los objetos deben tener un estado coherente y nos esforzamos escribiendo un código que intenta garantizarlo, ¿qué ocurre si hacemos lo siguiente?

```
Bombilla b = new Bombilla(100, 10);
for (i=1; i<=1000; i++)
{
    b.encender();
    b.apagar();
}
if (b.fundida == true)
    b.fundida = false;
```

Consultamos y **cambiamos** una propiedad que define el estado del objeto saltándonos todos los controles

Ocultación y Encapsulación

- No queremos que se pueda realizar ese tipo de manipulación externa del objeto, para ello debemos **ocultar las propiedades** de las clases utilizando el modificador de visibilidad **private.**

```
public class Bombilla {
    // Propiedades OCULTAS
    private int potencia;
    private int numEncendidos;
    private boolean encendida, fundida;
     // …
```

- Ya no podríamos hacer: b.fundida = **false;**
- Pero tampoco podríamos consultar el valor de ninguna propiedad: **if** (b.numEncendidos < 100)

Ocultación y Encapsulación

- Para solucionar este problema hacemos lo siguiente:
  1. Todas las propiedades se declaran como **private**
  2. Para las propiedades cuyo valor quiero que pueda ser **consultado** creo un método público que devuelva el valor:

```
public int obtenerNumEncendidos ( ) {
    return numEncendidos;
}
```

  3. Para aquellas propiedades cuyo valor quiero que pueda ser **modificado** creo un método público que reciba el nuevo valor y haga las comprobaciones pertinentes:

```
public void establecerPotencia (int nuevaPotencia) {
    if (nuevaPotencia > 0)
        potencia = nuevaPotencia;
}
```

Ocultación y Encapsulación

- De esta forma se dice que las propiedades están **ocultas** y que su acceso externo está **encapsulado** en uno o más métodos.
- Todos los objetos deben de cumplir este doble principio de **ocultación** y **encapsulamiento** si se quiere garantizar su buen funcionamiento.
- En inglés:
  - Los métodos de consulta reciben el nombre de **getters** (obtenedores) ya que se nombran como: *getNombrePropiedad*
  - Y los de modificación reciben el nombre de **setters** (establecedores) ya que son de la forma: *setNombrePropiedad*
- NetBeans es capaz de **generar automáticamente** los getters y setters de las propiedades de una clase
  - Basta hacer click con el botón derecho sobre el editor, seleccionar insertar codigo→Agregar propiedad

Ocultación y Encapsulación

- El objeto habla en primera persona: **la referencia *this***
  - ¿Qué ocurre si escribimos el siguiente código? ¿Cómo distinguimos la propiedad *potencia* del parámetro *potencia*?

```
public void establecerPotencia (int potencia)
{
    if (potencia > 0)
        potencia = potencia;
}
```

  - Podemos cambiar el nombre del parámetro *potencia* para que sea distinto del nombre de la propiedad pero también podemos usar la referencia **this.**

Ocultación y Encapsulación

- **this** es una referencia al objeto actual, al objeto con el que estemos trabajando en ese momento. Desde esta referencia podemos acceder a todas las propiedades y métodos del objeto.
- Usando **this** en el caso anterior, quedaría:

```
public void establecerPotencia (int potencia) {
    if (potencia > 0)
        this.potencia = potencia;
}
```

- Se usa normalmente para resolver conflictos de nombre o aclarar a qué objeto pertenece una propiedad o un método.
- Se puede utilizar en métodos y constructores.

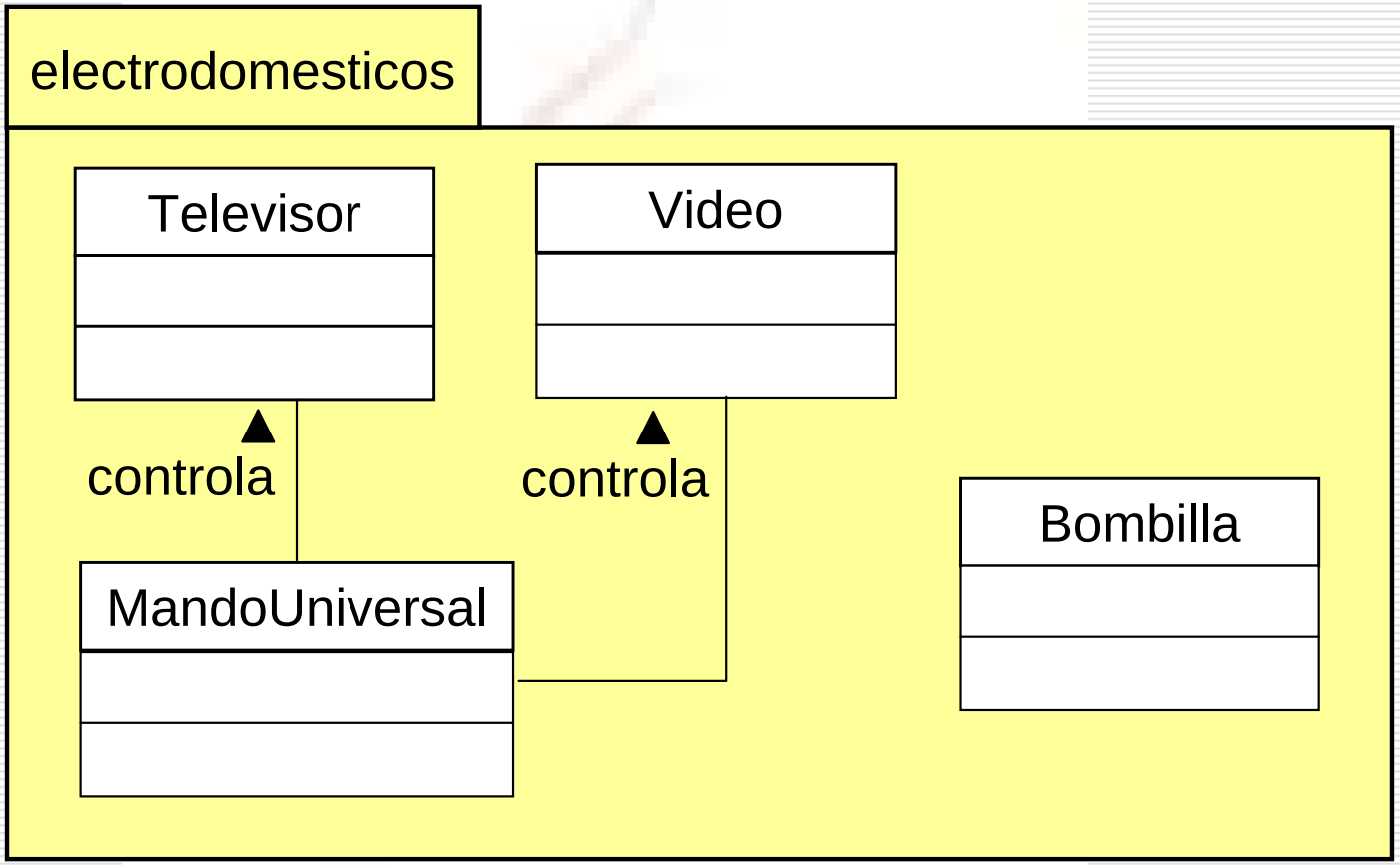
Packages

- **Que es un package**
  - Un ***package*** es una agrupación de clases.
  - Los nombres de los paquetes son palabras separadas por puntos y se almacenan en directorios que coinciden con esos nombres.
  - El usuario puede crear sus propios ***packages***.
  - Para que una clase pase a formar parte de un ***package*** llamado ***pkgName***, hay que introducir en ella la sentencia:

    package pkgName;
    - que debe ser la primera sentencia del fichero sin contar comentarios y líneas en blanco.
  - Los nombres de los ***packages*** se suelen escribir con minúsculas, para distinguirlos de las clases, que empiezan por mayúscula.
  - El nombre de un ***package*** puede constar de varios nombres unidos por puntos (por ejemplo ***java.awt.event***).
  - Todas las clases que forman parte de un ***package*** deben estar en el mismo directorio.

Packages

- Notación UML y ejemplo de un Paquete:



Packages

- **Declaración de un paquete**
  - Para indicar que una clase se incluye en un paquete sólo hay que poner en la primera línea de su código fuente la sentencia:

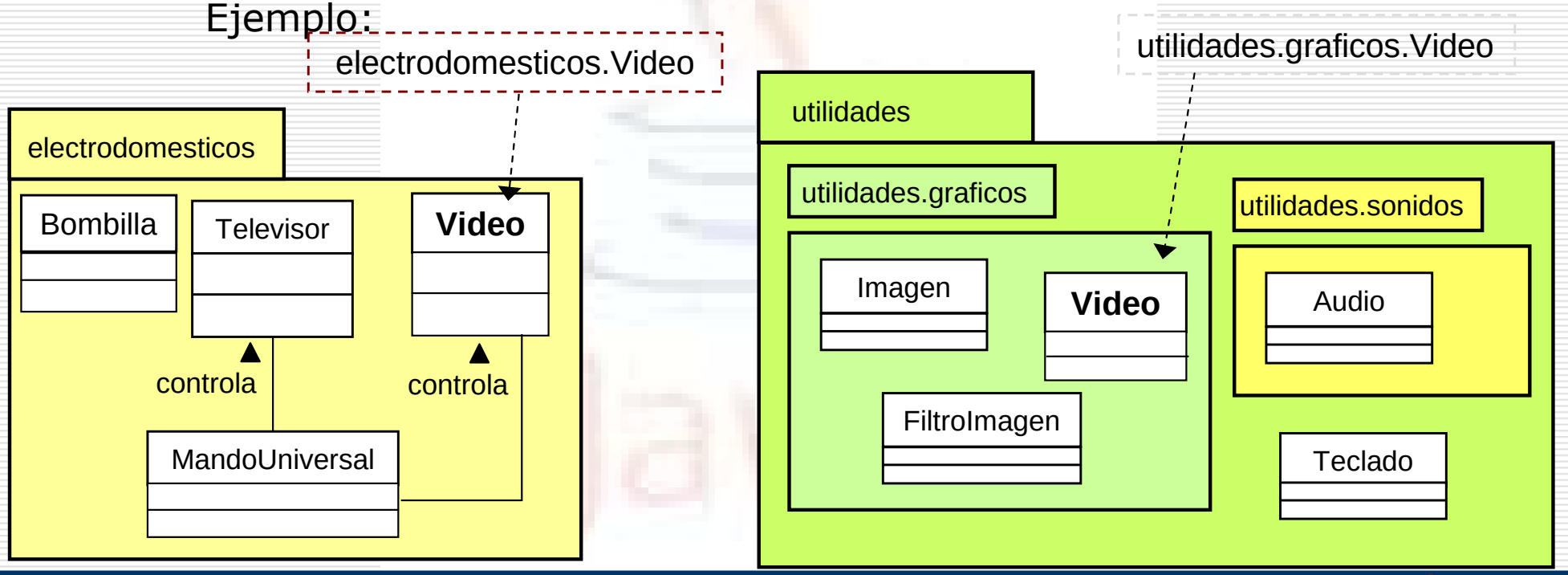
    package *nombredelpaquete*;
  - En el ejemplo anterior los ficheros Bombilla.java, Televisor.java, Video.java y MandoUniversal.java deben tener como primera línea:

    package electrodomesticos;
- **Reglas de nombrado de un paquete:**
  - Se escribe entero en minúsculas y sin espacios.
  - Debe comenzar con una letra o un '$' o un '_'
  - No puede ser una palabra reservada del lenguaje (**int**, **float**, **class**…)
  - Si un paquete está contenido en otro paquete, el nombre del paquete contenedor precede al nombre del contenido y se separan con un punto '**.**'
  - **Ejemplos**:

Nombres correctos	Nombres incorrectos
utilidades gestionclientes utilidades.graficos utilidades.sonidos	**12**meses **static** **G**estion**B**ajas futur**-**soft **double.**number

Packages

- **El espacio de nombres**
  - Un paquete define un espacio de nombres que actúa como **prefijo** del nombre de las clases que contiene.
  - Así dos clases se pueden llamar igual si pertenecen a paquetes distintos. Ejemplo:

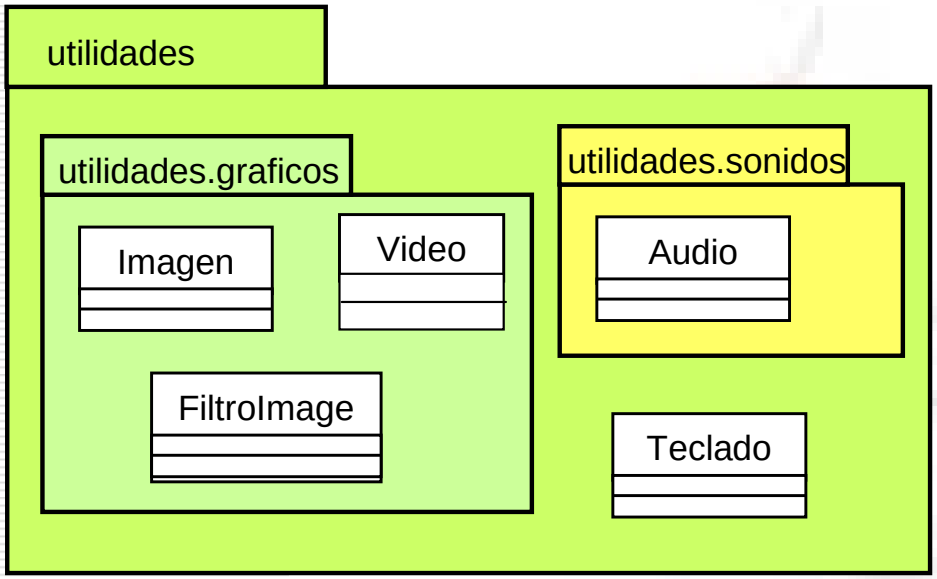


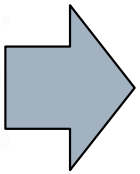
Packages

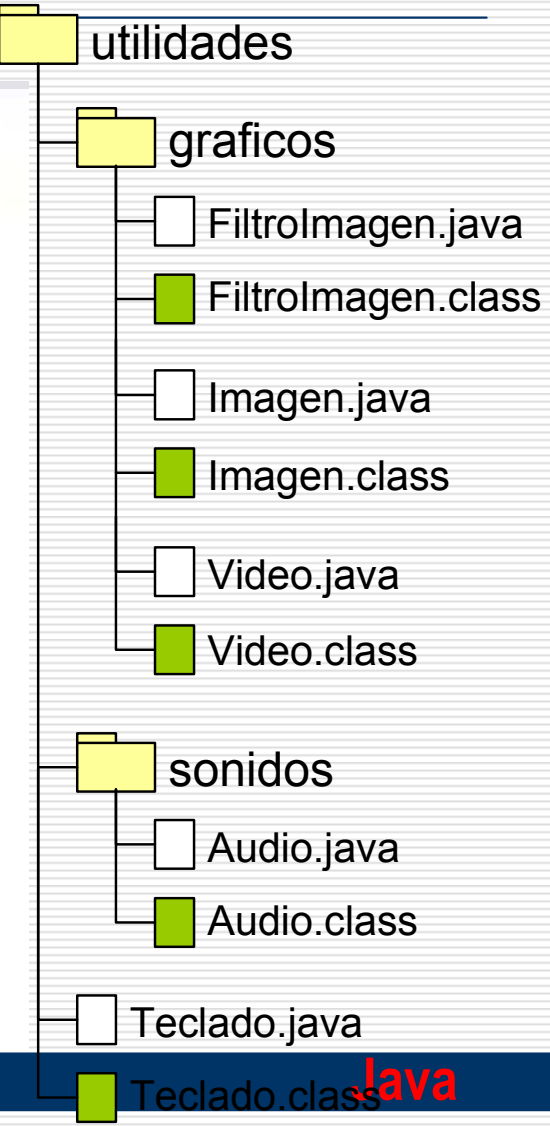
- **La estructura de carpetas asociada**
  - Los paquetes son contenedores de clases o de otros paquetes.
  - Por otro lado los sistemas operativos organizan sus ficheros en carpetas. De modo que una carpeta es un contenedor de ficheros o de otras carpetas.
  - Java establece una relación directa entre paquete-carpeta y clase-fichero.

Packages

- Ejemplo de Paquete:







Esta organización es la que el compilador espera, así que para cambiar una clase de paquete hay que:

1.- Cambiar la sentencia package y

2.- Mover el fichero a la carpeta que corresponda

Packages

- **Clases visibles dentro y fuera de un paquete**
  - Hasta ahora todas las clases que hemos escrito comenzaban con: **public** class … esto significa que la clase será visible/utilizable por las clases de dentro y de fuera del paquete
  - Sin embargo si una clase que pertenece a un paquete se declara **sin el modificador public,** sólo será visible por las clases que comparten el paquete con ella.
  - Esto permite escribir clases auxiliares o de apoyo a las clases visibles o públicas.

Packages

- **El modificador de visibilidad de paquete**
  - Los paquetes son contenedores de clases que guardan una relación entre ellas.
  - Si las clases de un paquete necesitan cooperar entre sí puede ser interesante que algunas propiedades o métodos tengan un **nivel de ocultación** intermedio entre **public** y **private**, de manera que se comparta información.
  - Este nivel de visibilidad recibe el nombre de **visibilidad de paquete** o **amigable** *(friendly)*.

Packages

- **Importar un Package**
  - Los paquetes de clases se cargan con la palabra clave ***import***, especificando el nombre del paquete como ruta y nombre de clase.
  - Se pueden cargar varias clases utilizando un asterisco.

```
import java.Date;
import java.awt.*;
```

  - Si un fichero fuente Java no contiene ningún *package*, se coloca en el paquete por defecto sin nombre
    - es decir, en el mismo directorio que el fichero fuente
    - Esta clase puede ser cargada con la sentencia *import* en otra clase java del mismo directorio
      - import MiClase;

Packages

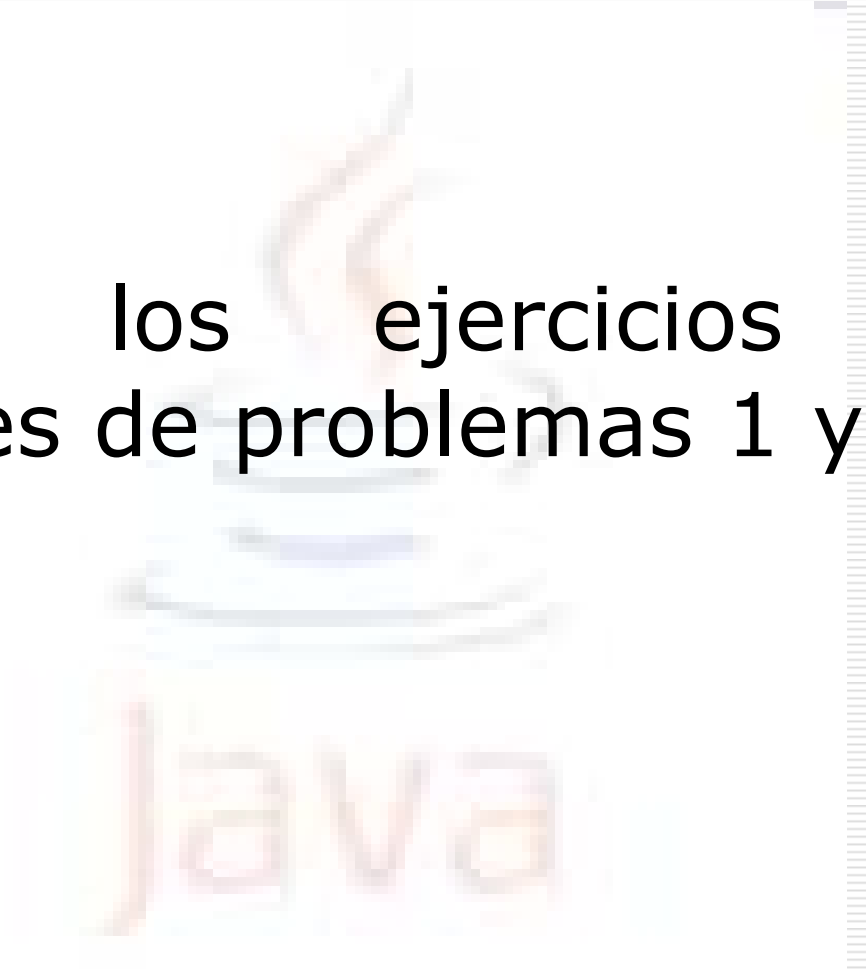
- **Importar un Package**
  - Cuando escribo una clase que necesita interactuar con una o más clases agrupadas en un paquete necesitamos escribir una **sentencia de importación** para que el compilador pueda encontrar dichos elementos.
  - Ejemplo:
    - **import** utilidades.Teclado; // Importa **sólo** la clase Teclado
    - **import** utilidades.graficos.*; // Importa **todas** las clases // **públicas** del paquete utilidades.graficos
  - Se pueden importar tantas clases como se desee.
  - Las sentencias **import** se escribe justo después de la sentencia **package** (si existe).
  - La palabra clave import puede colocarse al principio de un fichero, fuera del bloque de la clase.
    - El compilador reemplazará esa sentencia con el contenido del fichero que se indique, es decir, más clases

Packages

- Algunos Packages de JAVA
  - **java.lang**
    - Este paquete incluye las clases del lenguaje Java propiamente dicho: Object, Thread, Exception, System, Integer, Float, Math, String, etc.
  - **java.applet**
    - Este paquete contiene clases diseñadas para usar con applets.
    - Hay una clase Applet y tres interfaces: AppletContext, AppletStub y AudioClip.
  - **java.awt**
    - El paquete Abstract Windowing Toolkit (awt) contiene clases para generar widgets y componentes GUI (Interfaz Gráfico de Usuario).
    - Incluye las clases Button, Checkbox, Choice, Component, Graphics, Menu, Panel, TextArea y TextField.
  - **java.io**
    - El paquete de entrada/salida contiene las clases de acceso a ficheros: FileInputStream y FileOutputStream.
  - **java.net**
    - Este paquete da soporte a las conexiones del protocolo TCP/IP
    - Incluye las clases Socket, URL y URLConnection.
  - **java.util**
    - Este paquete es una miscelánea de clases útiles para muchas cosas en programación.
    - Se incluyen, entre otras, Date (fecha), Dictionary (diccionario), Random (números aleatorios) y Stack (pila FIFO).

Desarrollo de clases y utilización de Objetos

- Realizar los ejercicios de las relaciones de problemas 1 y 2



Copyright ©

Para la confección de parte de esta documentación se ha utilizado material con derechos reservados del Copyright del autor **Enrique José Royo Sánchez**, autorizando este su uso a **Antonio Blázquez Pérez** como material didáctico en el IES Polígono Sur de Sevilla