Unidad 7

Programación Orientada a Objetos

Contenidos

- **Paradigma**
  - **Tipos de paradigma**
- **Programación Orientada a Objetos (POO)**
  - **Las bases de POO**
- **Elementos de la POO**
  - **Clase y Objeto**
  - **Atributos y estado**
  - **Métodos y mensajes**
  - **Herencia y polimorfismo**
- **Beneficios de la POO**

Paradigma

Un paradigma de programación es un marco conceptual, un **conjunto de ideas** que describe una forma de entender la **construcción de un programa**.

Por tanto, un paradigma define una serie de herramientas conceptuales y unas reglas para combinarlas.

Existen lenguajes que se concentran en las ideas de un único paradigma así como hay otros que permiten la combinación de ideas provenientes de distintos paradigmas.

Tipos de paradigma

- **Paradigma imperativo**
  Sucesión de instrucciones o conjunto de sentencias
  - Programación estructurada
  - Programación procedimental
  - Programación modular
- **Paradigma declarativo**
  No se crea un algoritmo, sino que se define el problema y el mismo busca la solución mediante razonamiento lógico.
  - Programación lógica
  - Programación funcional
- **Programación orientada a objetos**
  Se construyen modelos de objetos que representan elementos (objetos) del problema a resolver, que tienen características y funciones
- **Programación reactiva**
  Se basa en escuchar lo que emite un evento o cambios en el flujo de datos, en donde los objetos reaccionan a los valores que reciben de dicho cambio.

Programación Orientada a objetos

Modelos de **objetos que representan elementos (objetos) del problema a resolver**, que tienen características y funciones.

Permite **separar los diferentes componentes de un programa**, simplificando así su creación, depuración y posteriores mejoras.

La programación orientada a objetos **disminuye los errores y promociona la reutilización del código**.

Es una manera especial de programar, que **se acerca de alguna manera a cómo expresaríamos las cosas en la vida real**.

Las bases de la POO

Abstracción: proceso mental de extracción de las características esenciales, ignorando los detalles superfluos.

Encapsulación: ocultar los detalles que dan soporte a un conjunto de características esenciales de una abstracción.
Dos partes: una visible que todos tienen acceso y se aporta la funcionalidad, y otra oculta que implementa los detalles internos.

Modularidad: descomponer un sistema en un conjunto de partes.

Jerarquía: un proceso de estructuración de varios elementos por niveles.

Elementos de la POO

- Clases y objetos
- Atributos y estado
- Métodos y mensajes
- Herencia y polimorfismo

Clase y Objeto

Un **objeto** es un trozo de software que contiene una colección de **métodos** (funciones) y campos (**atributos**).

Una **clase** es una **plantilla** que define los **métodos** y **atributos** con los que se construyen los objetos que pertenecen a dicha clase. Es decir, describe las estructuras de datos que lo forman y las funciones asociadas con él.

Un objeto es un ejemplar concreto de una clase.

Crear un objeto se llama **instanciar una clase**. Un objeto concreto que se crea de una clase se dice que es una instancia de dicha clase.

Ejemplo:

- **Alumno** es una **clase** que describe los atributos y métodos que debe tener cualquier alumno.
- **Pepe** y **María** son alumnos concretos, con sus datos propios. Por tanto, son **objetos** (instancias de la clase Alumno)

Atributos y Estado

Un **atributo** es cada uno de los **datos** que describen a una clase.

No incluyen los datos auxiliares utilizados para una implementación concreta.

El **estado** de un objeto es el conjunto de **valores de sus atributos** en un instante dado.

Métodos y Mensajes

Un **método** define una **operación sobre un objeto**.

En general, realizan dos posibles acciones: consultar el estado del objeto o modificarlo. Los métodos disponen de parámetros que permiten delimitar la acción del mismo.

Nos podemos encontrar con diversos tipos de métodos:

- **Consulta/Modificación** de un atributo (*getters & setters*)
- **Operaciones** sobre el conjunto de atributos, calculando valores o realizando modificaciones
- **Inicializan** los atributos al principio del ciclo de vida, o liberan los recursos al final del ciclo; nos referiremos a ellos como ***constructores*** o ***destructores***

Métodos y Mensajes

Un **mensaje** es la **invocación** de un **método** de un objeto.

Podemos decir que un objeto lanza un mensaje (quien realiza la invocación) y otro lo recibe (el que ejecuta el método).

Ejemplos:

- El profesor Andrés lanza el mensaje a la alumna María “dime tu edad”. María responde dando dicho dato a Andrés.
- El alumno Pepe está hablando, el profesor Andrés lanza el mensaje “calla” a Pepe, el alumno Pepe pasa al estado “callado”.

Herencia y Polimorfismo

La **herencia** es una característica que permite a las clases **definirse a partir de otras**, y así reutilizar su funcionalidad.

A la clase padre se le llama superclase, clase base..., y a la hija subclase, clase derivada....

El **polimorfismo** es la capacidad de que un mismo mensaje funcione con diferentes objetos.

Es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo concreto de objetos sobre el que se trabaja. El método opera sobre un conjunto de posibles objetos compatibles.

Beneficios de la POO

- Reutilización del código.
- Evita la duplicación de código.
- Convierte cosas complejas en estructuras simples reproducibles.
- Permite trabajar en equipo gracias al encapsulamiento ya que minimiza la posibilidad de duplicar funciones cuando varias personas trabajan sobre un mismo objeto al mismo tiempo.
- Al estar la clase bien estructurada permite la corrección de errores en varios lugares del código.
- Protege la información a través de la encapsulación, ya que solo se puede acceder a los datos del objeto a través de propiedades y métodos privados.
- La abstracción nos permite construir sistemas más complejos y de una forma más sencilla y organizada.

Unidad 7

Clases en Java

Contenidos

- Programas en Java
- Clases en Java
  - Declarar una clase
  - Crear un objeto (Instanciar)
  - Referencias a objetos
  - Recolector de basura
  - null
- Atributos
  - Acceso a los atributos
  - Errores de acceso a los atributos
- Métodos
  - Lanzar un método

Programas en Java

Un **programa** es una colección de uno o más archivos de texto que contienen clases escritas en Java, al menos una de las cuales es pública (*public*) y contiene un método llamado main() que tiene la siguiente forma:

```
public static void main(String[] args) {
    // enunciados del método
}
```

Es la entrada a nuestro programa, es decir, el primer método que se ejecutará al lanzar el programa.

Clases en Java

Una **clase** es un conjunto de atributos y métodos.

Es una categoría específica de objetos, similar a un tipo Java (como el *int*, *char*, etc.). Una clase de Java especifica el rango de valores que sus objetos pueden tener. Los valores que puede tener un objeto se llama su estado.

Las clases de Java tienen tres características esenciales que no tienen los tipos de datos primitivos:

1. Las clases pueden ser definidas por el programador
2. Los objetos de las clases pueden contener variables, incluyendo referencias a otros objetos
3. Las clases pueden contener métodos que le dan a sus objetos la habilidad de actuar

Declarar una clase

Para declarar una clase se usa la palabra reservada ***class***.

```
class NombreClase {
    // Definición de la clase
}
```

Delante de la palabra class se pueden poner **modificadores** (public, abstract, ...) que veremos más adelante.

Por ejemplo, si queremos crear la clase Persona:

```
class Persona {
    // Definición de la clase
}
```

Se recomienda usar un sustantivo descriptivo como nombre de la clase, en singular y debe empezar en mayúscula.

Crear un objeto (Instanciar)

Crear un objeto también se dice crear una **instancia de una clase**.

Para crear un objeto hacemos uso de la palabra reservada ***new***.

```
Persona p = new Persona();
```

Declaramos una variable ***p*** del tipo de datos Persona (clase que hemos definido en nuestro programa).

Creamos un objeto (instancia) nuevo de la clase Persona y se lo asignamos a la variable ***p***.

Referencias a objetos

Como en los arrays, las variables que contienen objetos, realmente son referencias a la posición de memoria donde se creó el objeto.

```
Persona pepe, maria, jose;
pepe = new Persona();
maria = new Persona();
jose = pepe;
System.out.println(pepe); // Referencia a la posición de memoria
                          // del objeto. No se ven
                          // las propiedades del objeto.
pepe == jose  // Es true, ambas variables apuntan al mismo objeto.
pepe == maria // Es false porque apuntan a objetos diferentes (aunque
              // luego sus atributos sean iguales).
```

Recolector de basura

El **recolector de basura** de Java se encarga de liberar los espacios de memoria que no están referenciados. Es decir, que hemos dejado de usar.

Por ejemplo:

```
Persona p = new Persona();     // Se crea un objeto Persona en memoria
                               // p referencia a ese objeto.
p = new Persona();              // Se crea otro objeto Persona en memoria
                               // p referencia al nuevo objeto.
                        // El objeto anterior deja de estar referenciado
                        // y se libera.
```

Por tanto, en Java no hay que preocuparse de ***destruir*** los objetos para liberar memoria.

null

La palabra reservada **null** es una **referencia nula**. Es decir, que no se apunta a ninguna posición de memoria.

Se puede poner una variable de una clase al valor null cuando no queremos que tenga ningún objeto.

Por ejemplo:

```
Persona p = new Persona();
p = null;    // Ahora p no apunta a ningún sitio y la memoria que
             // ocupaba el objeto Persona, deja de estar referenciada
             // y es liberada.
```

Atributos

Un atributo es una **información** que se almacena y es parte representativa del **estado** de un **objeto**.

Los atributos son **tipos primitivos o clases**.

```
class Persona {
     String nombre;
     int edad;
}
```

Se declaran en la clase y luego **cada instancia** (objeto) tendrá **valores propios** para dichos atributos.

Observar, que los atributos se declaran a nivel de instancia, fuera de los métodos, y por lo tanto, **son accesibles desde cualquier parte de la clase**.

Los atributos también pueden tener ***modificadores***.

Acceso a los atributos

Mientras no se indique lo contrario (con un modificador) los atributos son ***públicos*** (que es la opción por defecto). Por tanto, podemos acceder directamentes a ellos indicando la instancias y el atributo separados por un punto.

Por ejemplo:

```
Persona p = new Persona();
p.nombre = “Pepe”;              // Asigna al atributo nombre del
                                // objeto p el string “Pepe”.
System.out.print(p.nombre)      // Imprime el nombre (“Pepe”)
                                // de la instancia p.
```

Errores de acceso a los atributos.

Hay que tener mucho cuidado de tratar de acceder a atributos o métodos de objetos que no han sido **inicializados** o que apuntan a **null**.

Por ejemplo:

```
Persona p1;
p1.nombre;                          // ERROR

Persona p2 = null;
p2.nombre;                          // ERROR
```

Métodos

Son las **acciones u operaciones que pueden realizar los objetos** pertenecientes a una clase concreta.

Son **funciones** que se implementan **dentro de una clase**.

Por ejemplo:

```
class Persona {
     String nombre;
     int edad;

     void saludar() {
               System.out.println(“Hola, soy “ + nombre);
     }
}
```

Los métodos también aceptan ***modificadores*** que veremos más adelante.

Lanzar un Método

Todas las instancias de esa clase, podrán ejecutar los métodos definidos en ella.

Por ejemplo:

```
Persona p = new Persona();
p.nombre = “Pepe”;
p.saludar();  // Se lanza el método saludar de la clase
              // Persona pero con los datos de la
              // instancia p. Por eso, se muestra:
              // “Hola, soy Pepe”.
```

Unidad 7

Ámbito, this, constructores y paquetes

Contenidos

Inicializar atributos

Se puede asignar un valor por defecto a los atributos de una clase.

```
class Persona {
     String nombre = "Pepe";
     int edad = 18;
}
```

La inicialización también se puede hacer en el constructor como veremos más adelante.

Ámbito de las variables y atributos

Las **variables** declaradas **dentro** de los **métodos**, tienen un **ámbito local** al método o al **bloque de código** donde son declaradas (if, while, for,...).

También los **parámetros** de un **método**, como las funciones, tienen al ámbito del propio método.

Pero los **atributos**, al estar declarados fuera de los métodos, en la raíz de la clase, tienen un **ámbito global a toda la clase**, es decir, se pueden usar desde todos los métodos.

Variables locales vs Atributos

Cuando dentro de un método hay un parámetro o se declara una variable local que tiene el mismo identificador que un atributo, el atributo queda oculto al solaparse con un identificador más local. Por tanto, con ese identificador accederemos al parámetro o la variable local y no al atributo.

Una variable local que oculta el atributo:

```
class Persona {
    String nombre = "Pepe";

    String saludar() {
       String nombre = "Maria";
       return "Hola, me llamo " + nombre;
    }
}
```

Una parámetro que oculta el atributo:

```
class Persona {
   String nombre = "Pepe";

   String saludar(nombre) {
      return "Hola, me llamo " + nombre;
   }
}
```

Objeto this

Dentro de la implementación de un método, a veces se necesita referenciar a la propia instancia a la cual pertenece. Para ello está la palabra reservada **this**.

"this" quiere decir **éste** (objeto).

De esta forma ya podemos buscar los atributos del objeto aunque tengamos parámetros y variables locales que los oculten.

```
class Persona {
   String nombre = "Pepe";

   void setNombre(nombre) {
      this.nombre = nombre; // Actualiza el atributo nombre con el nuevo nombre, que se pasa
   }
}
```

Constructores

Los constructores son métodos especiales que reúnen las tareas de inicialización de los objetos de una clase; por lo tanto, el constructor establece el estado inicial de todos los objetos que se instancian.

Cuando se crea una instancia se llama de forma implícita al constructor.

No es obligatorio definir constructores, si no se realiza, existe un constructor por defecto sin parámetros.

El constructor **debe llamarse igual que la clase**. Aunque devuelve un objeto de esa clase, no hay que indicarlo en el método (está implícito).

Ejemplo de Constructores

El constructor implícito es:

```
Persona() {}
```

Si no tiene nada ni inicializa atributos, no hay que definirlo porque está implícito.

Pero si tengo otros constructores creados, éste deja de estar implícito y tengo que declararlo si quiero construir objetos sin pasarle parámetros.

A veces podemos crearlo para iniciar variables:

```
Persona() {
        this.nombre = “Pepe”;
        this.edad = 18;
}
```

También podemos crear constructores que aceptan parámetros para iniciar los atributos con estos parámetros.

```
Persona(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
}
```

En este caso, se crearán las instancias de la forma:

```
Persona p = new Persona(“María”, 20);
```

Sobrecarga

La sobrecarga es **definir dos o más métodos con el mismo nombre, pero con parámetros diferentes por cantidad o tipo**.

El objetivo de la sobrecarga es reducir el número de identificadores distintos para una misma acción pero con matices que la diferencian.

La sobrecarga se puede realizar tanto en **métodos generales**, como en **constructores**.

La sobrecarga es un polimorfismo estático, ya que es el compilador quien resuelve el conflicto del método a referenciar.

Sobrecarga en constructores

Cuando tenemos varios constructores en la misma clase, es habitual que unos llamen a otros para reutilizar código. La forma de llamar a esos constructores es usando **this()**.

```
Persona() {
            this(“Anonimo”, 18);      // Llama al otro constructor.
}

Persona(nombre, edad) {
            this.nombre = nombre;
            this.edad = edad;
}
```

Paquetes

Los paquetes **agrupan un conjunto de clases** que trabajan conjuntamente sobre el mismo ámbito.

Es una facilidad ofrecida por Java para agrupar sintácticamente clases que van juntas conceptualmente y definir un alto nivel de **protección para los atributos y los métodos**.

La **ventaja** del uso de paquetes es que las **clases** quedan **ordenadas** y **no hay colisión de nombres**. Si dos programadores llaman igual a sus clases y luego hay que juntar el código de ambos, basta explicitar a qué paquete nos referimos en cada caso.

Nombre y almacenamiento

La forma de nombrar los paquetes es con palabras (normalmente en minúsculas) separadas por puntos, estableciendo una jerarquía de paquetes.

> La jerarquía de paquetes es independiente de la jerarquía de clases.

Los archivos que contienen las clases deben almacenarse en una estructura de carpetas que coincidan con la estructura de paquetes.

Referencias a clases o importar paquetes

Para referenciar cualquier clase en Java, se debe poner el paquete al que pertenece

Ej: ***java.lang.***`String`

Aunque no es necesario porque el paquete *java.lang* se importa de forma automática

Para evitar poner siempre la jerarquía de paquetes, se puede utilizar la sentencia ***import***, con ello se puede importar **un paquete entero** (pero no los paquetes inferiores) o una clase.

Ej:

```
import java.util.Arrays;
import java.util.Scanner;
```

Archivo fuente

Un archivo fuente (.java) que se guarda en un paquete, contiene los siguientes elementos:

- ***`package nombre_del_paquete`*** que indica a que paquete pertenece.
- Una serie opcional de importaciones de clases definidas en otros paquetes (***`import`***).
- La **definición** de una o más **clases** (solo una puede ser pública por archivo). Aunque es recomendable que cada clase esté en un único archivo.

Unidad 7

Modificadores de acceso

Contenidos

- **Modificadores**
  - **Palabras reservadas modificadoras**
  - **Modificadores de acceso**
  - **Modificadores sin acceso**
  - **Static**

Modificadores

Son palabras clave que nos permiten **ajustar el acceso a nuestra clase** y sus miembros (atributos y métodos), su alcance y comportamiento en determinadas situaciones.

Por ejemplo, podemos controlar qué **clases / objetos** pueden acceder a ciertos miembros de nuestra clase, si una clase puede ser heredada o no, si deberíamos crear un método más tarde, si podemos crear instancias de una clase, etc..

Palabras reservadas modificadoras

Se escriben antes del atributo / método / clase.
Por ejemplo:

```
abstract public class Cuenta {...
    private double saldo;
    public String toString() {...
```

Los modificadores en Java se dividen en uno de dos grupos:

- **Acceso**: *public, private, protected*
- **Sin acceso**: *static, final, abstract, synchronized, volatile, transient, native*

Modificadores de acceso

Los modificadores de acceso se ocupan de la **visibilidad** de los miembros de la clase.

Controlan si otras clases pueden **ver o cambiar ciertos atributos** o **lanzar métodos** de nuestra clase.

Estos tipos de modificadores están estrechamente relacionados con una parte importante de la programación orientada a objetos llamada **encapsulamiento**.

Como recordatorio, la encapsulación es una idea que vincula los datos con el código que los manipula. Al controlar el acceso, puede evitar el uso indebido.

Modificadores de acceso

public	se puede acceder al miembro desde cualquier lugar
protected	el miembro solo es inaccesible desde una clase en otro paquete diferente siempre que no sea subclase
predeterminado	(paquete privado) también conocido como package acceso, el miembro puede ser accedido por cualquier clase dentro del mismo paquete
private	el miembro solo puede ser accedido por otros miembros dentro de la misma clase

Modificadores de acceso

	private	default	protected	public
Misma clase	**Si**	**Si**	**Si**	**Si**
Subclase (mismo paquete)	**No**	**Si**	**Si**	**Si**
No subclase (mismo paquete)	**No**	**Si**	**Si**	**Si**
Subclase (paquete diferente)	**No**	**No**	**Si**	**Si**
No subclase (paquete diferente)	**No**	**No**	**No**	**Si**

Modificadores sin acceso

static	Indica que pertenece a la clase y no a la instancia.
final	Los atributos no pueden ser cambiados una vez asignados. Los métodos no pueden ser sobreescritos.
abstract	En un método, se tiene que implementar en una subclase. En una clase, no se pueden crear instancias de esta clase.
	—Modificadores más avanzados y especializados—
synchronized	Controla el acceso de subprocesos a un bloque/método.
volatile	El valor de la variable siempre se lee desde la memoria principal, no desde la memoria de un subproceso específico.
transient	El miembro se omite al serializar un objeto.

Static

Indica que el atributo o método es estático, es decir, que pertenecen a la clase y no a una instancia de clase.

Por tanto, no hace falta crear una instancia para acceder a dicho atributo o método sino que se usa la propia clase.

Ejemplo: ***Clase.atributo*** o ***Clase.metodo()***

Los atributos estáticos se suelen usar par definir constantes o variables comunes a todas las instancias.

Los métodos estáticos se usan para realizar funciones que son independientes de los atributos de instancia ya que dichos métodos no pueden acceder a ellos.

También se usan para manipular (copiar, comparar, modificar) varias instancias de la clase.

Unidad 7

Métodos especiales

Contenidos

- **Problema de atributos accesibles**
- **Métodos get/set**
- **Métodos heredados de clase**
- **toString()**
- **equals()**
- **clone()**

Problema de atributos accesibles

Cuando un atributo puede ser accedido/modificado directamente, podemos encontrarnos con los inconvenientes de que pueden tomar valores que no están permitidos en nuestro ámbito de aplicación.

Por ejemplo, un valor negativo en la edad.

El programador que implementa esta parte, puede tenerlo presente y evitarlo, pero otro programador que realiza otra parte de la aplicación y que usa dicha clase, puede no tenerlo presente y cometer errores.

Métodos get/set

Por tanto, es preferible encapsular dichos atributos (hacerlos inaccesibles) y permitirles solo su uso a través de los métodos get/set donde controlaremos su comportamiento.

Podemos controlar si es visible, con el get e incluso bajo qué condiciones.

También podemos evitar que se modifique un atributo (haciéndolo de solo lectura si implementamos solo su método get y no el método set).

En el set, podemos comprobar qué valor se quiere poner a dicho atributo y si no cumple las condiciones necesarias, modificar el valor o directamente anular dicha modificación

Métodos get/set

Ejemplo get/set

Netbeans tiene una ayuda para crear los get y los set de nuestros atributos.

- Botón derecho sobre el código
- Refactor >
- Encapsulate Fields

En el ejemplo hemos puesto el nombre y la edad como inaccesibles directamente.

El nombre es de sólo lectura. Una vez creado no se puede modificar. Pero se puede leer por *getNombre()*.

En cambio la edad es de lectura (*getEdad*) y escritura (*setEdad*), pero controlando que no pueda haber una edad negativa.

```
public class Persona {

    private String nombre;
    private int edad = 0;

    public Persona(String nombre, int edad)
{
        this.nombre = nombre;
        setEdad(edad);
    }

    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        if (edad>=0) this.edad = edad;
    }
}
```

Métodos heredados de clase

Se trata de métodos especiales que debe incluir toda clase. Cada uno tiene un propósito diferente.

- **getClass:** obtiene la clase de un objeto en tiempo de ejecución
- **toString:** devuelve una cadena de texto que describe el objeto
- **equals:** determina si un objeto es igual a otro
- **clone:** se crea un nuevo objeto que es el mismo que el objeto que se está clonando (es decir, los mismos atributos).
- ...

toString()

toString() proporciona la representación String de un Objeto y se usa para convertir un objeto a Cadena (String).

El método predeterminado **toString()** para la clase Object devuelve una cadena que consiste en el nombre de la clase de la cual el objeto es una instancia, el carácter arroba '@' y la representación hexadecimal sin signo del código hash del objeto.

Pero esto lo podemos modificar (sobreescribir) para que se muestre como deseamos.

toString()

Ejemplo

Dada la clase Persona vista anteriormente, si la usamos de la siguiente forma:

```
public static void main(String[] args) {
    Persona p = new Persona("Pepe", 18);
    System.out.println(p);
}
```

Obtenemos una salida similar a ésta:

```
persona.Persona@123772c4
```

Que no nos muestra los datos que nos gustaría

Pero si sobreescribimos el método toString() que viene por defecto en las Clases y lo hacemos a nuestro gusto dentro de la Clase Persona:

```
public String toString() {
  return String.format("%s tiene %d años",
                nombre, edad);
}
```

En este caso, se obtiene el siguiente resultado:

```
Pepe tiene 18 años
```

Equals()

No se debe usar el operador == para comparar objetos (ni String, ni arrays) ya que esta operación sólo comparará las referencias a dichos objetos, devolviendo true si se trata de dos referencias al mismo objeto, o false en caso contrario.

Por tanto, se debe usar el método equals. Pero dicho método hay que definirlo. Porque sino dará falso.

Equals()

Ejemplo

Probando el siguiente código:

```
Persona p1 = new Persona("Pepe", 18);
Persona p2 = new Persona("Pepe", 18);
System.out.println(p1 == p2);       // false
System.out.println(p1.equals(p2));  // false
```

Siempre da falso porque no hemos definido el método equals en la clase Persona.

Definiendo el método equals en Persona:

```
public boolean equals(Persona p) {
   return this.nombre.equals(p.nombre)
     && this.edad == p.edad;
}
```

Ahora el código anterior da:

```
Persona p1 = new Persona("Pepe", 18);
Persona p2 = new Persona("Pepe", 18);
System.out.println(p1 == p2);  // false
System.out.println(p1.equals(p2));  // true
```

Clone()

También hemos visto que una expresión de la forma p1=p2; con p1, p2 objetos no hace que p2 sea una copia de p1, sino una nueva referencia a p1.

Para conseguir crear nuevos objetos que sean copias de objetos ya existentes debemos definir y utilizar el método clone. Una posible definición de clone para la clase Persona sería:

```
public Persona clone() {
    return new Persona(this.nombre, this.edad);
}
```

Unidad 7

Enumerados

Contenidos

- **Enumerados**
- **Uso de los enumerados**
- **Definición dentro de una clase**

Enumerados

Un enumerado (o Enum) es una clase "especial" (tanto en Java como en otros lenguajes) ***que limitan la creación de objetos a los especificados explícitamente en la implementación de la clase.***

La única limitación que tienen los enumerados respecto a una clase normal es que si tiene constructor, este debe de ser privado para que no se puedan crear nuevos objetos.

Enumerados

Ejemplo

Por ejemplo un tipo enumerado pueden ser los días de la semana, y se definirían de la siguiente forma:

```
enum DiaDeLaSemana {
        LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}
```

Un programa puede acceder a él de la forma:

```
        DiaDeLaSemana.Lunes;
```

Uso de los enumerados

Los enumerados se pueden asignar de la siguiente forma:

```
DiaDeLaSemana navidad = DiaDeLaSemana.JUEVES;
```

También se puede convertir un String de la siguiente forma

```
DiaDeLaSemana navidad = DiaDeLaSemana.valueOf(“JUEVES”);
```

Definición dentro de una clase

Se puede definir fuera de la clase y luego usarlo en los atributos.

```
enum DiaDeLaSemana {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO}

Class Festivo {
          String nombre;
          DiaDeLaSemana dia;
}
```

O dentro de la propia clase

```
Class Festivo {
  enum DiaDeLaSemana {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO}
  DiaDeLaSemana dia;
}
```