Tema 3. Clases y objetos
Segunda parte

Tipos de métodos según su funcionalidad

- Métodos inicializadores
  - Inicializan atributos y no se trata de constructoras.
  - Tienen interés cuando se quiere reinicializar los valores de los atributos sin invocar a la constructora.
- Métodos accedentes
  - Devuelven el contenido de los atributos (cada accedente devuelve un atributo).
  - Se conocen también como métodos *get*, pues se suelen tener como signatura la siguiente:
    - TipoDeAtributo getNombreDeAtributo()
  - Se utilizan cuando queremos que el *usuario* de la clase tenga acceso a los valores de los atributos (los atributos deberían ser siempre declarados como privados).
  - Este tipo de métodos hay que usarlos con cautela cuando el tipo del atributo es una clase.
    - Si se devuelve el objeto, dependiendo de las circunstancias, se puede estar devolviendo una referencia al valor interno que guarda el objeto y que se suponía debía proteger.
    - En casos sensibles, puede ser necesario copiar el objeto antes de devolverlo.

Tipos de métodos según su funcionalidad

- Métodos mutadores.
  - Establecen el contenido de los atributos (cada mutador, el contenido de un atributo).
  - Se conocen también como métodos *set*, pues se suelen tener como signatura la siguiente.
    - void setNombreDeAtributo(TipoDeAtributo nuevoValor)
  - De forma similar a los métodos accedentes, en este caso permiten al usuario cambiar el valor de un atributo a pesar de que ese atributo no sea público.

Tipos de métodos según su funcionalidad

- Métodos computadores
  - Realizan cálculos y generan resultados.
  - Uno de estos cálculos puede implicar convertir el objeto en una representación distinta, como la conversión de un objeto en una cadena.
  - En Java, todos las clases tienen un método heredado con esta signatura:
    - public String toString()
  - El cual se asume que convierte el objeto entero en una cadena que puede ser mostrada por pantalla o usada para otros fines.
  - Este es el método que se invoca al hacer un *System.out.println* y pasándole como parámetro un objeto o bien cuando se concatena un objeto con una cadena.
  - Por defecto, este método devuelve como valor el nombre de la clase, seguido de una arroba y el resultado de invocar al método *hashcode*, que también es un método heredado presente en todas las clases.
  - Hay que sobreescribir este método para que tenga la funcionalidad adecuada.

Visibilidad de atributos y métodos

- Podemos decir que una clase tiene dos tipos de usuarios:
  - Los usuarios externos a la clase: aquellos que crean instancias (objetos) de la clase y quieren invocar a sus métodos o acceder a sus atributos.
  - Los usuarios internos a la clase: o lo que es lo mismo, la implementación de los métodos.

- Esa división nos lleva a considerar dos categorías de visibilidad:
  - Visibilidad pública → Los elementos de una clase declarados públicos podrán ser utilizados tanto por los usuarios externos como por los internos.
  - Visibilidad privada → Los elementos declarados como privados serán sólo accesibles a los usuarios internos

Visibilidad de atributos y métodos

- Un atributo o método public se conocerá en cualquier otro programa o paquete, pudiendo ser entonces accedido por medio de objetos de esa clase (con la notación punto).
- Un atributo o método private sólo se conoce en la propia clase en la que está definido, pudiendo ser accedido tan sólo en ella.
  - Los métodos serán públicos si constituyen servicios que los objetos proporcionan al exterior y privados si son funciones auxiliares de la clase.

- Las directrices de la programación orientada a objetos nos aseguran un menor riesgo de errores en los programas si:
  - Los atributos se declaran siempre como privados.
  - La manipulación de los atributos se realiza, consecuentemente, por medio de métodos (públicos, claro).

Visibilidad de atributos y métodos

- En Java la división anterior de usuarios internos y externos puede subdividirse:
  - Usuarios externos que pertenecen al mismo paquete.
  - Usuarios externos que no pertenecen al mismo paquete.
- Dado que los paquetes vienen a representar agrupaciones de clases relacionadas, puede tener sentido darles un tratamiento especial en otras clases de un mismo paquete.
- En este caso Java permite darles ciertos privilegios de acceso a los elementos que no están accesibles para los usuarios (clases) situados en otros paquetes.
- En concreto, en Java se puede omitir el modificador de visibilibidad (ni public ni private).
- En ese caso:
  - El elemento es accesible para los usuarios internos (implementación de la propia clase).
  - El elemento es accesible (“público") para los usuarios externos del mismo paquete (para las clases del paquete).
  - El elemento NO es accesible para el resto (clases de otros paquetes).

Visibilidad

- Ejemplo

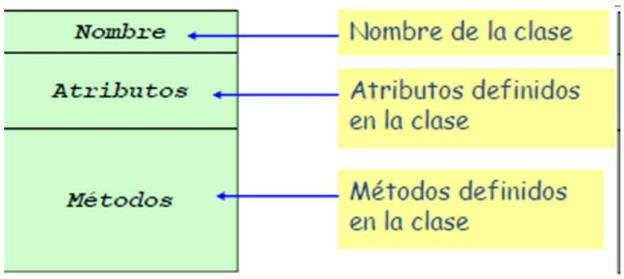
```
class A
{
        private int x, y; // Atributos
}

class B
{ ...
A a = new A();
a.x = 1; // Error de compilación
```

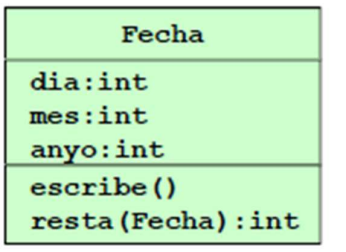
Visibilidad. Ejemplo

```
public class Fecha {
   private int dia;
   private int mes;
   private int anyo;
   // Método público; pueden utilizarlo los usuarios externos.
   public void escribe() {
      //Implementación: usuario "interno". Puede acceder a los elementos privados
      System.out.println("Día: " + this.dia);
   }
   public int resta(Fecha fecha) {
     // Implementación: usuario "interno". Puede acceder a los elementos
     // privados, no sólo del parámetro this, sino también del objeto fecha
     // recibido como parámetro (por ser de la misma clase). Es decir, tanto a
     // this.dia como a fecha.dia.
   ...
   }
```


Representación gráfica de clases

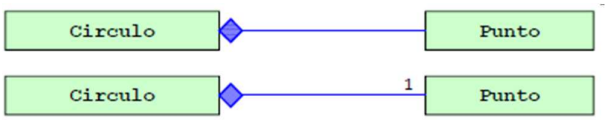
- Para representar gráficamente los elementos de los programas orientados a objetos y sus relaciones vamos a utilizar diagramas similares a los de UML que se usan en Ingeniería del Software.
- A medida que vayamos estudiando los conceptos y los mecanismos de la POO iremos viendo cómo representarlos gráficamente.
- De momento, comencemos con la representación de las clases:
- Si interesa se pueden omitir secciones o dejarlas en blanco dependiendo del nivel de detalle





Representación gráfica de clases

- Las relaciones entre las clases también se representan gráficamente.
- Relación de Clientismo: Una clase contiene objetos de otra.
- Por ejemplo: Los objetos de la clase **`Circulo`** contienen objetos de la clase **`Punto`**.
- La relación de clientelismo se representa conectando las clases mediante líneas con un rombo en un extremo.
- El rombo se coloca en la clase que contiene objetos de la otra:
- Podemos indicar el número de objetos que se contienen:



Igualdad entre objetos

- El operador == responde a la pregunta de si dos objetos son el mismo.
- El método equals responde a la pregunta de si dos objetos contienen la misma información.

```
Fecha f1 = new Fecha(12, 10, 1492);
Fecha f2 = new Fecha(1, 1, 1970);
…
if (f1.equals(f2))
    System.out.println("¡Nacieron el mismo día!");
else
   System.out.println("¡Nacieron días diferentes!");

f1 = f2;  // ¿Qué ocurre?
```

Clonación de objetos

- La asignación entre objetos es una asignación de referencias.
- Si queremos crear un objeto nuevo igual se debe clonar o copiar el objeto original
  - Método clone()
    - Crea un objeto estructuralmente igual que el original

```
Fecha f1, f2;
f2 = f1.clone(); // Copia del objeto

if (f1 == f2)    // Esto NO será cierto
  System.out.println("El mismo");

if (f1.equals(f2)) // Esto sí
  System.out.println("Iguales");
```


Tipos y paso de parámetros

- Existen dos categorías de tipos de datos:
  - Los tipos valor: las variables de estos tipos contienen directamente sus datos.
    - En Java los únicos tipos de datos de esta categoría son los conocidos como tipos primitivos: char, boolean, int, etc.
  - Los tipos referencia: las variables de estos tipos almacenan referencias a sus datos (que son normalmente objetos).
    - En Java todas las variables de clases y los arrays son tipos referencia.
    - Como su nombre indica, las variables de estos tipos referencian a una instancia del tipo al que pertenecen

Tipos y paso de parámetros

- Ejemplo tipos referencia

```
Fecha f = new Fecha(12, 10, 1492);
Fecha f2;
// ... inicializaciones varias ...
f2 = null; // OK

f2 = f;                                  // f y f2 referencian lo mismo.
f.avanzaUnDia();                         // Cambia también la instancia manejada con f2
f.getDia() == f2.getDia();  // true
```


Tipos y paso de parámetros

- En Java los arrays son tipos-referencia, por lo que deben construirse con new antes de utilizarse
- La asignación de un array a otro hace que ambas variables trabajen sobre el mismo array, por lo que el cambio en uno de ellos afecta al otro:

```
int [] v1 = new int [10];
int [] v2;
v2 = v1;
v1 [0] = 7;       // v2 [0] == 7 también
```

Tipos y paso de parámetros

- La construcción de un array bidimensional, por tanto, es en realidad un array de *referencias a arrays unidimensionales*.
- De ahí que haya que crear cada uno de forma individual:

```
int [][] m;
m = new int [3][];
m[0] = new int [2];
m[1] = new int [2];
m[2] = new int [2];
// m = new int [3] [2];   es equivalente a lo anterior
```


Tipos y paso de parámetros

- En Java los parámetros en las invocaciones a métodos se realiza siempre por valor, o lo que es lo mismo, son siempre de entrada.

```
// Este método NO funciona: los parámetros son de entrada.

public void swap(int a, int b) {
   int aux;
   aux = a;
   a = b;
   b = aux;
}
```

Tipos y paso de parámetros

- Sin embargo, la naturaleza de los tipos-referencia hacen que las operaciones realizadas sobre las variables de estos tipos dentro del método afecten a las instancias apuntadas por las variables de fuera.

```
public void hazAlgo(Fecha f) {
   f.avanzaUnDia();
}

Fecha vble = new Fecha(12, 10, 1492);
hazAlgo(vble);      // En este momento, vble.getDia() devolverá 13
```


Tipos y paso de parámetros

- Ejemplo: paso de parámetros de tipo-referencia.
- La instancia puede cambiar, pero el valor referenciado no.

```
public void hazAlgo(Fecha f) {
   f = new Fecha(1, 1, 2000);
}

Fecha vble = new Fecha(12, 10, 1492);
hazAlgo(vble);

// En este momento, vble sigue haciendo referencia a la fecha 12/10/1492.
// La función cambió la instancia a la que hacía referencia la variable f,
// pero NO la instancia a la que referenciaba vble.
```

Boxing-unboxing

- En Java para cada tipo primitivo, existe una clase que almacena uno de esos valores.
  - java.lang.Integer para almacenar un int
  - Java.lang.Float
  - …
- Los objetos de estas clases son inmutables: no se puede cambiar el valor que contiene.
- El lenguaje hace automáticamente conversiones de unos a otros, algo conocido como *boxing* y *unboxing*
- El proceso de *boxing* implica la creación de un objeto, que se realiza de forma transparente al programador.

```
Integer a;
int i = 0;
a = 3;      // Boxing: a = new Integer(3);
a = i;      // Boxing: a = new Integer(i);
i = a;      // Unboxing: i = a.intValue();
```


StringBuilder

- La clase StringBuilder permite manipular cadenas.
  - Los objetos de la clase String no se pueden modificar
- Se crea un objeto con StringBuilder()
- Luego se usan los métodos:
  - insert, append, setCharAt, ...
  - Para obtener un String hay que llamar al método toString()

```
StringBuilder holamundoBuilder = new StringBuilder();
holamundoBuilder.append("Hola, ");
holamundoBuilder.append("mundo");
String holamundo = holamundoBuilder.toString();
```