COMPOSICIÓN DE CLASES

El término "composición" tiene dos acepciones que generan confusión entre ellas:

- El significado riguroso que usa UML. En UML las clases tienen relaciones. Un tipo específico de relación entre clases es la *asociación*. Y dentro de la asociación cuando queremos relacionar "el todo con sus partes tenemos en uml " dos tipos de asociación: agregación(diamante blanco, la parte puede existir sin el todo) y composición(diamante negro, la parte no puede existir sin el todo)
- El significado más genérico. Nosotros de forma más informal entendemos por composición de clases el hecho de que una clase se defina en función de otras, esto es, que tenga un atributo(o varios) que sean variables referencias. Así que no distinguimos entre agregación y composición, le llamamos a todo composición.

EJEMPLO: la clase rectangulo utiliza la clase Punto para definir un atributo

```
class Punto {
   int x;
   int y;
   Punto(int x, int y) {
      this.x = x;
      this.y = y;
   }
}

class Rectangulo {
   Punto origen; //¡COMPOSICION DE CLASES!
   int ancho;
   int alto;

   Rectangulo(Punto p, int w, int h) {
      origen = p;
      ancho = w;
      alto = h;
   }

}
```

Principio de ocultación

Una "buena" práctica de programación es utilizar el principio de ocultación:

(de wikipedia)

https://es.wikipedia.org/wiki/Principio_de_ocultaci%C3%B3n

Principio de ocultación

Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una "interfaz" a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción.

Observa que puse en rojo "el quiz de la cuestión".

Realmente ya utilizamos en cierta medida el principio de ocultación ya que vimos que:

1. Se debe usar siempre que se pueda private para los atributos y acceder a ellos a través de set/get.
2. Los beneficios son:
    a. Evitar que otros programadores al acceder directamente a los atributos pongan en estado inconsistente un objeto (ej. Persona con edad negativa)
    b. Mayor resistencia a cambios (los set/get en base instrucciones pueden enmascarar cambios internos de la clase).

Esto es sobre todo importante cuando estamos escribiendo clases públicas, es decir, clases que usarán otros programadores.

Para hacer una buena ocultación de los atributos debemos proceder de forma un poco diferente si son de tipo primitivo o si son referencias

Ocultar atributos de tipos primitivos.

Como indicamos arriba, realmente la ocultación de tipos primitivos ya la discutimos cuando vimos los métodos set/get

Supongamos que un programador *X* escribió la clase *Rectangulo*, y tal y como él concibió el uso de su clase *Rectangulo* no tienen sentido las coordenadas negativas. Supongamos que el programador *X* colgó su clase en Internet y es descargada por un programador *Y* que está escribiendo la clase *Unidad5* que usa *Rectangulo*. El programador *Y,* no sabe que la clase Rectangulo no está preparada para trabajar con coordenadas negativas. Intenta introducir coordenadas negativas y si para ello utiliza setAncho() se impide la coordenada negativa pero si accede directamente al atributo ancho, el programador *Y* no se entera que hizo algo mal y que puede provocar que la clase Rectangulo empiece a hacer cosas raras.

```
class Rectangulo {
   int ancho;//no debe tener valores negativos por la razón que sea
   int largo;
   void setAncho(int ancho){
      if(ancho>0)
         this.ancho=ancho;
      else
         this.ancho=0;
   }
}

class Unidad5 {
   public static void main(String[] args) {
         Rectangulo r1 = new Rectangulo();
         r1.ancho=-2;//rectángulo inconsistente, el resto del programa puede funcionar mal
        System.out.println(r1.ancho);
         r1.setAncho(-2); //set evita ancho negativo
         System.out.println(r1.ancho);

   }

}
```

Conclusión:

¿Cómo ocultar miembros que son atributos primitivos?: Haciéndolos private y generando set/get para acceder indirectamente a ellos.

Ocultar atributos referencias

Cuando los atributos son referencias ¡mayor peligro todavía! Ya que aunque se definan como private pueden darse igualmente "efectos colaterales indeseados" si el código de los métodos set, get y constructores no es el adecuado.

Vamos a estudiar una serie de casos.

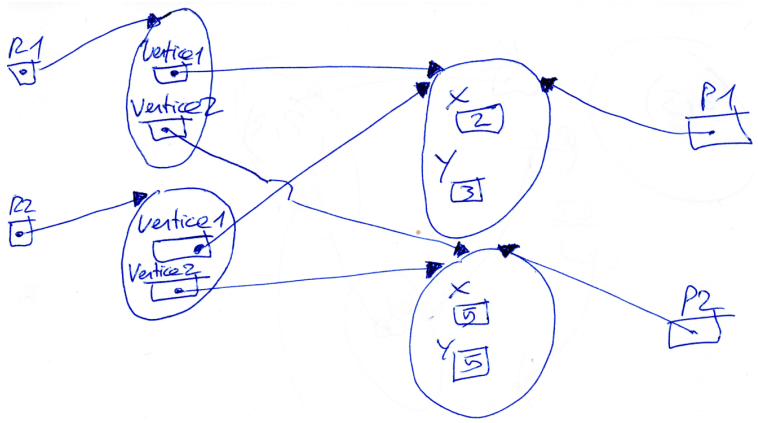
Problemas de ocultación de atributos referencia debido al funcionamiento del constructor

Estudia el siguiente ejemplo y haz un dibujito de objetos para observar que es inútil que los vértices sean private

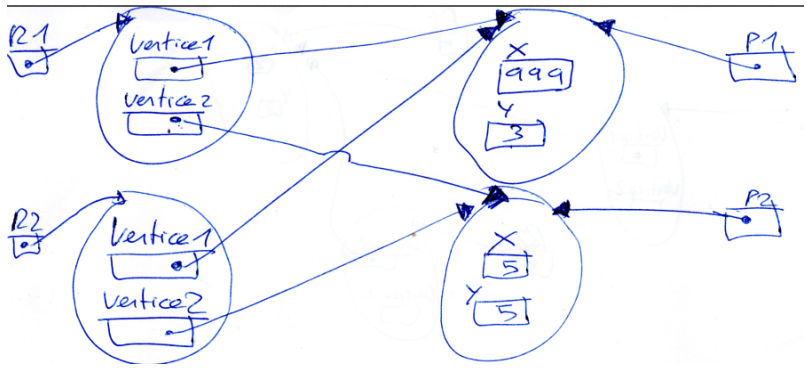
```
class Punto {
   private int x = 0;
   private int y = 0;
   Punto(int x, int y) {
        this.x = x;
        this.y = y;
   }
   public Punto() {
        x=0;
        y=0;
   }
   public Punto(Punto p){
      x=p.x;
      y=p.y;
   }
   public int getX() {
      return x;
   }
   public void setX(int x) {
      this.x = x;
   }
   public int getY() {
      return y;
   }
   public void setY(int y) {
      this.y = y;
   }

}

class Rectangulo {
  //hacer  private los vertices no es suficiente para ocultarlos
   private Punto vertice1;
   private Punto vertice2;


   //dos versiones de constructor  con la misma firma por eso una la comento.
   public Rectangulo(Punto vertice1, Punto vertice2){
      this.vertice1=vertice1;
      this.vertice2=vertice2;
   }

   /*
   //esta es la versión que correcta que cumple el principio de ocultación
   public Rectangulo(Punto vertice1, Punto vertice2){
      this.vertice1=new Punto(vertice1);
      this.vertice2=new Punto(vertice2);
   }
   */
```

```
    public String pasarAString(){
      String v1="vertice1: ("+ vertice1.getX()+", "+vertice1.getY()+")      ";
      String v2="vertice2: ("+ vertice2.getX()+", "+vertice2.getY()+")";
      return v1+v2;
   }
}

class Unidad5 {
   public static void main(String[] args) {
      Punto p1= new Punto(2,3);
      Punto p2= new Punto(5,5);
      Rectangulo r1 = new Rectangulo(p1,p2);
      Rectangulo r2 = new Rectangulo(p1,p2);
      System.out.println("p1: ("+p1.getX()+","+p1.getY()+")");
      System.out.println("r1: "+r1.pasarAString());
      System.out.println("r2: "+r2.pasarAString());
      System.out.println("REFLEXIÓN: que pasa si cambio p1.x a 999");
      p1.setX(999);
      System.out.println("p1: ("+p1.getX()+","+p1.getY()+")");
      System.out.println("r1: "+r1.pasarAString());
      System.out.println("r2: "+r2.pasarAString());
      System.out.println("¡peligro!: cambio el punto y los vertices de r1 y r2 aunque los vertices  sean private");

  }

}
```

Vamos a dibujar la situación anterior para constatar que hay un objeto punto referenciado desde tres referencias



Por lo tanto
p1.setX(999);
Provoca que también se cambie el vertice1 de r1 y r2



SOLUCIÓN: El código del constructor, para eliminar toda posible referencia externa debe crear para sus vértices nuevos objetos punto

Ejercicio: comprueba esto usando la versión de constructor comentada y observa como ahora los vértices de los rectángulos quedan inalterados

```
public Rectangulo(Punto vertice1, Punto vertice2){
       this.vertice1=new Punto(vertice1);
       this.vertice2=new Punto(vertice2);
   }
```

EJERCICIO U5_B2_E1:

Hacer dibujos similares a los anteriores pero con los matices del nuevo constructor empleado.

Problemas de ocultación de atributos referencia por mal funcionamiento de los métodos set

El problema analizado anteriormente en el constructor se extiende también a los set(). Observa el siguiente ejemplo.

```
class Punto {
   private int x = 0;
   private int y = 0;
   Punto(int x, int y) {
        this.x = x;
        this.y = y;
   }
   public Punto() {
        x=0;
        y=0;
   }
   public Punto(Punto p){
      x=p.x;
      y=p.y;
   }
   public int getX() {
      return x;
   }
   public void setX(int x) {
      this.x = x;
   }
   public int getY() {
      return y;
   }
   public void setY(int y) {
      this.y = y;
   }

}

class Rectangulo {
   private Punto vertice1;
   private Punto vertice2;



   public Rectangulo(Punto vertice1, Punto vertice2){
      this.vertice1=new Punto(vertice1);
      this.vertice2=new Punto(vertice2);
   }

   public void setVertice1(Punto vertice1){
      this.vertice1=vertice1;
   }


    public String pasarAString(){
      String v1="vertice1: ("+ vertice1.getX()+", "+vertice1.getY()+")      ";
      String v2="vertice2: ("+ vertice2.getX()+", "+vertice2.getY()+")";
      return v1+v2;
```

```
    }
}

class Unidad5 {
    public static void main(String[] args) {
        Punto p1= new Punto(2,3);
        Punto p2= new Punto(5,5);
        Rectangulo r1 = new Rectangulo(p1,p2);

        System.out.println("r1: "+r1.pasarAString());

        System.out.println("cambio vertice con setVertice1()");
        Punto p3=new Punto(44,44);
        r1.setVertice1(p3);
        System.out.println("r1 cambiado: "+r1.pasarAString());
        p3.setX(99);
        System.out.println("¡peligro!: al cambiar p3 se cambia el vertice1 de r1 aunque los vertice1  sea private");
        System.out.println("r1 inesperadamente cambiado: "+r1.pasarAString());


    }

}
```

SOLUCIÓN: similar al constructor. El set debe crear un nuevo punto independiente.

```
class Punto {
    private int x = 0;
    private int y = 0;
    Punto(int x, int y) {
         this.x = x;
         this.y = y;
    }
    public Punto() {
         x=0;
         y=0;
    }
    public Punto(Punto p){
        x=p.x;
        y=p.y;
    }
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }

}

class Rectangulo {
    private Punto vertice1;
    private Punto vertice2;




    public Rectangulo(Punto vertice1, Punto vertice2){
        this.vertice1=new Punto(vertice1);
        this.vertice2=new Punto(vertice2);
    }

    public void setVertice1(Punto vertice1){
        this.vertice1=new Punto(vertice1);
    }
```

```
    public String pasarAString(){
       String v1="vertice1: ("+ vertice1.getX()+", "+vertice1.getY()+")      ";
       String v2="vertice2: ("+ vertice2.getX()+", "+vertice2.getY()+")";
       return v1+v2;
    }
}

class Unidad5 {
   public static void main(String[] args) {
       Punto p1= new Punto(2,3);
       Punto p2= new Punto(5,5);
       Rectangulo r1 = new Rectangulo(p1,p2);

       System.out.println("r1: "+r1.pasarAString());

       System.out.println("cambio vertice con setVertice1()");
       Punto p3=new Punto(44,44);
       r1.setVertice1(p3);
       System.out.println("r1 cambiado: "+r1.pasarAString());
       p3.setX(99);
       System.out.println("al cambiar p3 ahora no se cambia el vertice1 de r1");
       System.out.println("r1 no es alterado por p3: "+r1.pasarAString());


  }

}
```

Problemas de ocultación de atributos referencia por mal funcionamiento de métodos get

Observa ahora que por el código inapropiado de getVertice1(), aunque vertice1 sea private puedo cambiarlo sin usar el set()

```
class Punto {
   private int x = 0;
   private int y = 0;
   Punto(int x, int y) {
        this.x = x;
        this.y = y;
   }
   public Punto() {
        x=0;
        y=0;
   }
   public Punto(Punto p){
       x=p.x;
       y=p.y;
   }
   public int getX() {
       return x;
   }
   public void setX(int x) {
       this.x = x;
   }
   public int getY() {
       return y;
   }
   public void setY(int y) {
       this.y = y;
   }

}

class Rectangulo {
   private Punto vertice1;
   private Punto vertice2;



   public Rectangulo(Punto vertice1, Punto vertice2){
```

```
        this.vertice1=new Punto(vertice1);
        this.vertice2=new Punto(vertice2);
    }

    public Punto getVertice1(){
        return this.vertice1;
    }


     public String pasarAString(){
        String v1="vertice1: ("+ vertice1.getX()+", "+vertice1.getY()+")      ";
        String v2="vertice2: ("+ vertice2.getX()+", "+vertice2.getY()+")";
        return v1+v2;
    }
}

class Unidad5 {
    public static void main(String[] args) {
        Punto p1= new Punto(2,3);
        Punto p2= new Punto(5,5);
        Rectangulo r1 = new Rectangulo(p1,p2);

        System.out.println("r1: "+r1.pasarAString());
        System.out.println("¡¡¡Peligro!! el get me devuelve una referencia con la que puedo cambiar vertice1");
        Punto p3=r1.getVertice1();
        p3.setX(99);p3.setY(99);
        System.out.println("r1 inesperadamente cambiado: "+r1.pasarAString());

   }

}
```

SOLUCIÓN: el get no debe devolver el objeto original, sólo una copia con los valores.

```
class Punto {
    private int x = 0;
    private int y = 0;
    Punto(int x, int y) {
          this.x = x;
          this.y = y;
    }
    public Punto() {
          x=0;
          y=0;
    }
    public Punto(Punto p){
        x=p.x;
        y=p.y;
    }
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }

}

class Rectangulo {
    private Punto vertice1;
    private Punto vertice2;
```

```
    public Rectangulo(Punto vertice1, Punto vertice2){
        this.vertice1=new Punto(vertice1);
        this.vertice2=new Punto(vertice2);
    }

    public Punto getVertice1(){
        return new Punto(this.vertice1);
    }


     public String pasarAString(){
        String v1="vertice1: ("+ vertice1.getX()+", "+vertice1.getY()+")      ";
        String v2="vertice2: ("+ vertice2.getX()+", "+vertice2.getY()+")";
        return v1+v2;
    }
}

class Unidad5 {
    public static void main(String[] args) {
        Punto p1= new Punto(2,3);
        Punto p2= new Punto(5,5);
        Rectangulo r1 = new Rectangulo(p1,p2);

        System.out.println("r1: "+r1.pasarAString());
        System.out.println("AHORA aunque cambie p3 no afecta a r1");
        Punto p3=r1.getVertice1();
        p3.setX(99);p3.setY(99);
        System.out.println("r1 ok: "+r1.pasarAString());

   }

}
```

El constructor copia

Ya que la clase Rectangulo para protegerse a menudo necesita crear copias de sí mismo una versión de constructor útil es

```
public Rectangulo(Rectangulo r){
        this.vertice1= new Punto(r.vertice1);
       this.vertice2=new Punto(r.vertice2);
}
```

A este tipo de versión de constructor, que crea un objeto copiando los valores de un objeto de su misma clase se le suele llamar constructor copia.

Recuerda que this puede acceder a los miembros private de cualquier objeto Rectangulo. Esto ya se discutió en otro boletín.

Podemos probarlo

```
class Punto {
   private int x = 0;
   private int y = 0;

   Punto(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public Punto() {
      x = 0;
      y = 0;
   }

   public Punto(Punto p) {
      x = p.x;
```

```
        y = p.y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

}

class Rectangulo {
    private Punto vertice1;
    private Punto vertice2;

    public Rectangulo(Rectangulo r) {
        this.vertice1 = new Punto(r.vertice1);
        this.vertice2 = new Punto(r.vertice2);
    }

    public Rectangulo(Punto vertice1, Punto vertice2) {
        this.vertice1 = new Punto(vertice1);
        this.vertice2 = new Punto(vertice2);
    }

    public String pasarAString() {
        String v1 = "vertice1: (" + vertice1.getX() + ", " + vertice1.getY() + ")      ";
        String v2 = "vertice2: (" + vertice2.getX() + ", " + vertice2.getY() + ")";
        return v1 + v2;
    }
}

public class Unidad5 {
    public static void main(String[] args) {
        Punto p1 = new Punto(2, 3);
        Punto p2 = new Punto(5, 5);
        Rectangulo r1 = new Rectangulo(p1, p2);
        Rectangulo r2 = new Rectangulo(r1);// invocamos a constructor copia
        System.out.println("r2: " + r2.pasarAString());
    }

}
```

Variables de clase static final y principio de ocultación

Si un atributo es static final no puede ser "mal utilizado" ya que es inmutable. No se puede cambiar mal algo que no se puede cambiar. Desde este punto de vista por tanto no pasa nada si lo declaro public. Interesa a veces calificarlo de *private* para excluirlo del javadoc y del autocompletar de los IDEs si consideramos que es un detalle interno de la clase que complica a sus usuarios su entendimiento.

Ya indicamos en el boletín de static que los atributos final suelen ser también static, pero pueden declararse también sólo con final las consideraciones son las mismas desde el punto de vista de principio de ocultación.

EJERCICIO U5_B2_E1: Genera código java de la clase Circulo de forma que cumpla el principio de ocultación. Un circulo de define con un punto origen P y un radio

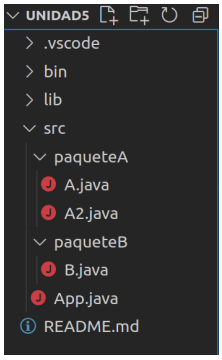
```
class punto{
   private int x;
   private int y;
   Punto(int x, int y) {
      this.x = x;
      this.y = y;
   }
   Punto() {
      x=0;
      y=0;
   }
   int getX() {
      return x;
   }

   void setX(int x) {
      this.x = x;
   }

   int getY() {
      return y;
   }

   void setY(int y) {
      this.y = y;
   }

}

class Unidad5{
   public static void main(String[] args) {
      Punto p=new Punto(4,5);
      Circulo c1= new Circulo(p,10);
      System.out.println(c1.circuloAString());
      p.setX(99);
      System.out.println("principio ocultación OK");
      System.out.println(c1.circuloAString());
   }
}
```

run:
4 5 10
principio ocultación OK
4 5 10

MODIFICADOR PUBLIC AL TRABAJAR CON VARIOS PAQUETES

No vamos a ver exhaustivamente el sistema de paquetes java, simplemente algunas pinceladas para trabajar desde el IDE con varios paquetes.

Creamos el siguiente proyecto que consta de 3 paquetes: paqueteA, paqueteB y paquete por defecto



El código de las cuatro clases es el siguiente

```
//App.java
import paqueteA.A;
public class App {
   public static void main(String[] args) throws Exception {
      A a= new A();
      a.metodoDeA();
   }
}


//B.java
package paqueteB;
import paqueteA.A;

public class B {
   A a;
   B(){
      a= new A();
   }
}


//A,java
package paqueteA;
public class A {
   public A(){
      System.out.println("Constructor de A ...");
   }
   public void metodoDeA(){
      System.out.println("metodoDeA...");
   }
}
//A2.java
package paqueteA;

public class A2 {
   A2(){
      A a= new A();
      a.metodoDeA();
   }
}
```

Analiza los siguientes puntos:

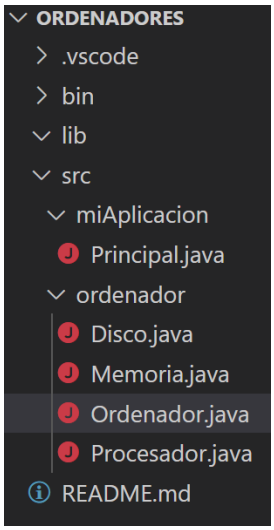
1. Porqué necesito en B y App usar import pero no lo necesito en A ni en A2
2. Que pasa si elimino el import en B o en App
3. Que pasa si no elimino los import pero retiro el public en A() o en metodoDeA()

CONCLUSIÓN BÁSICA:

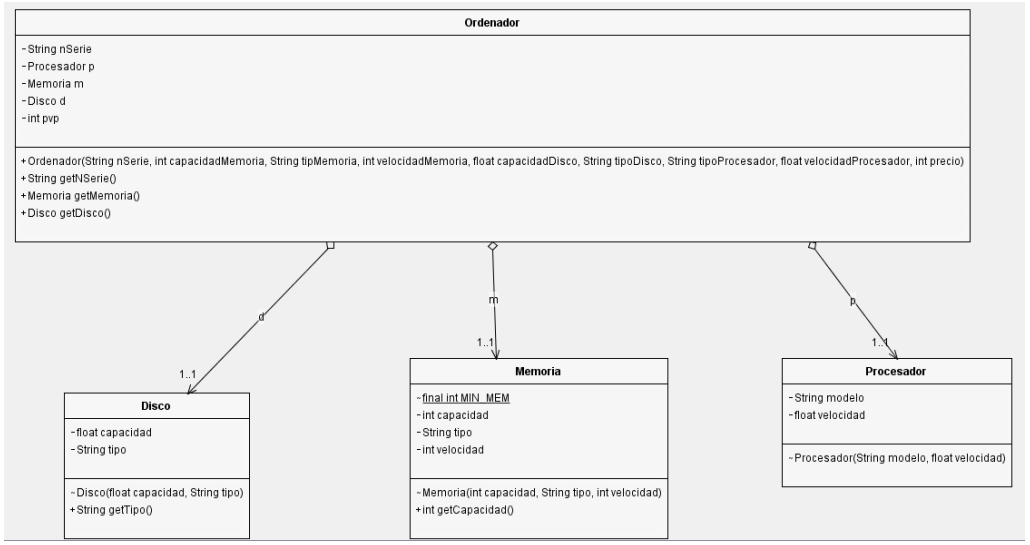
- Para accesos dentro de mismo paquete basta con modificador de acceso por defecto(no poner nada)
- Para acceso entre paquetes necesito public para las partes que quiero hacer visibles.

EJERCICIO U5_B2_E2:

Queremos crear un proyecto Ordenadores con la siguiente estructura



Las clases del paquete ordenador tienen entre ellas una relación de composición de forma que un "todo"(Ordenador) se compone de "partes"(Disco, Memoria, Procesador)



Se pide escribir en Java el paquete ordenador teniendo en cuenta:

- Los atributos de todas las clases deben ser obligatoriamente privados. Ignora la constante MIN_MEN del gráfico, no lo vamos a usar.
- Genera sólo los métodos get() necesarios para que funcione Principal. No más. Métodos set() no hacen falta.
- Cumple al escribir los métodos get() el principio de ocultación.
- El control de acceso de los constructores y métodos debe ser el más restrictivo posible. Haz public sólo lo estrictamente necesario para que funcione el main(). Esto nos fuerza a no escribir el código mecánicamente.

El código de Principal.java es el siguiente y no se puede cambiar.

Observa que no usar import es muy incómodo, pero está bien no usarlo de vez en cuando para madurar el concepto de "paquete"

```
//Principal.java
package miaplicacion;
//sin import = pesadilla
class Principal {
    public static void main(String[] args) {
```

```
        //no usamos import a proposito para prácticar

        //formato contructor: nSerie,capacidadMemoria,tipMemoria,velocidadMemoria,capacidadDisco,tipoDisco,tipoProcesador,velocidadProcesador,precio
        ordenador.Ordenador miOrdenador1 = new ordenador.Ordenador("Ab155", 8, "DDR3", 533, (float) 3.0, "plato", "i5", (float) 3.3, 400);
        ordenador.Ordenador miOrdenador2 = new ordenador.Ordenador("Bx333", 8, "DDR3", 533, (float) 1.5, "SSD", "i5", (float) 3.3, 400);
        ordenador.Ordenador miOrdenador3 = new ordenador.Ordenador("zx900", 8, "DDR3", 533, (float) 1.0, "M2", "i7", (float) 3.3, 400);
        ordenador.Ordenador[] misOrdenadores = {miOrdenador1, miOrdenador2, miOrdenador3};
        for (ordenador.Ordenador o : misOrdenadores) {
            System.out.println(o.getNSerie() + ", memoria " + o.getMemoria().getCapacidad() + "GB, disco " + o.getDisco().getTipo());
        }
    }
}
```

En el siguiente boletín veremos un diseño del paquete ordenador un poco mejorado para evitar que el programador de Principal tenga que tener conocimiento de la existencia de clases como *Memoria* o *Disco* y no sea necesarias utilizar cosas tan enrevesadas como

```
o.getDisco().getTipo()
```