Tema 6: Introducción a la Programación Orientada a Objetos

Introducción a la Programación
Grado en Ingeniería Informática, EPI Gijón

{jdiez,oluaces,juanjo}@uniovi.es

Departamento de Informática - Universidad de Oviedo en Gijón

Objetivos

- Ser capaz de diseñar y programar correctamente una clase completa en lenguaje Java.
- Conocer la utilidad de los constructores de una clase y saber elegir el conjunto de constructores adecuados dada la funcionalidad esperada de una clase.
- Saber crear métodos sobrecargados.
- Comprender el concepto de composición y saber diseñar clases que contengan objetos de otras clases.
- Introducir el concepto de herencia a traves de la Clase `Object`.

Contenidos

Resumen del Tema 3

Clases y objetos. Atributos y métodos. Parte pública y parte privada

- Cuando escribimos el código de una clase estamos definiendo el comportamiento de los objetos de esa clase.
- Las clases constan fundamentalmente de:
  1. atributos: los datos que permiten representar los distintos objetos de esa clase, y
  2. métodos: que implementan las acciones que los objetos pueden hacer.
- Al escribir el código de una clase se ocultan los detalles de implementación (parte privada) y se proporciona una interfaz (parte pública) que los programadores emplean en sus programas para usar los objetos de la clase.
- A partir de la definición de una o varias clases podemos hacer un programa que empleando objetos de esas clases realice las tareas para las que está diseñado.

Necesitamos más cosas

Hay que cuidar muchos detalles para construir un clase correctamente

1. Cómo se inicializan los objetos de una clase (constructores).
2. Cómo se representan las clases y sus objetos, cómo se guardan los métodos y los atributos (elementos estáticos y no estáticos).
3. Cómo se relacionan las clases entre sí, por ejemplo, hay objetos que contienen otros objetos para representarse (composición).
4. Cómo se organizan las clases en Java y la relación jerárquica que existe entre ellas (herencia).

Programa: Inicializar objetos Círculo

Dar una valor inicial a los atributos desde su creación

Enunciado

Modificar la clase `Círculo` *de forma que los objetos de la clase se puedan crear e inicializar de tres formas distintas:*

1. *Indicando un valor real para el atributo* `radio`.
2. *Indicando otro objeto* `Círculo`, *ambos tendrán el mismo valor para el* `radio`.
3. *Por defecto: el* `radio` *debe valer 1.*

Hasta ahora inicializábamos los objetos mediante dos instrucciones:

```
9         Círculo c = new Círculo();
14        c.setRadio(teclado.nextDouble());
```

Sería mejor hacerlo con una sola, por ejemplo:

```
15        Círculo c2 = new Círculo(teclado.nextDouble());
```

Constructor

Permite inicializar los objetos

Constructor

Es un método de una clase cuyo objetivo es permitir la inicialización de los objetos de esa clase.

Los constructores cumplen las siguientes propiedades:

1. Son métodos públicos que se llaman como la clase,
2. no devuelven ningún valor,
3. podemos tener distintas versiones, con distintos parámetros (sobrecarga),
4. su misión es inicializar correctamente los objetos de la clase, y
5. se invocan implícitamente cuando se crea el objeto con `new`.

Importante

Los constructores son los únicos métodos que no tienen tipo de retorno, no se puede poner nada, tampoco `void`

Clase Círculo: Constructores

Tres opciones para inicializar un objeto Círculo

```
 5 public class Círculo {
       ...
16     /** Constructor por defecto, sin parámetros, radio = 1 */
17     public Círculo() {
18         setRadio(1.0);
19     }

21     /** Constructor copia, un parámetro Círculo, el objeto se inicializa
22      *  con los mismos datos que el objeto Círculo pasado como argumento
23      * @param c objeto Círculo, radio=c.radio  */
24     public Círculo(Círculo c) {
25         setRadio(c.getRadio());
26     }

28     /** Constructor con un parámetro double valor inicial del radio
29      * @param r  valor inicial del radio, radio = r */
30     public Círculo(double r) {
31         setRadio(r);
32     }

       ...
54 }
```

Creación de objetos con **new**

Se selecciona el constructor para inicializar el objeto

Creación de un objeto

```
clase nombre = new clase(argumentos)
```

```
11        Círculo c1 = new Círculo();
15        Círculo c2 = new Círculo(teclado.nextDouble());
17        Círculo c3 = new Círculo(c2);
```

El constructor que se aplica es aquél cuyos parámetros casan en número y tipo con los argumentos que se suministran al usar el operador new.

Importante

Crear un objeto con unos argumentos que no casan con los de ningún constructor de la clase es un error sintáctico

Constructor por defecto

Inicialización por defecto

Su papel es generar un estado inicial típico en aquellos objetos para los que no se especifica una inicialización más concreta.

```
17      public Círculo() {
18          setRadio(1.0);
19      }
```

1. no tiene parámetros, y
2. asigna valores iniciales típicos a los atributos.

Importante

Si se declaran otros constructores y no se incluye el constructor por defecto, entonces no podrán crearse objetos con la sintaxis

$$clase\ objeto = \texttt{new}\ clase();$$

Constructor copia

Crea un objeto igual que otro ya existente

Sirve para crear un objeto que sea una copia de otro que ya existe.

```
24      public Círculo(Círculo c) {
25          setRadio(c.getRadio());
26      }
```

1. tiene como parámetro un objeto de la propia clase,
2. inicializa el nuevo objeto para que sea una copia del objeto pasado como argumento, y
3. tras la creación del nuevo objeto, cualquier modificación en uno de los objetos no afecta al otro.

Importante

Se debe emplear el constructor copia en lugar del operador de asignación cuando lo que se quiere es crear una copia independiente de un objeto ya existente

Otros constructores

¿Cuántos? ¿Cuáles?

- No se trata de incluir cuantos más constructores mejor.
- Basta con crear los constructores útiles y necesarios.
- El número de constructores y los parámetros de cada uno de ellos dependen totalmente de la clase.
- El programador debe pensar en las formas más naturales e intuitivas, desde el punto de vista del programador que usa la clase, para inicializar los objetos.
- No deben hacerse constructores duplicados.

```
30      public Círculo(double r) {
31          setRadio(r);
32      }
```

Si una clase no tiene constructores...

¿Por qué y cómo funciona la creación de objetos?

Importante

Cuando una clase **no tiene ningún constructor**, *el compilador de Java genera de forma automática un constructor por defecto*

El constructor generado hace lo siguiente:

- Incluye solamente el código necesario para inicializar los atributos de la clase.
- Hay dos opciones:
  1. Si el atributo tiene un valor inicial en su declaración, entonces el constructor incluye esa inicialización.
  2. Si el atributo no tiene un valor inicial, entonces se asignan los valores iniciales típicos según el tipo de dato:
     - enteros y reales: el valor inicial es 0.
     - `boolean`: valdrá `false`.
     - objetos (incluyendo vectores): el valor inicial es `null`.

Clases sin constructores

Ejemplos de temas anteriores

Ejemplo1: Clase `Círculo` del Tema 3:

```
6     private double radio;
```

El constructor por defecto generado hace que el valor inicial del atributo `radio` sea 0.

Ejemplo2: Clase `Temperaturas` del Tema 5:

```
6     private int[ ] grados = new int[24];
```

El constructor por defecto generado crea un vector de 24 componentes para cada objeto `Temperaturas`.

Importante

Si escribimos una clase sin constructores, la forma de asignar un valor inicial concreto a un atributo para cada objeto que se cree se consigue dando ese valor inicial en la declaración del atributo

Valores iniciales de los atributos

Doble inicialización

- El valor inicial de un atributo, si tiene, es siempre el valor inicial de ese atributo para cualquier objeto que se declare.
- Antes de ejecutarse las sentencias del constructor, los valores iniciales de los atributos del objeto son los valores iniciales dados en la declaración de los atributos en la clase.
- Hay una doble inicialización: primero se asignan los valores iniciales y luego se ejecuta el constructor aplicable según la sentencia `new`

```
public class ClaseX {
    private int número=5;  //con valor inicial

    public ClaseX(int n) {
        //número=5
        número=n;
    }
}
```

Valores iniciales de los atributos

Si no hay un valor inicial en la declaración, se toma el del tipo

En ausencia de valores iniciales, se emplean los valores iniciales típicos correspondientes al tipo del atributo (0, `false` o `null`).

```
public class ClaseY {
    private int número;  //sin valor inicial

    public ClaseY(int n) {
        //número=0
        número=n;
    }
}
```

Importante

Es preferible especificar claramente en los constructores el valor que va a tomar cada atributo

Tareas de los constructores

Se encargan también de la adquisición de recursos

- La principal misión de un constructor es dar un valor inicial a los atributos.
- Cuando los atributos no son de los tipos básicos eso implica más cosas. Dos ejemplos:
  1. Si el atributo es un objeto de otra clase (composición) hay que crearlo con new e inicializarlo con el constructor oportuno.
  2. Si el atributo es un vector habrá que reservar el espacio de memoria suficiente con new e inicializar sus componentes.
- Cuando la clase extiende una clase ya existente (herencia), entonces se debe inicializar la parte del objeto que se hereda de la clase que se está extendiendo.

Importante

Construcción significa creación e inicialización de todos los elementos que posea el objeto

Recolector de basura

Destrucción de objetos

Recolector de basura

Es el proceso implícito que se encarga de liberar la memoria de los objetos que ya no son usados.

- Se encarga de controlar, para cada objeto que se crea, cuántas variables lo están referenciado.
- Cuando un objeto no está siendo referenciado por ninguna variable, su memoria es candidata a ser liberada.
- Cada cierto tiempo se invoca al Recolector de basura que comprueba en la lista de objetos creados cuáles ya no se usan y libera la memoria que ocupan.
- Hay varias estrategias para decidir cuándo ejecutar el Recolector, por ejemplo:
  1. Cuando no quede memoria libre.
  2. Cuando la cantidad de memoria libre sea menor que un umbral.
  3. Justo antes de reservar nuevos objetos.

Recolector: ventajas y desventajas

Alternativa: que la liberación la hicieran explícitamente los programadores

- Ventajas:
  1. Es un mecanismo libre de errores.
  2. Es un proceso transparente para el programador, no tiene que ocuparse de realizar la liberación de memoria y puede centrarse en las tareas de su programa.
- Desventajas:
  1. La gestión de la memoria es más ineficiente.
  2. El Recolector de basura consume tiempo de ejecución, cuanto más veces se ejecute, más ralentiza las aplicaciones.

Importante

La liberación de recursos por parte del Recolector es un proceso más seguro pero menos eficiente que si fuera llevado a cabo por los programadores

El método finalize()

Apenas se usa en la práctica

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

- Su misión es realizar las tareas necesarias previas a la destrucción de un objeto.
- No implica liberar memoria, se encarga de otras tareas.
- Por ejemplo, si un objeto de una clase realiza operaciones de E/S, antes de destruir el objeto es conveniente hacer todas las transacciones de E/S pendientes.
- El método es invocado (en teoría) antes de que el Recolector libere el objeto, por lo que tampoco el programador tiene control sobre la ejecución del mismo.
- En la práctica no se suele usar.
- Se puede definir otro método que haga las tareas que deseemos cuando el objeto ya no siga usándose.

Dos tipos de clases

Clases para crear objetos, clases para hacer aplicaciones

Hemos hecho dos tipos de clases:

1. Clases a partir de las creamos objetos, como la clase `Círculo`.
2. Clases para hacer aplicaciones de consola.

Esto mismo ocurre con las clases de Java:

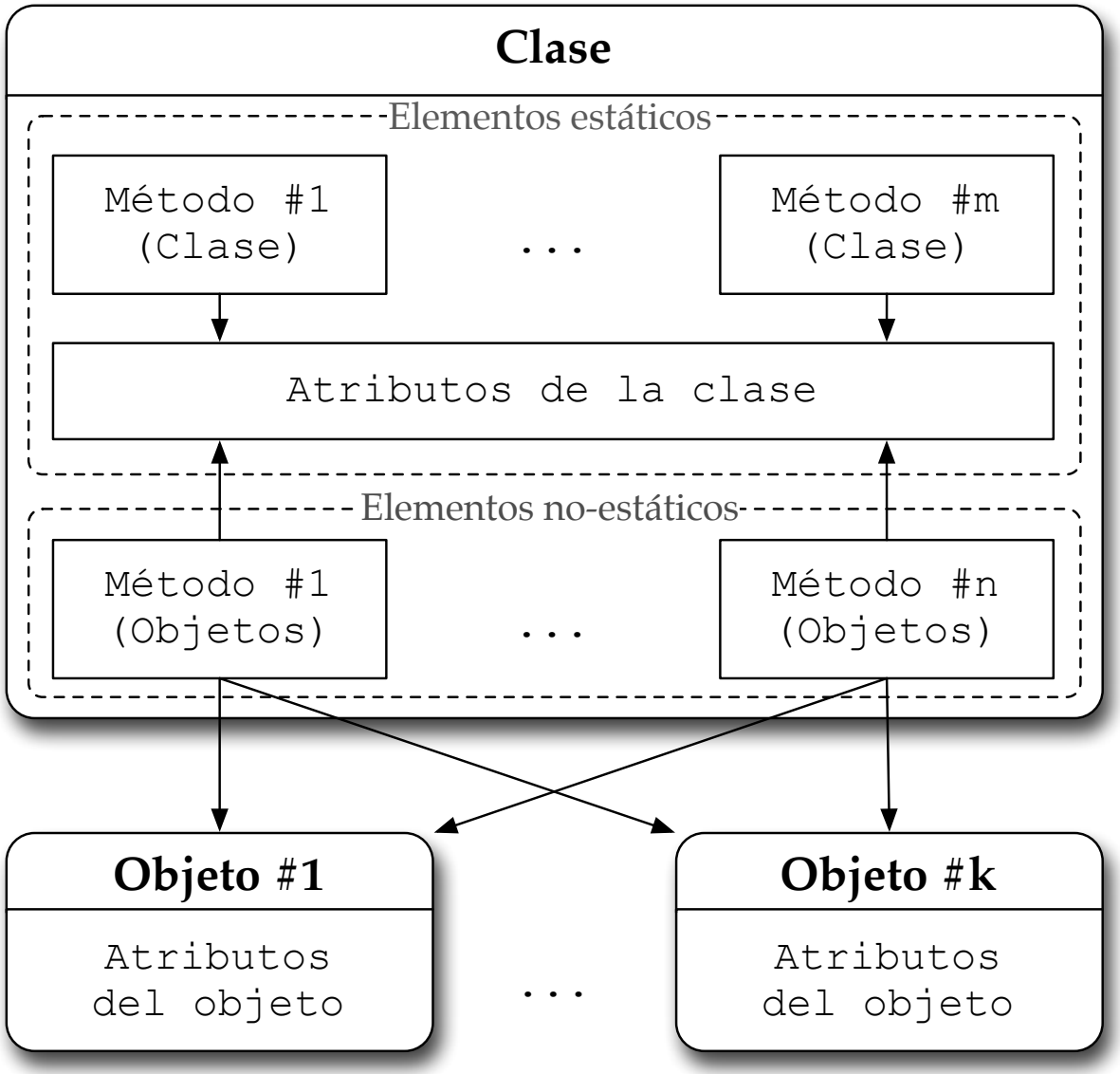
1. Hay clases que nos sirven para crear objetos, como `String` o `Scanner`, y
2. otras que solamente nos sirven para usar sus métodos, como `Math`.

Importante

Hay veces que los elementos de una clase están ligados a la clase, no a los objetos de la clase

Representación de clases y objetos

¿Cómo se mantienen en memoria los atributos, métodos y objetos?



Enunciado: clase Conversor

Un programa con métodos asociados a la clase

Enunciado

Crear la clase `Conversor` *que permita realizar conversiones entre kilómetros y millas, y entre kilogramos y libras.*

- ¿Tiene atributos esta clase?
- ¿Qué tiene que ver una cantidad en kilómetros, con otra en kilogramos?
- Es preferible seguir un enfoque como el de clase `Math`.
- La clase `Conversor` va a proporcionar un **conjunto de métodos asociados con la clase**, y no va a servir para crear objetos, ni va a guardar las cantidades que convierta.
- Para lograrlo necesitamos usar el modificador `static`.

Elementos estáticos (**static**)

Pertenecen a la clase, no a los objetos

Importante

El modificador `static` *indica que un elemento está asociado con la clase, no con los objetos de la clase*

Características:

- Se añade al declarar un elemento dentro de la clase, ya sea un método o un atributo.
- Indica que ese elemento será de la clase.
- Atributos estáticos: los objetos no tendrán ese atributo. Sólo existirá una copia del atributo y se guardará en la clase.
- Métodos estáticos (diferencias con los no estáticos):
  - Solamente pueden acceder a los atributos estáticos.
  - Solamente pueden llamar a otros métodos estáticos.

Clase Conversor

Constantes estáticas y eliminación del constructor

```
 5 public class Conversor {

 7     //Atributos estáticos
 8     /** Constante para convertir de Millas a Km*/
 9     public static final double MILLAS_KM = 1.609344;
10     /** Constante para convertir de Libras a Kg*/
11     public static final double LIBRAS_KG = 0.45359237;

13     //Evitar la construcción de objetos
14     /** Para que no se puedan crear objetos de la clase*/
15     private Conversor(){}
      ...
```

Clase Conversor

Métodos estáticos

```
...
17      //Métodos estáticos

16      public static double millasAKm(double millas) {
17          return millas*MILLAS_KM;
18      }

28      public static double kmAMillas(double km) {
29          return km/MILLAS_KM;
30      }

35      public static double librasAKg(double libras) {
36          return libras*LIBRAS_KG;
37      }

42      public static double kgALibras(double kg) {
43          return kg/LIBRAS_KG;
44      }
45  }
```

Usando un clase con elementos estáticos

El operador punto aplicado sobre la clase, no sobre un objeto

Acceso a los elementos estáticos (operador punto .)

`clase.elemento_estático_público`

Por ejemplo, con la clase `Math` podemos calcular:

```
double senoPI=Math.sin(Math.PI); //Seno de la constante PI
```

Y lo mismo podemos hacer con nuestra clase `Conversor`:

```
18    case 1://Km a millas
19        System.out.print("Introduce km: ");
20        km =teclado.nextDouble();
21        millas = Conversor.kmAMillas(km);
22        System.out.printf("%.2f Km son %.2f millas",km,millas);
23        break;
```

El objeto **this**

Cómo indicarle a los métodos el objeto sobre el que trabajan

Importante

Como los objetos comparten la implementación de los métodos, tenemos que decirles a los métodos no estáticos de alguna manera sobre qué objeto deben desempeñar su tarea

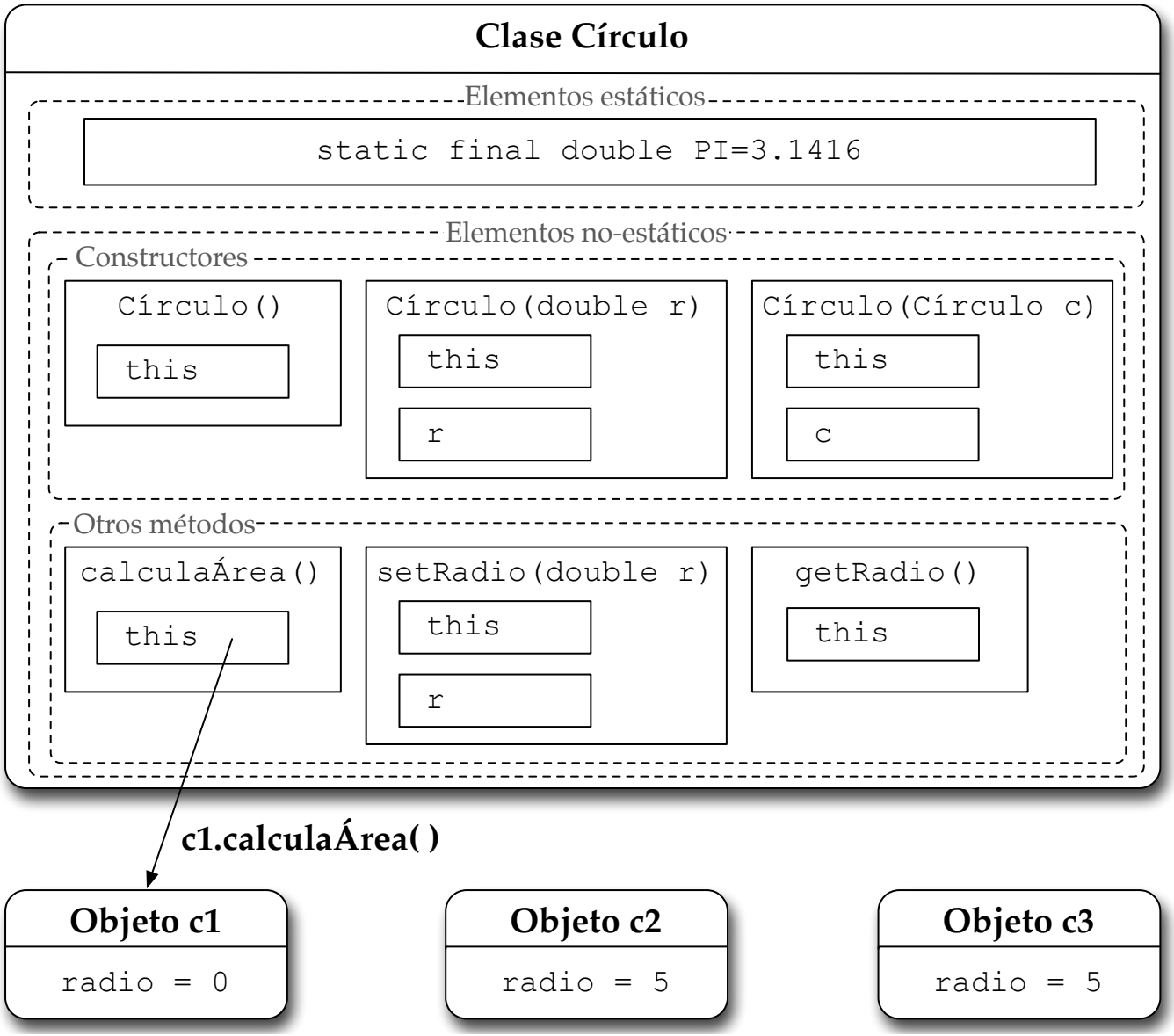
```
11          Círculo c1 = new Círculo();
15          Círculo c2 = new Círculo(teclado.nextDouble());
17          Círculo c3 = new Círculo(c2);

19          System.out.printf("c1 área %f\n",c1.calculaÁrea());
20          System.out.printf("c2 área %f\n",c2.calculaÁrea());
21          System.out.printf("c3 área %f\n",c3.calculaÁrea());
```

this

Es el parámetro oculto que tienen todos los métodos no estáticos y que se refiere al objeto que ha invocado el método.

El objeto **this**

Una representación gráfica



this está implícito

Cada acceso a un atributo o llamada a un método, lleva implícito **this**

Importante

Cada vez que en las sentencias de un método aparece un atributo u otro método se reemplaza por la misma expresión añadiendo `this`

Escribimos

```
37      public double getRadio() {
38          return radio;
39      }
```

pero es como escribir implícitamente

```
    public double getRadio() {
        return this.radio;
    }
```

Uso #1: llamar a otro constructor

Invocar un constructor desde otro constructor

```
 5 public class Círculo {
       ...
17     public Círculo() {
18         this(1.0);
19     }
24     public Círculo(Círculo c) {
25         this(c.getRadio());
26     }
30     public Círculo(double radio) {
31         this.setRadio(radio);
32     }
       ...
66 }
```

Importante

La llamada de un constructor a otro usando `this` *debe hacerse en la primera sentencia del constructor*

Uso #2: acceder a los atributos

Los parámetros ocultan los atributos por llamarse igual

- Al dar al parámetro el mismo nombre que tiene el atributo, se oculta el atributo.
- Necesitamos usar `this` para referirnos al valor del atributo.

```
 5 public class Círculo {
       ...
35     /**Devuelve el valor del radio del objeto Círculo
36      * @return el radio del objeto */
37     public double getRadio() {
38         return this.radio;
39     }
44     /**Cambia el valor del radio del objeto, para que valga r
45      * @param radio nuevo valor para el radio del objeto
46      * @return nada  */
47     public void setRadio(double radio) {
48         if ( radio >= 0 )   this.radio=radio;
49     }
       ...
66 }
```

Uso #3: referirnos a todo el objeto

Para devolverlo o para hacer algo con todo el objeto

```
 5 public class Círculo {
       ...
54     //Adaptación de los métodos heredados de la clase Object
55     /**Devuelve cierto si dos objetos Círculo son iguales
56      * @param obj el objeto con el que se va a comparar
57      * @return true si el radio de los dos objetos es igual*/
58     @Override
59     public boolean equals(Object obj) {
60         if ( this == obj ) return true;
61         if ( obj instanceof Círculo )
62             return ( this.getRadio() == ((Círculo)obj).getRadio() );
63         else return false;
64     }

66 }
```

Métodos sobrecargados

Ejecutar el mismo método con distintos datos

Sobrecarga

Un método está sobrecargado en Java cuando existen distintas versiones que se diferencian en el número, orden, y/o el tipo de los parámetros.

Los métodos sobrecargados deben diferenciarse en:

- el número de parámetros, o
- el tipo de los parámetros, o
- en el orden de los parámetros.

Importante

Si dos métodos sobrecargados se diferencian solamente por el valor de retorno es un error sintáctico

Método print()

Un ejemplo de método sobrecargado

- Pertenece a la clase `PrintStream`, la clase de `System.out`.
- Tiene una versión para imprimir los tipos básicos, vectores de tipo `char` y objetos de las clases `String` y `Object`.

```
public void print(boolean b)
public void print(char c)
public void print(char[ ] s)
public void print(float f)
public void print(double d)
public void print(int i)
public void print(long l)
public void print(Object obj)
public void print(String s)
```

Enunciado: Clase Máximos

Otra clase estática como un método sobrecargado

Enunciado

Crear la clase `Máximos` *que proporcione métodos para calcular el máximo de:*

1. *dos números enteros, y*
2. *tres números reales.*

Dado el enunciado debemos aplicar varios conceptos:

- La clase puede implementarse mediante métodos estáticos.
- Necesitamos un método sobrecargado ya que tenemos funcionalidades diferentes para una misma acción.

Clase Máximos

Métodos sobrecargados estáticos

```
 4 public class Máximos {
       ...
15     public static int max(int v1, int v2) {
16         return ( v1 >= v2 ? v1 : v2 );
17     }

24     public static double max(double v1, double v2, double v3) {
25         if ( v1 >= v2 && v1 >= v3 ) return v1;
26         else return ( v2 >= v3 ? v2 : v3 );
27     }

29 }
```

Resolución de sobrecarga

Reglas para elegir la versión del método sobrecargado a ejecutar

Por orden de aplicación:

1. Si la lista de argumentos coincide en tipo, orden y número con la lista de parámetros que tenga una versión del método sobrecargado, entonces se invoca dicho método.
2. Si no hay ninguna versión que case exactamente, entonces se busca una que coincida en número de parámetros y cuyos argumentos se puedan promover para que concuerden con el tipo de los parámetros.
3. Si ninguna de las dos opciones anteriores funciona, entonces se busca si la llamada casa con algún método que tenga un número variable de argumentos, realizando conversiones automáticas si fuera necesario.

```
6   System.out.println(Máximos.max(5,6));    //max(int,int)
7   System.out.println(Máximos.max(4,7,3)); //max(double,double,double)
```

Relaciones entre clases

Las aplicaciones grandes no se resuelven mediante una sola clase

- Las aplicaciones usan varias clases para poder hacer sus tareas.
- Muchas veces esas clases no son independientes entre sí.
  Existen dos tipos de relaciones principales entre clases:
  1. **Composición**: los objetos de una clase pueden estar compuestos por objetos de otras clases. Es la relación *tiene-un*.

     Ejemplos:
     - Un coche tiene un volante, un motor,...
     - Un `Círculo` tiene un `Punto` para describir su `centro`.
  2. **Herencia**: una clase puede ser una especialización de otra ya existente heredando su comportamiento. Es la relación *es-un*.

     Ejemplos:
     - Un humano es un mamífero.
     - Un `CírculoGráfico` es un `Círculo` que además se puede representar gráficamente.

Ventajas de la Composición y la Herencia

Es una forma de reutilización de código

- Al hacer que una clase use de alguna forma otra clase, bien sea mediante una relación de composición o de herencia, se está favoreciendo la reutilización del código.
- Se reutilizan clases existentes que ya han sido probadas y verificadas.
- Esto evita que, al diseñar una nueva clase, el programador tenga que reinventar la rueda.
- Muchos beneficios:
  1. Agiliza el tiempo de desarrollo,
  2. mejora la fiabilidad del software, y
  3. reduce costes.

Enunciado: Clase Círculo (con centro)

Una relación de composición

Enunciado

Modificar la clase `Círculo` *de manera que los objetos guarden además del* `radio`*, las coordenadas* `X` *e* `Y` *de la posición donde se sitúa el centro del* `Círculo`

- La primera idea sería declarar dos nuevos atributos: `X` e `Y`.
- Pero según el enfoque de la POO, el `centro` es un objeto.
- Tendríamos que un objeto de la clase `Círculo` *tiene-un* objeto de la clase `Punto` para describir su `centro`.
- A esa relación se la denomina **composición**: la clase `Círculo` está compuesta por la clase `Punto`.
- En la implementación tendremos que considerar:
  1. Constructores: deben crear el objeto `Punto`.
  2. Métodos: se deben basar en la funcionalidad ofrecida por los métodos públicos de la clase `Punto`.

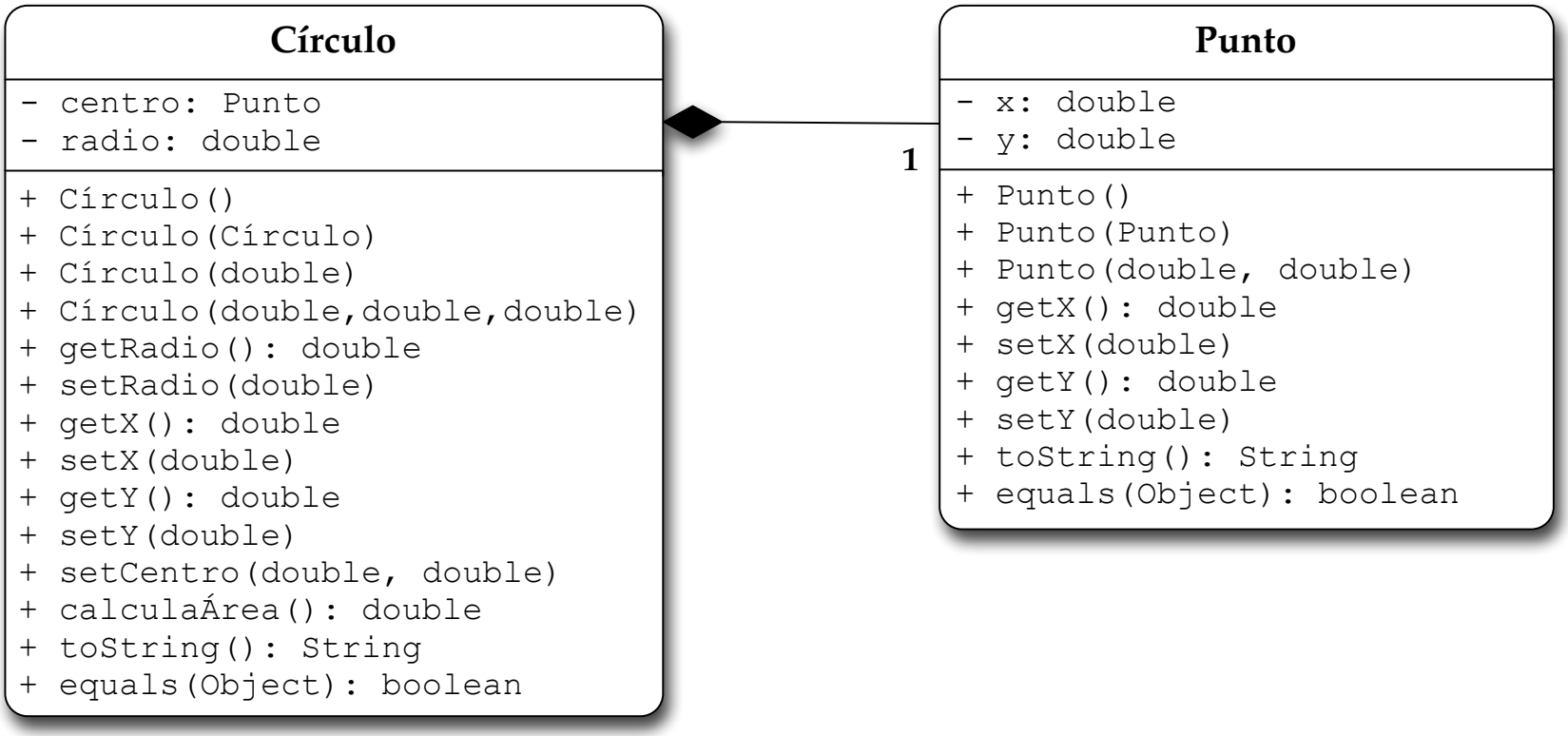
Composición

Es la relación *tiene-un*

Composición

Se produce cuando los objetos de una clase tienen entre sus atributos objetos de otras clases.

Un objeto `Círculo` tiene un objeto `Punto` para describir su centro



Clase Círculo: constructores

Hay que crear el objeto Punto para el atributo centro

```
 5 public class Círculo {
       ...
13     private Punto centro;
15     private double radio;
19     public Círculo() {
20         this(1.0,0.0,0.0);
21     }
26     public Círculo(Círculo c) {
27         this(c.getRadio(),c.getX(),c.getY());
28     }
32     public Círculo(double radio) {
33         this(radio, 0.0, 0.0);
34     }
41     public Círculo(double radio, double x, double y) {
42         this.centro=new Punto(x,y);
43         this.setRadio(radio);
44     }
       ...
```

Clase Círculo: métodos

Se implementan basándonos en los métodos públicos de la clase Punto

```
62      public double getX() {
63          //usamos el método getX de la clase Punto
64          return this.centro.getX();
65      }
69      public double getY() {
70          return this.centro.getY();
71      }
75      public void setX(double x) {
76          //usamos el método setX de la clase Punto
77          this.centro.setX(x);
78      }
82      public void setY(double y) {
83          this.centro.setY(y);
84      }
89      public void setCentro(double x, double y) {
90          this.setX(x);
91          this.setY(y);
92      }
```

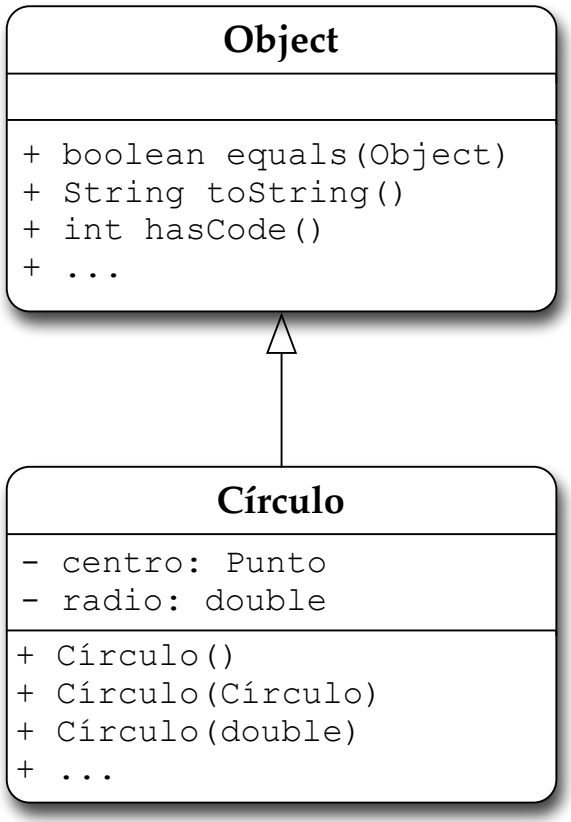
Herencia

Es la relación *es-un*

Herencia

Es la relación jerárquica que se establece entre una nueva clase y otra ya existente, y que permite que la nueva clase herede los atributos y métodos de la otra clase.

- La nueva clase (subclase) extiende la funcionalidad de la clase de la que hereda (superclase).
- La subclase puede modificar el comportamiento de su superclase:
  1. añadiendo nuevos atributos y métodos, o
  2. sobrescribiendo los atributos o métodos que hereda.



La clase Object

Es la clase base en la jerarquía de objetos en Java

Método	Descripción
`boolean equals(Object)`	Devuelve cierto si los dos objetos que compara son iguales.
`String toString()`	Retorna una representación en forma de String con la información que almacena el objeto.
`int hashCode()`	Devuelve un código entero que representa al objeto. Si dos objetos de la clase tienen códigos distintos es que son diferentes (lo contrario no siempre es cierto).
`Object clone()`	Produce y devuelve un nuevo objeto que es una copia del objeto con el que se invoca el método.
`void finalize()`	Incluye las tareas que deben hacerse antes de que el objeto se destruya.
`Class getClass()`	Retorna una objeto de la clase Class con información sobre la clase del objeto.
`void notify()`	
`void notifyAll()`	
`void wait()`	Tienen que ver con tareas para programas multi-hilo.
`void wait(long)`	
`void wait(long , int)`	

Método equals()

Comprueba que dos objetos sean iguales, aunque no sean el mismo objeto

- El operador `instanceof`: devuelve `true` cuando el objeto indicado como primer operando es de la clase indicada como segundo operando.
- Convertir mediante una conversión explícita el parámetro `Object` a un objeto de la clase que estemos programando.

```
104     @Override
105     public boolean equals(Object obj) {
106         if ( this == obj ) return true;
107         if ( obj instanceof Círculo ) {
108             Círculo c = (Círculo) obj;
109             return ( this.centro.equals(c.centro) &&
110                     this.getRadio() == c.getRadio() );
111         }
112         else return false;
113     }
```

Método toString()

Convierte un objeto en String

Se aplica (implícitamente) en dos situaciones:

- cuando el objeto se imprime como una cadena (modificador %s), o
- cuando el objeto se concatena mediante el operador + con un `String`.

```
11          Círculo c1 = new Círculo();
19          System.out.printf("\nc1 %s", c1);
```

```
115     /** Convierte a String la información del objeto Círculo
116      * @return un String con la información del objeto   */
117     @Override
118     public String toString() {
119         return String.format("%s radio %.2f",
120                              this.centro, this.getRadio());
121     }
```

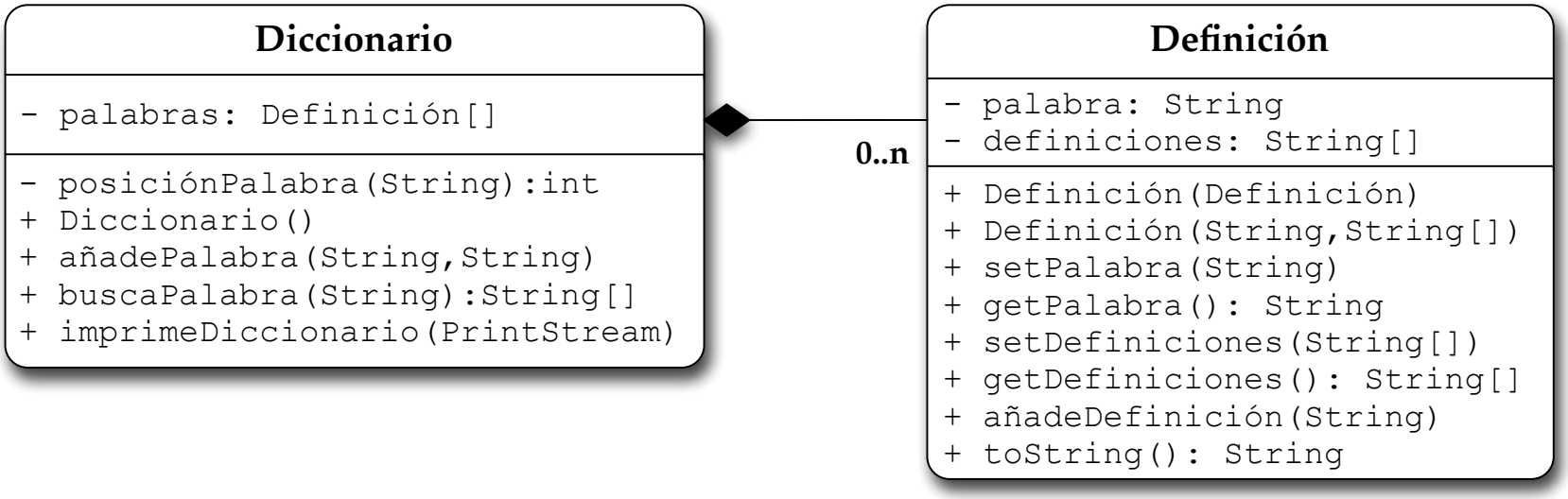
Enunciado: Diccionario

Una relación de composición con un vector de objetos de otra clase

Enunciado

Realizar una clase que permita representar y trabajar con un diccionario de palabras.

- Relación de composición con varios objetos.
- Necesitaremos un vector para representarlos.



Clase Definición

Atributos, constructores y métodos

```
 4 public  class Definición {

 8     private String palabra;

10     private String[ ] definiciones;

15     Definición(Definición d) {
16         this(d.getPalabra(),d.getDefiniciones());
17     }

22     Definición(String palabra, String[ ] definiciones) {
23         this.setPalabra(palabra);
24         this.setDefiniciones(definiciones);
25     }

30     public String[ ] getDefiniciones() {
31         return this.definiciones;
32     }

36     public void setDefiniciones(String[ ] definiciones) {
37         this.definiciones=new String[definiciones.length];
38         for (int i=0; i<definiciones.length;i++)
39             this.definiciones[i]=definiciones[i];
40     }

       ...
```

Clase Definición

Más métodos y sobrescritura de toString()

```
44      public String getPalabra() {
45          return this.palabra;
46      }
50      public void setPalabra(String palabra) {
51          this.palabra=palabra;
52      }
56      public void añadeDefinición(String definición) {
57          String[ ] tmp = this.definiciones;
58          this.definiciones=new String[tmp.length+1];
59          for (int i=0; i<tmp.length;i++)
60              this.definiciones[i]=tmp[i];
61          this.definiciones[this.definiciones.length-1] = definición;
62      }
68      public String toString() {
69          String s = String.format("%s: ", this.getPalabra());
70          for (int i=0; i<this.definiciones.length;i++)
71              s = s + String.format("\n\t#%d %s",i+1,this.
                    definiciones[i]);
72          return s;
73      }
74  }
```

Clase Diccionario

Atributo, constructor y métodos

```
 6 public class Diccionario {
10     private Definición[ ] palabras;
14     public Diccionario() {
15         this.palabras=new Definición[0]; //Crea un diccionario vacío
16     }
23     public void añadePalabra(String p, String d) {
24         int index=this.posiciónPalabra(p);
25         if (index!=-1) {
26             this.palabras[index].añadeDefinición(d);
27         }
28         else {
29             Definición[ ] tmp=this.palabras;
30             this.palabras=new Definición[tmp.length+1];
31             for (int i=0; i<tmp.length; i++)
32               this.palabras[i]=tmp[i];
33             String[] defs=new String[1];
34             defs[0]=d;
35             this.palabras[this.palabras.length-1]=new Definición(p,
                   defs);
36         }
37     }
       ...
```

Clase Diccionario

Más métodos

```
43      public String[ ] buscaPalabra(String p) {
44          int index=this.posiciónPalabra(p);
45          if (index!=-1) {
46               return this.palabras[index].getDefiniciones();
47          }
48          else return null;
49      }

53      public void imprimeDiccionario(PrintStream ps) {
54          for (int i=0; i<this.palabras.length;i++)
55              ps.println(this.palabras[i]);
56      }

63      private int posiciónPalabra(String p) {
64          int i=0;
65          while ( (i<this.palabras.length) &&
66                  (!p.equals(this.palabras[i].getPalabra())) )
67              i++;
68          return ( i < this.palabras.length ? i : -1);
69      }
70  }
```

Clase Diccionario

Programa de ejemplo

```
 9   public static void main(String[] args) throws FileNotFoundException {
10     //Objeto File para acceder a un fichero
11     File f = new File ( "palabras.txt" );
12     //Objeto Scanner asociado con el objeto f
13     Scanner fichero= new Scanner(f);
14     //Creamos el objeto diccionario, inicialmente vacío
15     Diccionario dicc=new Diccionario();
16     //Leemos el número de palabras que tiene el fichero
17     int palabras = fichero.nextInt();
18     //Leemos las palabras y sus significados y se añaden al diccionario
19     for (int i=1; i <= palabras; i++) {
20       String p=fichero.next();
21       String d=fichero.nextLine();
22       dicc.añadePalabra(p, d);
23     }
24     System.out.println("Diccionario"); //Lo imprimimos
25     dicc.imprimeDiccionario(System.out);

       ...
40   }
```

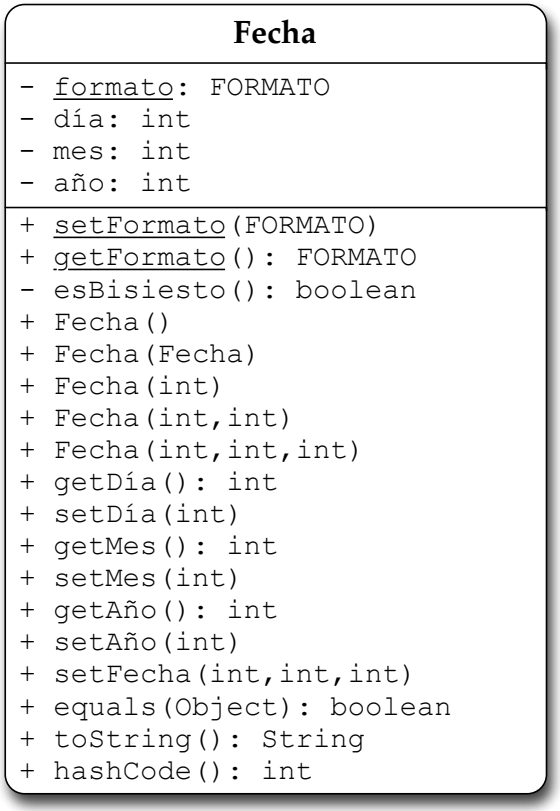
Clase Fecha

Una versión completa

Enunciado

Realizar la clase `Fecha`*, que sobrescriba los métodos de* `equals()`*,* `toString()` *y* `hashCode()` *y que permita imprimir las fechas en dos formatos: dd/mm/aaaa y dd de Mes de aaaa.*

- Es una clase completa, con una funcionalidad clara y útil.
- Vamos a programar los aspectos del formato usando:
  1. Una enumeración.
  2. Elementos estáticos.
- Además programaremos el método `hashCode()` y explicaremos su utilidad.



Clase Fecha

Enumeración FORMATO, elementos estáticos y atributos

```
 5 /** Enumeración FORMATO
 6  * <p>Define los dos tipos de formatos para mostrar objetos Fecha:
 7  * <it>FORMATO.DDMMAAAA dd/mm/aaaa, p.e. 31/12/2050</it>
 8  * <it>FORMATO.TEXTO dd de Mes de aaaa, p.e. 31 de Diciembre de 2050</it> */
 9 enum FORMATO { DDMMAAAA, TEXTO }

17 public class Fecha {

19     private static FORMATO formato=FORMATO.DDMMAAAA;

24     public static void setFormato(FORMATO f){
25         formato=f;
26     }

30     public static FORMATO getFormato() {
31         return formato;
32     }

       ...
```

Clase Fecha

Atributos y constructores

```
36      private int día;

38      private int mes;

40      private int año;

46      public Fecha() {
47          this(1, 1, 2000);
48      }

50      public Fecha(Fecha f) {
51          this(f.getDía(),f.getMes(),f.getAño());
52      }

56      public Fecha(int año) {
57          this(1, 1, año);
58      }

63      public Fecha(int mes, int año) {
64          this(1, mes, año);
65      }

71      public Fecha(int día, int mes, int año) {
72          this.setFecha(día,mes,año);
73      }

        ...
```

Clase Fecha

Método setDía(): usando un vector local

```
 98      public int getDía() {
 99          return this.día;
100      }


104      public void setDía(int día) {
105          int[ ] días_mes={0,31,28,31,30,31,31,30,31,30,31,30,31};
106          if (this.getMes() == 2 && this.esBisiesto())
107                días_mes[2] = 29;
108          if ( (día >= 1) && (día <= días_mes[this.getMes()]) )
109              this.día = día;
110      }
         ...
```

Clase Fecha

Método toString(): la misma idea

```
140     @Override
141     public String toString() {
142         if (formato==FORMATO.DDMMAAAA)
143           return String.format("%02d/%02d/%04d",
144                   this.getDía(),this.getMes(),this.getAño());
145         else {
146             String[ ] meses = { "", "Enero", "Febrero", "Marzo",
147                                 "Abril", "Mayo", "Junio",
148                                 "Julio", "Agosto", "Septiembre",
149                                 "Octubre","Noviembre","Diciembre" };

151             return String.format("%d de %s de %d",
152                    this.getDía(),meses[this.getMes()],this.getAño());
153         }
154     }
        ...
```

Clase Fecha

Métodos equals() y hashCode()

```
159     @Override
160     public boolean equals(Object obj) {
161         if (this==obj) return true;
162         if (obj instanceof Fecha) {
163            Fecha f = (Fecha) obj;
164            return ( this.getDía()==f.getDía() &&
165                     this.getMes()==f.getMes() &&
166                     this.getAño()==f.getAño()   );
167         }
168         else return false;
169     }

174     @Override
175     public int hashCode() {
176         return this.getDía()%10*100 +
177                 this.getMes()%10*10 + this.getAño()%10;
178     }
179 }
```

Clase Fecha

Programa de ejemplo

```
3  public class Ejemplo3Fecha {
5      public static void main(String[ ] args) {
6          //Objetos de la clase Fecha inicializados con los
7          //cinco constructores de la clase
8          Fecha f1 = new Fecha();
9          Fecha f2 = new Fecha(2010);
10         Fecha f3 = new Fecha(8,2010);
11         Fecha f4 = new Fecha(31,12,2010);
12         Fecha f5 = new Fecha(f4);

14         //Los mostramos con el formato por defecto
15         System.out.printf("f1: %s\n",f1);
           ...
21         //Cambiamos el formato
22         Fecha.setFormato(FORMATO.TEXTO);
23         System.out.printf("f1: %s\n",f1);
           ...
28     }
29 }
```

Otra implementación de Fecha

Sólo hay que cambiar los atributos y los métodos set() y get()

```
10 public class Fecha {
13     /** Representación de una Fecha con 8 dígitos DDMMAAAA */
14     private int fecha;

20     /** Vector con el número de día de cada mes, se usa en setDía()
           */
21     private final static int[ ] días_mes={0,31,28,31,30,31,31,
22                                           30,31,30,31,30,31};

97     public int getDía() {
98         return  (this.fecha/1000000);
99     }

104    public void setDía(int día) {
105        int último=( this.getMes()==2 && esBisiesto() ?
106                     29 : días_mes[this.getMes()]);
107        if ( (día >= 1) && (día <= último) ) {
108            this.fecha = día*1000000 +
109                         this.getMes()*10000 + this.getAño();
110        }
111    }
       ...
```

Otra implementación de Fecha

Sólo hay que cambiar los atributos y los métodos set() y get()

```
115     public int getMes() {
116         return  ((this.fecha/10000)%100);
117     }
121     public void setMes(int mes) {
122         if ( mes>=1 && mes<=12 )
123             this.fecha = 1000000*this.getDía() +
124                          10000*mes + this.getAño();
125     }
129     public int getAño() {
130         return (this.fecha%10000);
131     }
136     public void setAño(int año) {
137         if ( año > 0 )
138             this.fecha = 1000000*this.getDía() +
139                          10000*this.getMes() + año;
140     }
        ...
179 }
```