



Clases, objetos y métodos

Índice



Clases, objetos y métodos

1 Principios de la ordenación de objetos	3
2 Definición de clases: datos y métodos	9
3 Crear objetos: operador new	13
4 Constructores y sobrecarga de constructores	15
5 La referencia this	17
6 Variables y métodos de clase: miembros static	19
7 Sobrecarga de métodos	21
8 Métodos "getter" y "setter"	23
9 El método toString	26
10 El método equals	28
11 Garbage collector y el método finalize	30
12 Clases anidadas e internas	33

1. Principios de la ordenación de objetos

Java se basa en la programación orientada a objetos. De alguna manera todos los programas en Java son orientados a objetos. Los realizados hasta este momento necesitan una clase que implemente un método como main para empezar la ejecución, aunque no se necesite la creación de ningún objeto se ha estado utilizando el concepto de clase.

Un programa orientado a objetos es el resultado de la colaboración de varios tipos de objetos entre sí, cada uno con sus funcionalidades prestando o solicitando servicios a otros. Un objeto que no es utilizado por otros o que no utiliza a otros, no tiene sentido.

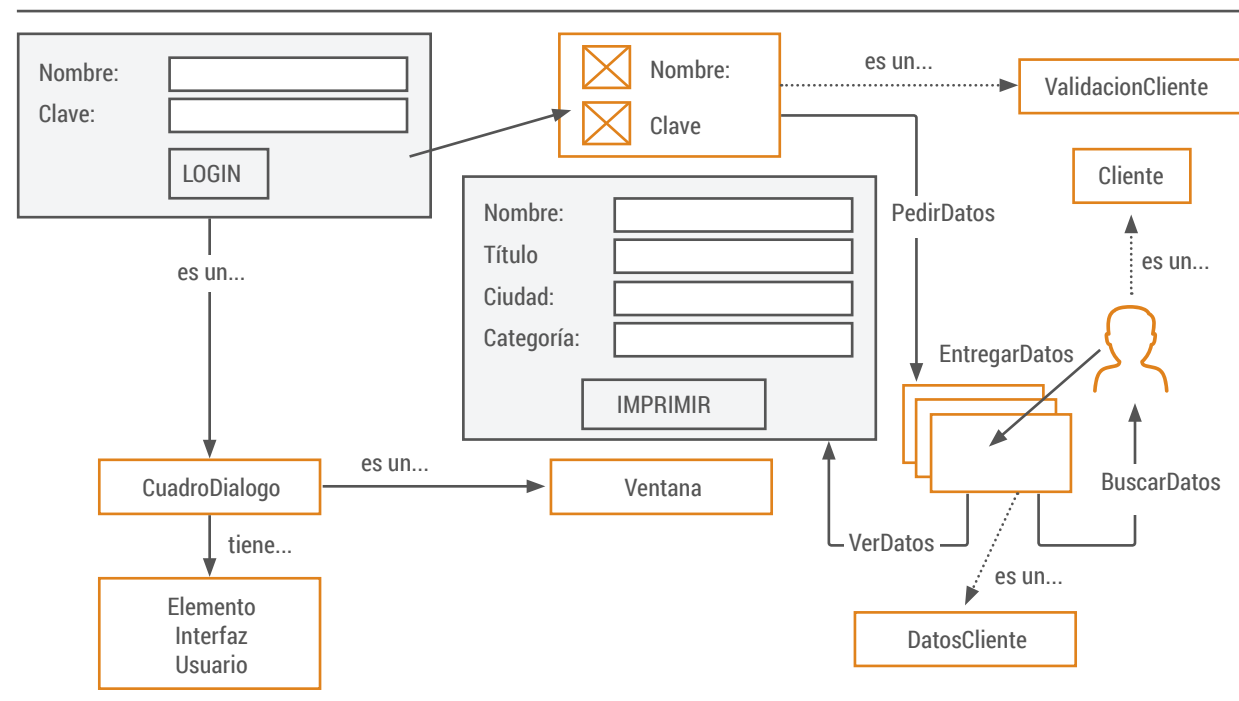


IMAGEN 5.1: POO, OBJETOS QUE COLABORAN ENTRE SÍ.

Los objetos son concreciones (realidades) instanciadas de las clases. Estas son abstracciones que representan como son los objetos de dicha clase y cuáles son sus funcionalidades.

Para poder implementar la programación orientada a objetos todos los lenguajes que se basan en esta metodología se basan en cuatro principios: **abstracción**, **encapsulación**, **herencia** y **polimorfismo**.

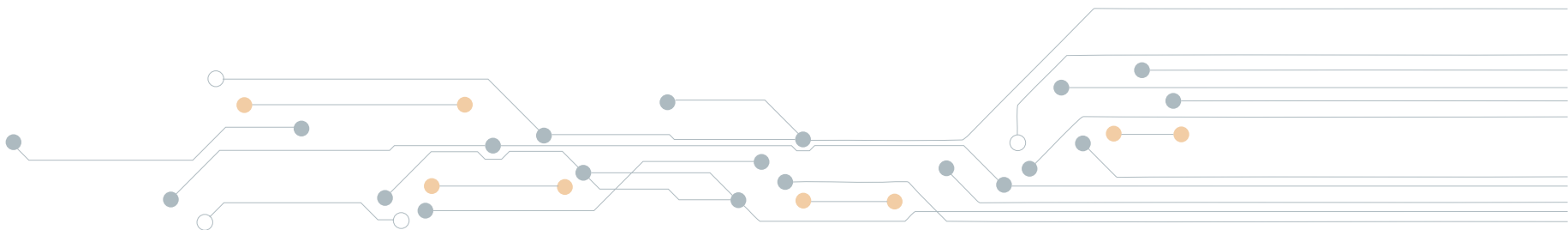
- **Abstracción:** Es el mecanismo que permite concentrarse en los aspectos que se necesitan, obviando los que no son necesarios. Las clases son el mecanismo para implementar la abstracción en Java, representan lo que los objetos de esa clase tienen y saben hacer, agrupan lo común en representación y funcionalidad de los objetos. Una clase es una abstracción para un tipo concreto y determinado de objetos.



Clase Empleado

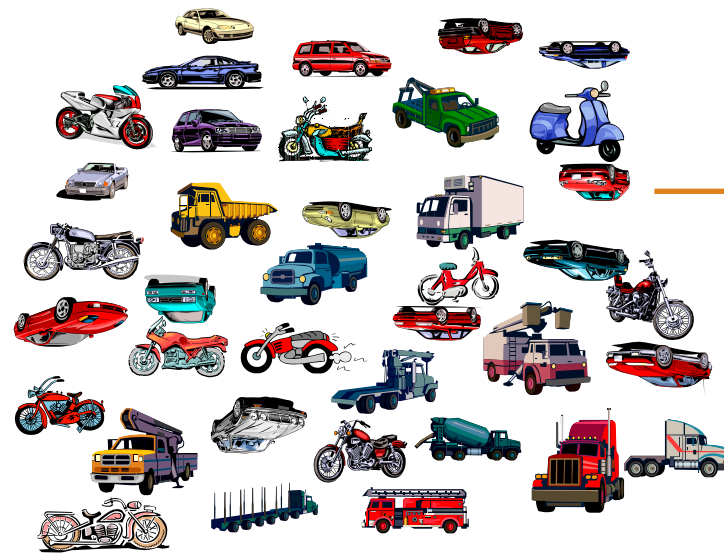
Define objetos personas, con un oficio que prestan sus servicios en una empresa

IMAGEN 5.2: LAS CLASES SON LAS ABSTRACCIONES.



Un ejemplo: Cuando utilizamos un coche de una marca, modelo y matrícula concreto, estamos usando un objeto cuyas características y funcionalidades son las que se establecen en la abstracción que representa la clase de coche que es. Si subimos un poco más en el nivel de abstracción, no necesitamos conocer ni la marca, ni el modelo de un coche para reconocer que es un coche, conocemos la abstracción coche y por tanto reconocemos todas las concreciones que de esa abstracción identifiquemos.

La abstracción es la facultad de los seres humanos en concentrar el pensamiento en un elemento, característica, propiedad o noción, centrándose en él y olvidando todos los demás



ABSTRACCIÓN

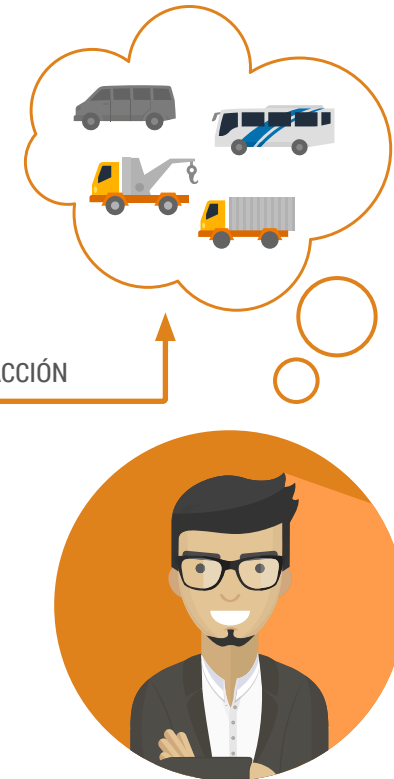


IMAGEN 5.3: EJEMPLO DE ABSTRACCIÓN.

- **Encapsulación:** Es el mecanismo por el cual los datos y los códigos están protegidos de su acceso desde el exterior; es decir por parte de otros objetos que necesitan su colaboración. Un objeto sólo expone las funcionalidades expresadas en su interfaz público y ocultan tanto su código como su representación. La clase con sus diferentes tipos de acceso es quien implementa la encapsulación.

La encapsulación consiste en ocultar los detalles de realización de una clase

Número de serie
Depósitos de ingredientes
Depósitos de monedas
Dispositivos de preparación de bebidas
Tubos y tuberías de entrada de agua
Tubos y tuberías de salida de bebidas
Dispositivos de reconocimiento de monedas
Dispositivos de cálculo de precios
Dispositivos de cálculo de cantidades
.....

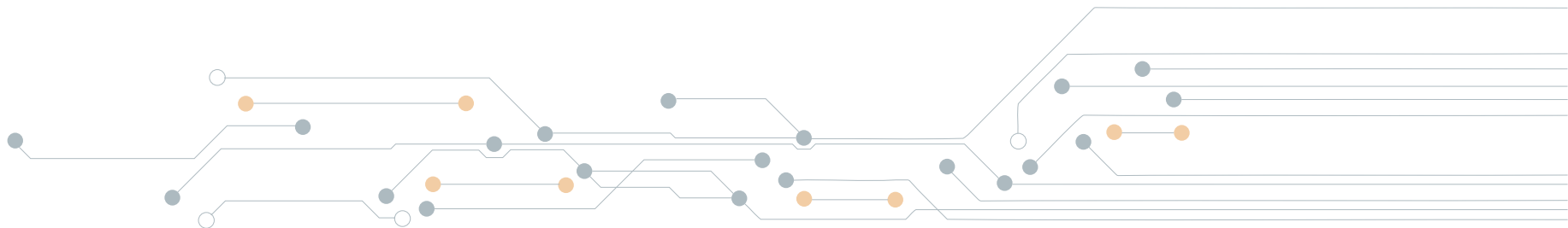


IMAGEN 5.4: ENCAPSULACIÓN

Un ejemplo: Para poder conducir un coche todos sabemos que tenemos que interactuar con una serie de elementos y dispositivos, pero no necesitamos saber cómo funcionan ni como están hechos, sólo necesitamos acceder a ellos y utilizarlos según el protocolo de uso de ese elemento o dispositivo. Pisamos el freno y sabemos que el coche para, todo lo que hay más allá del pedal del freno es oculto para quien lo utiliza, sabemos el efecto que produce y no necesitamos saber más.



IMAGEN 5.5: ENCAPSULACIÓN, PEDALES COCHE.



- **Herencia:** Es el mecanismo por el cual un objeto puede adquirir propiedades y funcionalidades de otro tipo de objetos. Es un mecanismo de generalización de propiedades y funcionalidades para las clases de las que se hereda (superclases) y de especialización de las clases que heredan (subclases).

El concepto de herencia implica una clasificación jerárquica, de tal forma que si no se usara en la orientación objetos, cada clase diferente de objetos tendría que definir explícitamente todas sus propiedades y funcionalidades. Por tanto la herencia permite que un objeto sea una instancia específica de un caso más general.

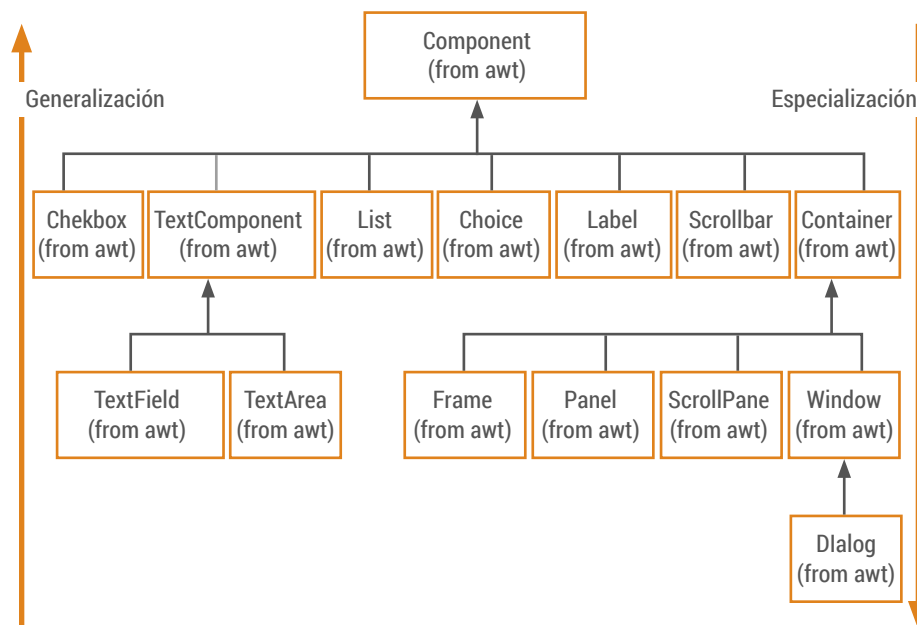


IMAGEN 5.6: HERENCIA, JERARQUÍA, GENERALIZACIÓN, ESPECIALIZACIÓN.

Por ejemplo un coche deportivos es un tipo de objeto especializado a partir de la clase más genérica coche, así hay coches que son deportivos y otros que son 4*4, los dos son coches, los dos heredan, de la clase más general, propiedades y funcionalidades, pero uno se especializa en la velocidad y el otro en la fuerza.

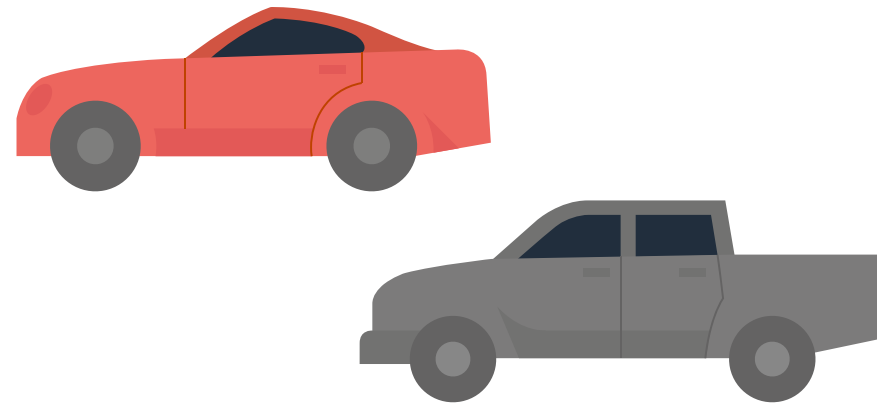


IMAGEN 5.6: HERENCIA, TIPOS DIFERENTES DE COCHES.

- **Polimorfismo:** Es el mecanismo por el cual con la misma forma se ejecutan códigos diferentes. En una jerarquía de herencia de clases, se implementa una funcionalidad determinada en las superclases (de las que se hereda) y se va especializando su funcionamiento de acuerdo a las subclases (que heredan dicha funcionalidad).

Se puede expresar también como “una interfaz, múltiples métodos”. Se puede diseñar una interfaz genérica para un grupo determinado de funcionalidades relacionadas y por tanto usar esa misma interfaz para diferentes grupos de objetos que implementan esas mismas funcionalidades, pero cada grupo de una forma diferente.

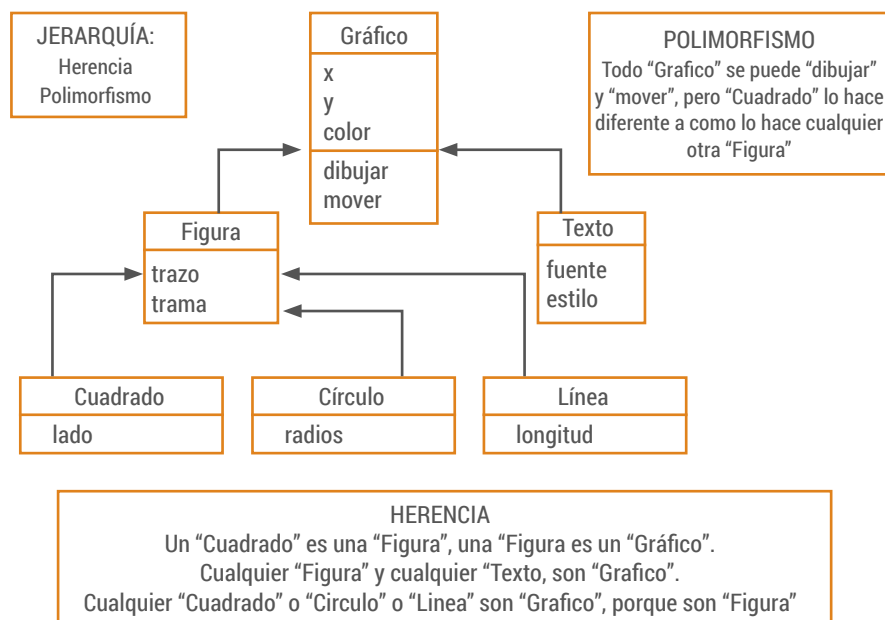


IMAGEN 5.7: POLIMORFISMO Y HERENCIA.

Por ejemplo, en cualquier coche se tiene la interfaz pedal acelerador, pedal embrague y pedal freno, pero sin embargo cada modelo de cada marca en particular ejecuta la funcionalidad de cada uno de esos pedales de acuerdo a su tecnología empleada. En todos la forma de usarlos y el efecto es el mismo, pero en cada tipo de coche el fabricante emplea una tecnología y variantes diferentes, no es lo mismo el proceso de aceleración en un coche tipo gasoil, que en un coche tipo gasolina. No es lo mismo frenar en un coche con dispositivo de frenos de disco, que de frenos de tambor, que de frenos ABS.

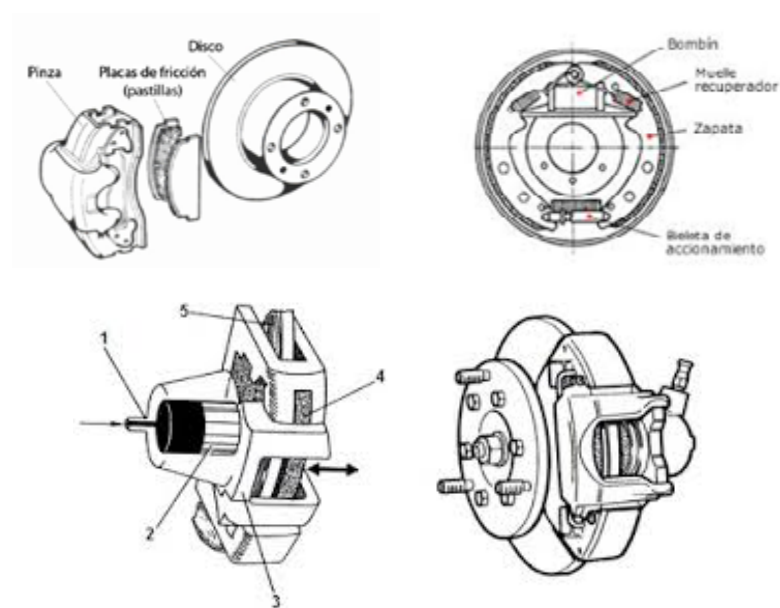


IMAGEN 5.8: POLIMORFISMO, FRENAR CON DIFERENTES TECNOLOGÍAS.

2. Definición de clases: datos y métodos

Las clases definen la estructura y el comportamiento de los objetos de un tipo particular. Podemos decir que las clases son en realidad modelos de objetos del mismo tipo. En Java las clases se implementan con la palabra reservada **class**. Una class es un bloque { } en el que se definen las propiedades o atributos (variables) y los métodos o funcionalidades (**funciones**).

Un formato genérico de definición de una **class** en Java es como se muestra a continuación:

```
<modificador_acceso> class <identificador>{  
    <definicion_variables>  
    <definicion_metodos>
```

Las **variables** miembro dentro de la definición de una clase son de dos tipos:

- **de instancia:** son los atributos que tendrá cada objeto, son espacios de memoria en los que se almacenan los datos de cada objeto. Se crean cuando se crea el objeto y se liberan cuando se liberan los objetos.
- **de clase:** son atributos que comparten todos los objetos de una clase, no pertenecen a ningún objeto son de la clase. Se definen utilizando la palabra **static**.



La definición de cualquiera de los dos tipos es casi como se ha visto hasta este momento para la definición de variables dentro de la función main, añadiendo el modificador de acceso a dicha variable, que se define con las palabras clave private, protected, public o sin ninguna de ellas, tal y como sigue:

```
<modificador_acceso> [static] <tipo> <identificador>;
0
<modificador_acceso> [static] <tipo> <identificador> =
<expresion_inicializacion>
```

Los **métodos** son las funciones miembro de la clase que van a definir el comportamiento de los objetos y como las variables son de dos tipos:

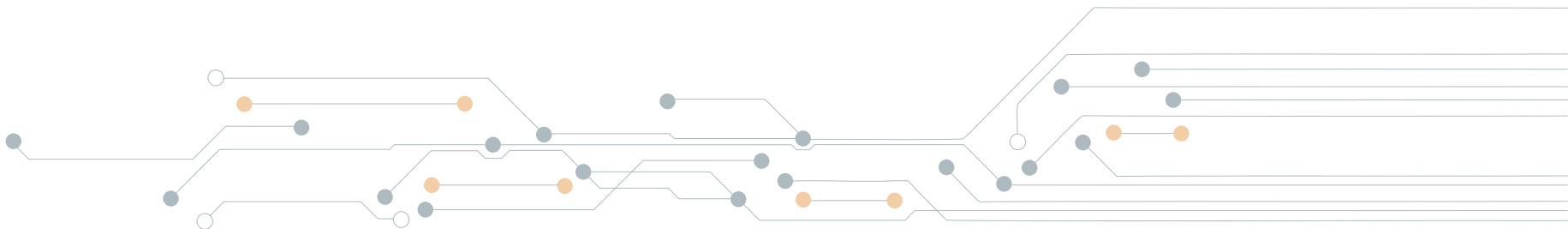
- **de instancia:** son las funciones miembro que definen realmente el comportamiento de los objetos. Son llamadas para un objeto en concreto y en su código pueden acceder a las variables (atributos) del objeto implementando la funcionalidad que las corresponda.
- **de clase:** son funciones de las clases, no de los objetos y por tanto sólo pueden acceder a sus variables de clase. Se definen como las funciones anteriores pero utilizando la palabra **static**.

La definición de una función miembro de una clase es como sigue:

```
<modificador_acceso> <tipo> <identificador> ([<lista_
parametros>]){
    <sentencias>
}
```

El valor devuelto **<tipo>** puede ser **void**, cualquier tipo primitivo del lenguaje o una referencia a cualquier tipo de objeto (cualquier identificador de clase).

El ámbito tanto en la clase, como en las variables, como en los métodos define quien puede acceder a la clase, a la variable o al método.



En orientación a objetos para implementar la encapsulación hay que definir los datos (variables) en la parte privada, para así ocultar la representación. Y exponer en una parte pública las funcionalidades (métodos o funciones), para así definir el interfaz de uso de los objetos de una clase.

En Java existen cuatro modificadores de acceso para definir el ámbito desde el que puede ser usadas las clases, las variables y los métodos, tal y como e indica en la tabla siguiente:

Modificadores de acceso	Accesibilidad		
	Clase	Variable	Método
private	Accesible sólo dentro del archivo o clase donde se define	Accesible dentro de la clase	
protected	No se aplica	Accesible dentro de la propia clase y funciones de clases que hereden (subclases)	
sin modificador	Accesible dentro de la propia clase y las clases que estén dentro de su mismo paquete (package)		
public	Accesible dentro de la propia clase y de las que en su archivo la importen (import)		

Tabla 5.1: Modificadores de acceso y ámbitos.

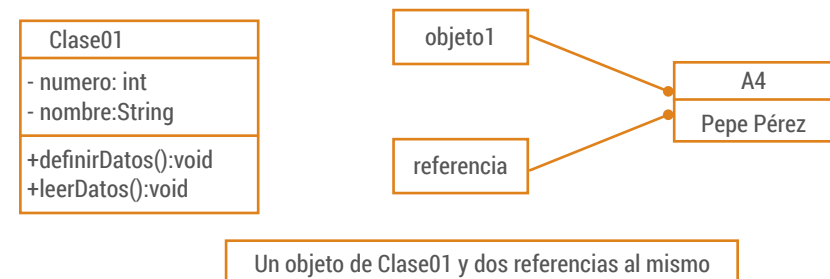


Ejemplo:

```
package com.juan.clases01;    //paquete donde se
                             encuentra Clase01.java
import java.util.Scanner;    //para utilizar Scanner
public class Clase01 {
    private int numero; //variable de instancia privada
    private String nombre;    //variable de instancia
                             privada
    public void definirDatos(){ //método de acceso publico
        Scanner teclado = new Scanner(System.in);
        System.out.println("Teclea un nombre: ");
        nombre = teclado.nextLine();
        System.out.println("Teclea un numero: ");
        numero = teclado.nextInt();
    }

    public void leerDatos(){    //método de acceso
                             publico
        System.out.println("Nombre: " + nombre);
        System.out.println("Numero: " + numero);
    }
}
```

En la imagen se muestra el resultado de ejecutar un código que crea un objeto de la clase anterior, referenciándolo dos veces con dos identificadores diferentes. La clase esta representada tal y como se hace en UML (lenguaje de modelado universal).



```
package com.juan.clases01;
import com.juan.clases01.Clase01;
public class Programa {
    public static void main(String[] args) {
        Clase01 objeto1= new Clase01();
        objeto1.definirDatos();
        objeto1.leerDatos();
        Clase01 referencia=objeto1;
        referencia.leerDatos();
    }
}
```

IMAGEN 5.9: EJEMPLO DE USO DE OBJETOS DE UNA CLASE SENCILLA

3. Crear objetos: operador new

Para poder crear objetos de una clase se utiliza el operador new. Este operador devuelve una referencia al objeto instanciado.

Las variables de instancia no hace falta inicializarlas, cuando se crea un objeto, el valor que se almacene en el espacio de dicha variable es 0 para los números, false para los booleanos y null si son referencias a otros tipos de objetos (por ejemplo String).

Cuando se crea el objeto se invoca a una de sus funciones constructoras. El objetivo de estas funciones es inicializar con valores determinados las variables miembro de la clase.

Si en una clase no se ha definido ningún constructor, es como si Java pusiese uno que no hace nada. Esto es así, porque a la derecha del operador **new** siempre hay especificar el nombre de la clase y entre paréntesis la lista de parámetros de uno de los constructores definidos, al no tener ninguno hay que especificar el nombre de la clase y los paréntesis, tal y como se aprecia en la figura anterior.

El formato genérico de utilización del operador new es el siguiente:

```
<nombre_clase> <identificador > = new <nombre_clase>
(<lista_parametros>);
```

El espacio que ocupa un objeto es liberado por el **recolector de basura (Garbage Collector)** en uno de los momentos siguientes:

- Cuando acaba el ámbito en el que fue creado, lo que es lo mismo que decir que se llega a la llave final "}" del bloque en el que se creó el objeto.
- Cuando un objeto deja de estar referenciado. Esta situación puede producir errores si después de esta situación se intenta utilizar la referencia que antes tuvo el objeto, ahora tendría un valor null y produce la excepción **NullPointerException**.

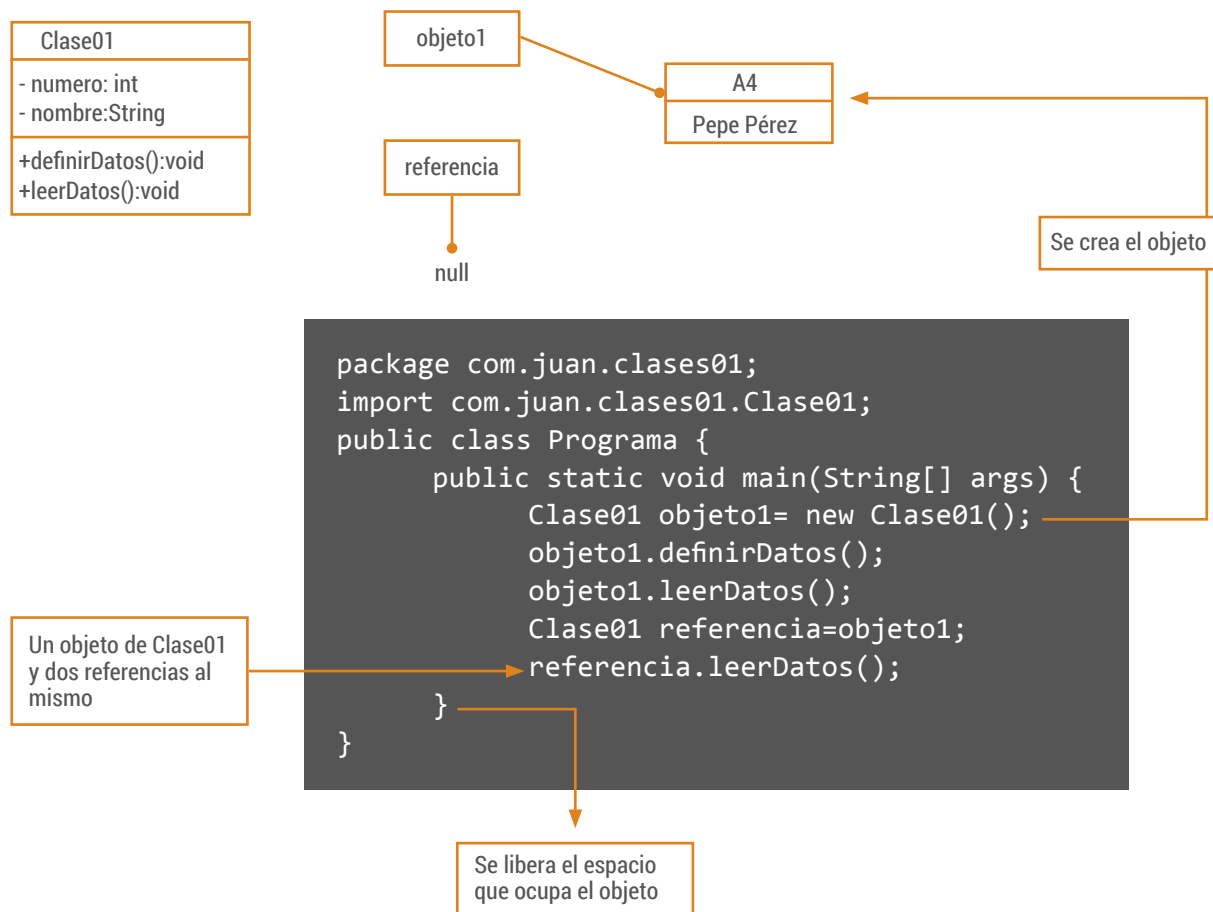
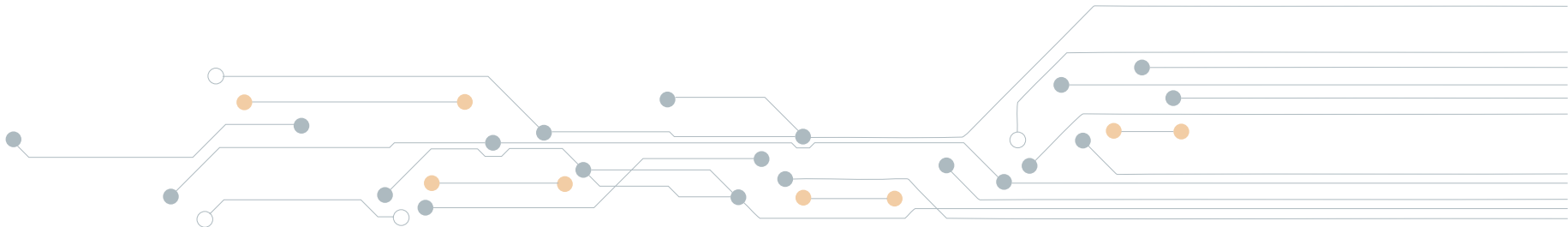


IMAGEN 5.9: EJEMPLO DE USO DE OBJETOS DE UNA CLASE SENCILLA



4. Constructores y sobrecarga de constructores

Los constructores son funciones que se ejecutan cuando se crean los objetos. Cuando se crea un objeto con `new`, primero se reserva el espacio para almacenar los valores de cada uno de sus tributos (variables de instancia) y a continuación el código de un constructor definido en la clase. Las funciones constructoras tienen el mismo nombre que la clase, una lista de parámetros que puede estar vacía y no devuelven nada (ni `void`). Son las únicas funciones en Java que no devuelven ningún valor.

En una clase se pueden definir tantos constructores como se necesiten. Cada constructor representa una forma distinta de crear los objetos de una clase.

Cuando una clase tiene varios constructores se dice que estas funciones constructoras están sobrecargadas. La sobrecarga es el mecanismo por el cual en una clase pueden existir varias funciones que teniendo el mismo nombre se diferencian en el número y/o el tipo de los parámetros. Este es el caso de los constructores, todos tienen el mismo nombre, pero se diferencian en la lista de parámetros.



En el código siguiente se muestra una clase con tres constructores:

```
package com.juan.clases;

public class Clase02 {
    private int numero;
    private String nombre;

    //Constructor sin parámetros
    public Clase02 (){
        numero = 99;
        nombre ="anonimo";
    }
    //Constructor con dos parámetros
    public Clase02(int num, String cad){
        numero = num;
        nombre = cad;
    }
    //Constructor que recibe referencia objeto de
    Clase02
    public Clase02(Clase02 o){
        numero= o.numero;
        nombre= o.nombre;
    }

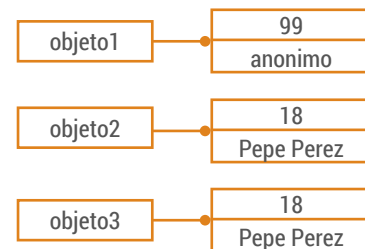
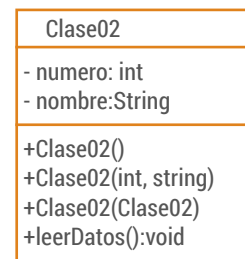
    public void leerDatos(){
        System.out.println("Nombre: " +
```

```
nombre);
        System.out.println("Numero: " +
        numero);
    }
}
```

El objetivo principal de los constructores es inicializar de un modo determinado las variables de instancia para el objeto que se crea. En los códigos típicos de las funciones constructoras se asignan valores a las variables de la clase.

Tal y como se comento anteriormente, si una clase no define ningún constructor, es como si Java ejecutase el constructor sin parámetros y sin código alguno. Pero es muy importante recordar que en cuanto se defina un sólo constructor, la funcionalidad anterior deja de ser efectiva.

Otra característica que hay que tener en cuenta con los constructores es que tienen que estar en un ámbito como public para que así se pueda acceder desde donde se intente crear el objeto.



```
Clase02 objeto1= new Clase02();
objeto1.leerDatos();

Clase02 objeto2= new Clase02(18, "Pepe Perez");
objeto2.leerDatos();

Clase02 objeto3= new Clase02(objeto2);
objeto3.leerDatos();
```

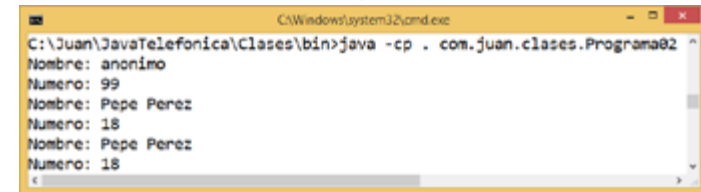
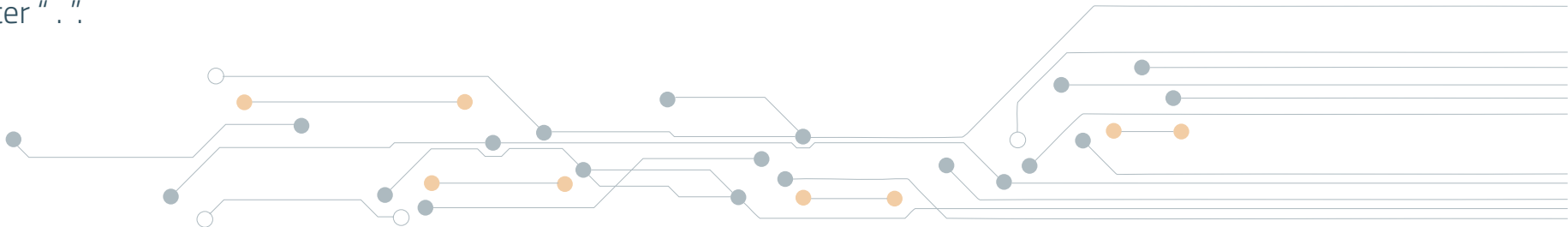


IMAGEN 5.10: CONSTRUCTORES SOBRECARGADOS

5. La referencia this

La palabra reservada **this** es una referencia al propio objeto para el que se está ejecutando el código de una función miembro de instancia o un constructor.

Sólo se puede usar dentro de las funciones de instancia y constructores de las clases. Cuando dentro de una de estas funciones se teclea **this** y a continuación un `""` se puede poner el identificador de cualquier miembro de instancia de la clase (variable o función). Por omisión es como si el acceso a todas las variables de instancia dentro de todas las funciones de instancia de la clase fueran precedidas de **this** y el carácter `."`.



Sin embargo hay casos en los que es necesario utilizar **this**. Por ejemplo cuando el identificador de un parámetro coincide con el identificador de una variable de la clase. ¿Qué necesidad existe para poner el mismo identificador al parámetro que al miembro de la clase? No es necesidad, ni obligación, es por claridad en cuanto a que se refiere y representa el identificador y por estilo de programación Java.

En el código que sigue se ilustra el uso de **this** en un constructor y en una función.

```
//Constructor con dos parámetros
public Clase03(int numero, String nombre){
    this.numero = numero;
    this.nombre = nombre;
}

//Función que accede a las variables de instancia y cambia su
valor
public void modificaDatos(String nombre, int numero){
    this.numero = numero;
    this.nombre = nombre;
}
```



6. Variables y métodos de clase: miembros static

Los miembros de clase, ya sean funciones o variables vienen precedidos del modificador **static**. Las **variables static** son espacios compartidos por todos los objetos de la clase. Pueden ser accedidos desde cualquier función miembro, sea o no static. Las funciones static son código de la clase, no de los objetos, por tanto no pueden utilizar la referencia this. Sólo pueden acceder a los miembros static de la clase.

Es estilo de Java poner los identificadores de los miembros **static** en **cursiva**.

En el código se muestra la definición de una variable y una función static, y como en los constructores se modifica el valor de la variable.

```
package com.juan.clases;

public class Clase04 {
    private static int numPersonas; //variable de
    clase

    private int numero;
    private String nombre;

    public static int cuentaPersonas(){
        return numPersonas;
    }
}
```

```
//Constructor sin parámetros
public Clase04 (){
    numPersonas++;
    numero = 99;
    nombre ="anonimo";
}

//Constructor que recibe referencia objeto de
Clase02
public Clase04(Clase04 o){
    numPersonas++;
    numero= o.numero;
    nombre= o.nombre;
}
```

Un ejemplo de utilización de la función miembro **static** es la del código siguiente. Se crean un número aleatorio de objetos, almacenando la referencia a cada uno en un array.

```
package com.juan.clases;
import java.util.Random;

public class Programa04 {
    public static void main(String[] args) {
        int num= (new Random()).nextInt(11)+5;
        Clase04 [] personas = new Clase04 [num];

        for (int i=0; i<num; i++){
            personas[i] = new Clase04(i,"Objeto
+i);
        }

        System.out.println("Se han creado "+
Clase04.
cuentaPersonas()+ " personas");
    }
}
```

Independientemente del modificador **static**, en Java se puede utilizar el modificador **final** para indicar que el valor de una variable no puede ser modificada. Cuando a una variable **static** se la modifica su comportamiento con **final**, se está definiendo una propiedad de la clase que no puede ser cambiada nunca, por ningún tipo de función. Es estilo de Java poner en mayúsculas y cursiva los miembros **final**, al igual que los valores de las enumeraciones.

En el código siguiente se define un miembro **static final** de tipo **int**, que se utiliza para que ningún objeto pueda tener en una de sus variables miembro un valor mayor que dicho miembro **final**.

```
public class Clase05 {
    private static int numPersonas; //variable de
clase
    final public static int EDAD_MAX= 65;

    private int numero;
    private String nombre;

    public Clase05 (){
        numero = EDAD_MAX;
        nombre ="anonimo";
    }
    public Clase05(int numero, String nombre){
        this.numero = numero > EDAD_MAX ? EDAD_
MAX : numero;
        this.nombre = nombre;
    }

    public void modificaDatos(String nombre,
int numero){
        this.numero = numero > EDAD_MAX ? EDAD_
MAX : numero;
        this.nombre = nombre;
    }
    //Otras funciones miembro de la clase
}
```

7. Sobrecarga de métodos

Al igual que se sobrecargan los constructores, cualquier otro método o función miembro de una clase puede estar sobrecargado. Se pueden tener varias **funciones que tengan el mismo nombre y que devolviendo el mismo tipo, se diferencien en la lista de parámetros, en su número o en sus tipos o en ambas cosas.**

La sobrecarga se resuelve en compilación. El compilador por la forma de la función, por la lista de parámetros, sabe qué función es la llamada.

El código que se muestra a continuación presenta una función sobrecargada de 4 formas.

```
public Clase06{
    private static int numPersonas; //variable de
    clase
    final public static int EDAD_MAX= 65;
    private int numero;
    private String nombre;
    //Constructores

    //Función que accede a las variables de
    instancia y cambia su valor
    //sobrecargada de 4 formas
```

```
        public void modificaDatos(String nombre, int
        numero){
            this.numero = numero > EDAD_MAX ? EDAD_
            MAX : numero;
            this.nombre = nombre;
        }
        public void modificaDatos(String nombre){
            this.nombre = nombre;
        }
        public void modificaDatos(int numero){
            this.numero = numero > EDAD_MAX ? EDAD_
            MAX : numero;
        }
        public void modificaDatos(){
            this.nombre = "anonimo";
            this.numero = EDAD_MAX;
        }
        //Otras funciones
    }
```

En la imagen se muestra código que utiliza la función sobrecargada anterior:

```
public class Programa06 {  
    public static void main(String[] args) {  
        Clase06 objeto1= new Clase06();  
        objeto1.modificaDatos("Luis Ruiz", 72);  
  
        Clase06 objeto2= new Clase06(18, "Pepe Perez");  
        objeto2.modificaDatos(Clase06.EDAD_MAX);  
  
        Clase06 objeto3= new Clase06(objeto2);  
        objeto3.modificaDatos("Antonio Gil");  
  
        Clase06 objeto4 = new Clase06(objeto1);  
        objeto4.modificaDatos();  
    }  
}
```

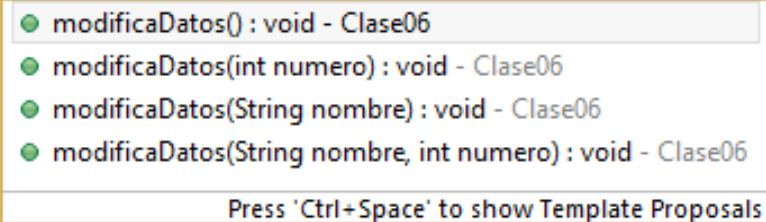
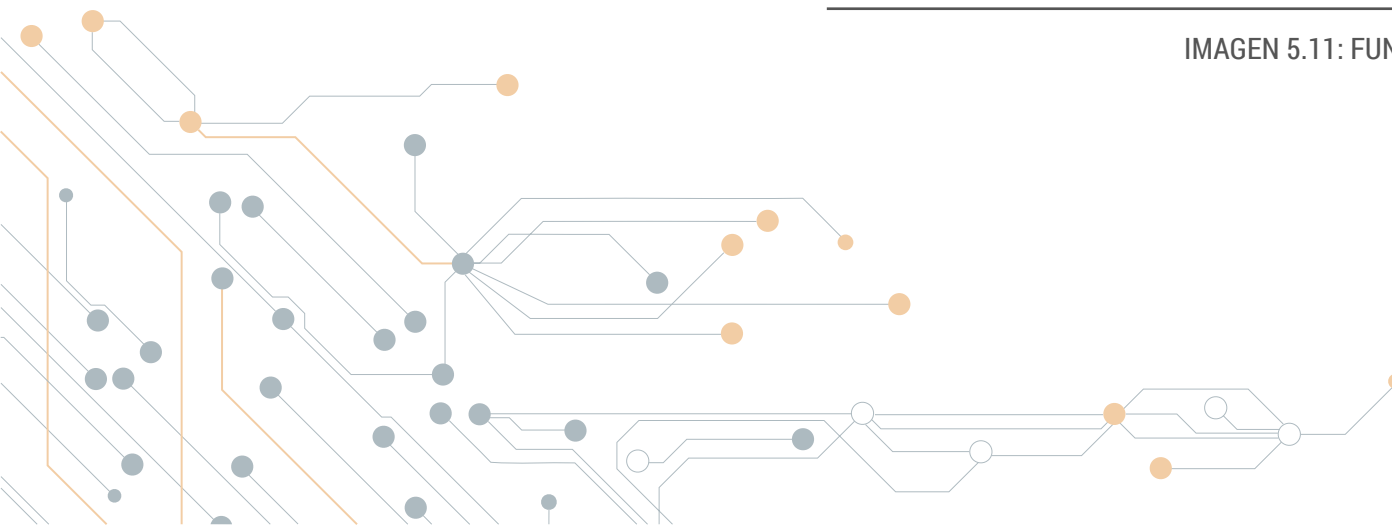


IMAGEN 5.11: FUNCIONES SOBRECARGADOS



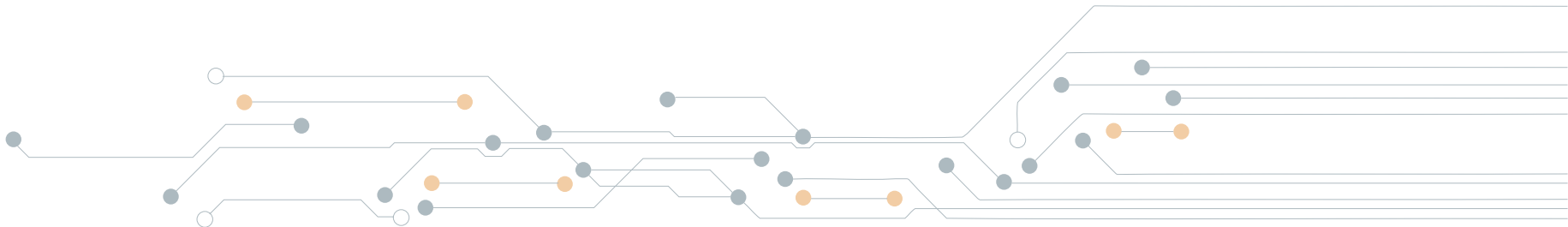
8. Métodos "getter" y "setter"

Los métodos "getter" y "setter" son métodos de acceso a los datos de los objetos y son siempre públicos. Cada método accede a un sólo dato (una sola variable de instancia) para obtener su valor o cambiarlo. Las funciones "getter" retornan el valor de una variable de clase y las funciones "setter" cambian el valor de una variable de clase. Por estilo estas funciones utilizan identificadores que comienzan por set o get y a continuación el nombre de la variable miembro comenzando por mayúscula.

La clase siguiente define estas funciones para sus variables de instancia.

```
public class Clase07 {  
    private static int numPersonas; //variable de  
    clase  
    final public static int EDAD_MAX= 65;  
    private int numero;  
    private String nombre;  
  
    public int getNumero() {  
        return numero;  
    }  
}
```

```
public void setNumero(int numero) {  
    this.numero = numero > EDAD_MAX ? EDAD_  
MAX : numero;  
}  
public String getNombre() {  
    return nombre;  
}  
public void setNombre(String nombre) {  
    this.nombre = nombre;  
}  
// El resto de funciones  
}
```

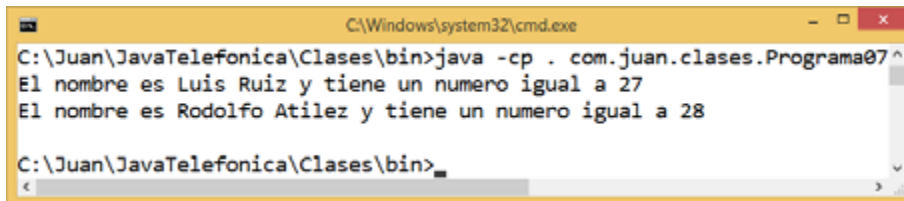


Una posible utilización de estas funciones es como la que muestra la imagen siguiente:

```
public static void main(String[] args) {
    Clase07 objeto1= new Clase07();
    objeto1.modifcaDatos("Luis Ruiz", 27);

    System.out.print("El nombre es "+objeto1.getNombre());
    System.out.println(" y tiene un numero igual a "+objeto1.
        getNumero());

    Clase07 objeto2 = new Clase07(objeto1.getNumero(),"Rodolfo
        Atilez");
    objeto2.setNumero(objeto2.getNumero()+1);
    System.out.print("El nombre es "+objeto2.getNombre());
    System.out.println(" y tiene un numero igual a "+objeto2.
        getNumero());
}
```



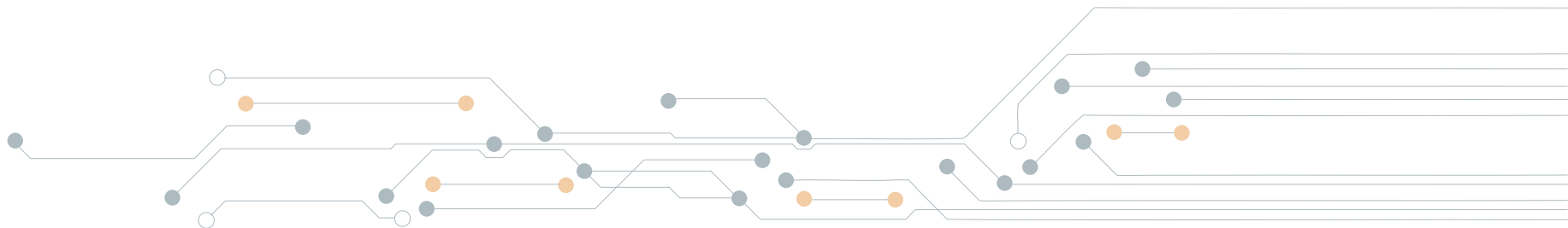
```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\Clases\bin>java -cp . com.juan.clases.Programa07
El nombre es Luis Ruiz y tiene un numero igual a 27
El nombre es Rodolfo Atilez y tiene un numero igual a 28
C:\Juan\JavaTelefonica\Clases\bin>
```

Las clases que representan entidades(tablas) de bases de datos y para las que cada objeto instanciado es la representación de un registro, son clase que sólo suelen tener:

- Una variable miembro que se corresponde con cada campo de la tabla.
- El constructor sin parámetros.
- Funciones get y set para cada una de las variables miembro.

Un **JavaBean**, también llamado simplemente **Bean** es una clase simple en Java que cumple con las normas anteriores y además es serializable (se estudia más adelante).

IMAGEN 5.11: FUNCIONES GET Y SET



Un ejemplo de **Bean** que representa un usuario típico:

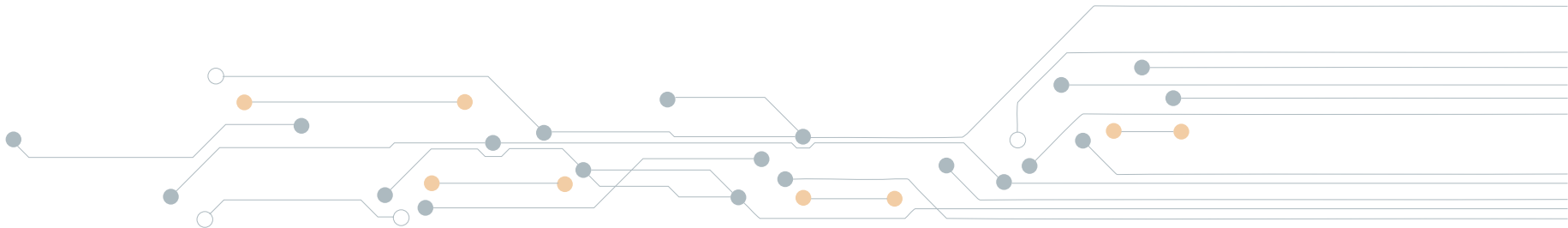
```
package com.juan.clases;

public class UsuarioBean implements java.io.Serializable{
    private String usuario;
    private String clave;
    private String email;
    private byte edad;

    public UsuarioBean() {

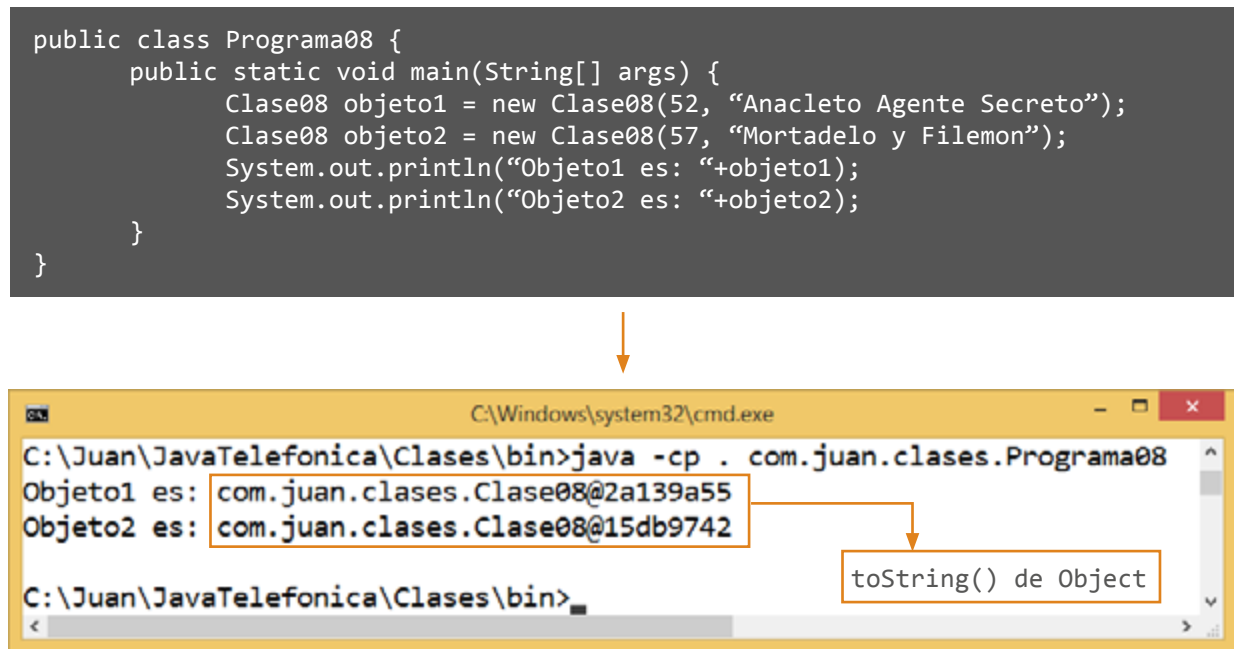
    }
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
}
```

```
    public String getClave() {
        return clave;
    }
    public void setClave(String clave) {
        this.clave = clave;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public byte getEdad() {
        return edad;
    }
    public void setEdad(byte edad) {
        this.edad = edad;
    }
}
```



9. El método toString

Cuando en la expresión de la llamada a las funciones “print” o “println”, se utiliza la referencia a un objeto instanciado de una clase, Java llama al método toString, de dicha clase. Si dicha clase no lo tiene implementado se ejecuta el de la clase Object, esta es la clase de la que heredan todas las clases de Java (es la raíz de todas). Lo que sucede en este caso es lo que la imagen muestra:



Para que no se ejecute el código heredado de `Object` es necesario **"sobrescribir"**, definiendo la función `toString()` en las clases en las que esta funcionalidad se va a necesitar. Sobrescribir funciones es modificar el comportamiento de las funciones que se heredan de las superclases. Las funciones tienen que ser exactamente iguales en su nombre, el valor devuelto y la lista de parámetros. Con esta característica es con la que se implementa el polimorfismo.

FIGURA 5.12: TOSTRING DE OBJECT

El código siguiente muestra como en la clase del ejemplo se sobrepasa la función toString().

```
public class Clase09 {
    private static int numPersonas; //variable de
    clase
    final public static int EDAD_MAX= 65;

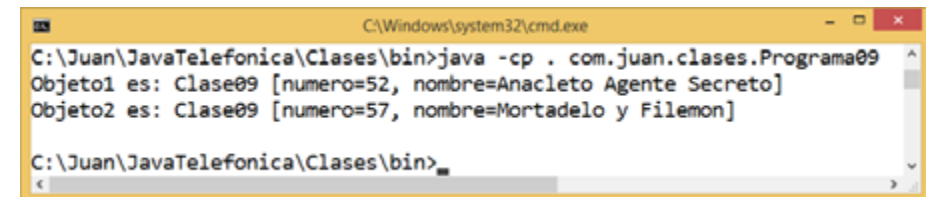
    private int numero;
    private String nombre;

    @Override
    public String toString() {
        return "Clase09 [numero=" + numero + ",
        nombre=" + nombre + "]";
    }
    //Otras funciones de la clase
}
```

El uso de esta funcionalidad se muestra en la imagen siguiente:

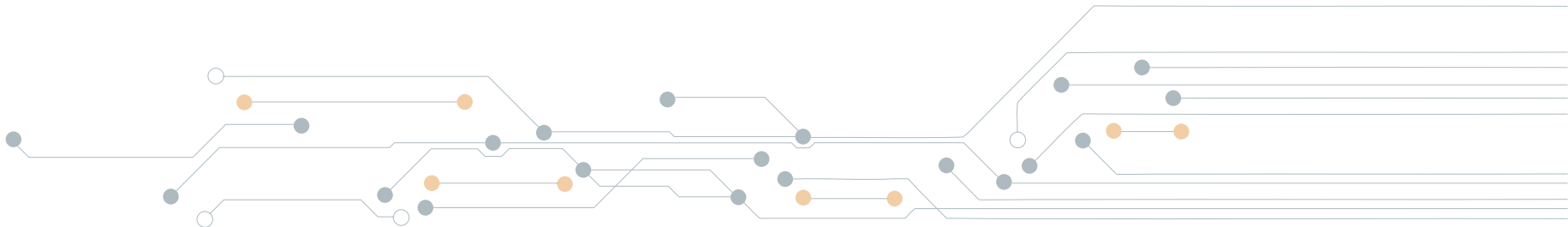
```
public class Programa08 {
    public static void main(String[] args) {
        Clase09 objeto1 = new Clase09(52, "Anacleto
        Agente Secreto");
        Clase09 objeto2 = new Clase09(57,
        "Mortadelo y Filemon");
        System.out.println("Objeto1 es: "+objeto1);

        System.out.println("Objeto2 es: "+objeto2);
    }
}
```



```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\Clases\bin>java -cp . com.juan.clases.Programa09
Objeto1 es: Clase09 [numero=52, nombre=Anacleto Agente Secreto]
Objeto2 es: Clase09 [numero=57, nombre=Mortadelo y Filemon]
C:\Juan\JavaTelefonica\Clases\bin>
```

FIGURA 5.13: SOBRESCRIBIR TOSTRING



10. El método equals

El operador `==` no se puede utilizar para comparar si dos objetos son iguales o no. Para poder realizar esta operación hay que sobrescribir la función **`equals(Object o)`** de **`Object`**, que por omisión no ejecuta ninguna funcionalidad. En el código de `equals` se implementan las reglas que para los objetos de la clase determinas la igualdad. Como por ejemplo:

```
public class Clase10 {
    private int numero;
    private String nombre;
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Clase10 other = (Clase10) obj;
        if (nombre == null) {
            if (other.nombre != null)
                return false;
        } else if (!nombre.equals(other.nombre))
            return false;
        if (numero != other.numero)
            return false;
        return true;
    }
}
```

Hay que tener claro que si se utiliza el operador `==` con dos referencias a objetos, lo que se está comparando es si se referencia o no al mismo objeto, las referencias no son los objetos, un objeto puede tener varias referencias.

Así en el código se examinan las siguientes condiciones:

- Si la referencia `obj` recibida referencia al mismo objeto que `this`, si son iguales se trata del mismo objeto.
- Si la referencia recibida es `null`, es evidente que no es el objeto para el que se invoca la función `equals`.
- Si la referencia recibida no es de la misma clase que el objeto que referencia `this`, está claro que no pueden ser iguales, son objetos de clases diferentes.
- Después de estas comprobaciones ya se puede decir que tanto como la referencia recibida como `this` son de objetos distintos pero de la misma clase, por tanto se empieza a comprobar la igualdad variable miembro a variable miembro.

En código siguiente comprueba las diferentes posibilidades de utilización de la función equals anterior:

```
public static void main(String[] args) {  
    Clase10 objeto1 = new Clase10(52, "Anacleto");  
    Clase10 objeto2 = new Clase10(57, "Mortadelo y Filemon");  
    Clase10 clonobjeto2 = new Clase10(objeto2);  
    Clase10 objeto3 = new Clase10(objeto1.getNumero(), "Carpanta");  
    Clase10 objeto4 = new Clase10(27, objeto1.getNombre());  
    Clase10 objetonull=null;  
    Clase10 objeto=objeto1;  
  
    System.out.println(objeto1.equals(objeto));           //true  
    System.out.println(objeto1.equals(objetonull));       //false  
    System.out.println(objeto1.equals("Anacleto"));       //false  
    System.out.println(objeto1.equals(objeto3));          //false  
    System.out.println(objeto1.equals(objeto4));          //false  
    System.out.println(objeto2.equals(objeto1));          //false  
    System.out.println(objeto2.equals(clonobjeto2));      //true  
}
```



11. Garbage collector y el método finalize

Cuando un objeto llega al final del ámbito en el que creado o se queda sin referencia, JVM libera el espacio que ocupa. de esta liberación se encarga el Garbage Collector. Pero la intervención de Garbage Collector no es inmediata, se esta ejecutando en segundo plano y cuando le toque intervenir hace la limpieza de todos los espacios que deben ser liberados de la zona de memoria "heap", que es en la que se almacena todos los objetos que se crean con new.

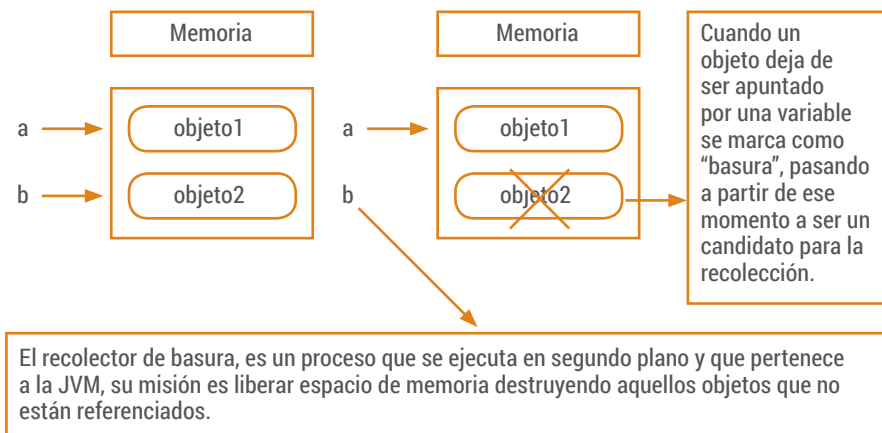
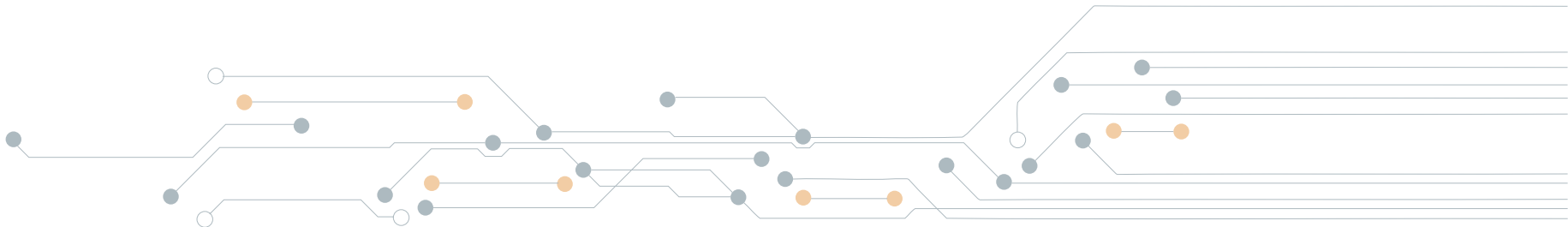


FIGURA 5.14: GARBAGE COLLECTOR (RECOLECTOR DE BASURA)

Se puede provocar que intervenga el GC invocándole explícitamente de la forma siguiente:

```
Runtime garbage = Runtime.getRuntime();
garbage.gc();
```

Un instante antes de liberar la memoria de un objeto JVM invoca a la función **finalize()**, si no se ha sobrepasado esta funcionalidad en la clase, se ejecuta **finalize()** de Object, que en realidad no ejecuta ningún código. Por tanto las clases sobrescribirán esta funcionalidad cuando tengan que realizar algo concreto antes de liberar el espacio de las variables miembros de los objetos. Esto no suele ser necesario normalmente, pero puede suceder en que en determinadas situaciones haya que liberar recursos que se hayan solicitado, como por ejemplo aquellos que tengan funcionalidades de open y close.



En el ejemplo siguiente se ilustra la utilización de `finalize`, que realiza una función complementaria a la realizada en los constructores.

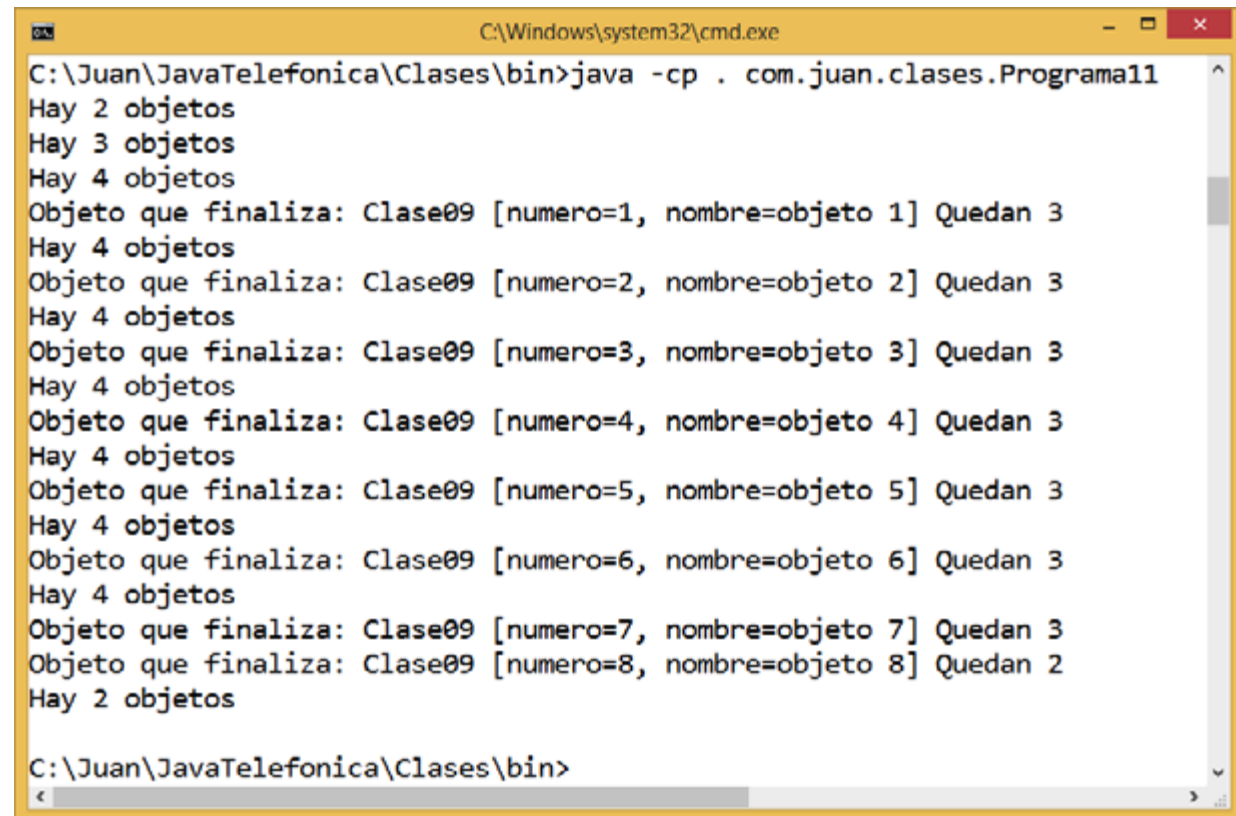
```
@Override
protected void finalize() throws Throwable {
    numPersonas--;
    System.out.println("Objeto que finaliza: "+this
        + " Quedan " + numPersonas);
}
```

Con `finalize()` hay que tener en cuenta que no esta garantizado que se ejecute antes de la finalización del código, porque lo habitual es que el GC intervenga cuando el código finaliza, justo instantes después de llegar a la llave final de los bloques de código `{}`.

En el ejemplo siguiente, para forzar la ejecución de `finalize`, se invoca explícitamente a GC y además se introduce un retraso en la ejecución del código de 5 segundos con **`Thread.sleep(5000)`**.

```
public static void main(String[] args)
    throws InterruptedException {
    Clase11 objeto1= new Clase11(28,"uno");
    Clase11 objeto2= new Clase11(38,"dos");
    System.out.println("Hay "+Clase11.
        cuentaPersonas()+" objetos");
    int num= (new Random()).nextInt(9)+1;
    for(int i=1; i<=num; i++){
        new Clase11(i, "objeto "+i);
        System.out.println("Hay "
            +Clase11.
        cuentaPersonas()+" objetos");
        Runtime garbage = Runtime.
        getRuntime();
        garbage.gc();
    } //Fin del ámbito de cada objeto que
    //se crea en cada iteración
    Thread.sleep(5000); //Retraso de la
    ejecución de 5 segundos
    System.out.println("Hay "+
        Clase11.cuentaPersonas()+"
    objetos");
}
```

El resultado de la ejecución de este código puede ser como el que se muestra en la imagen siguiente:



```
C:\Windows\system32\cmd.exe
C:\Juan\JavaTelefonica\Clases\bin>java -cp . com.juan.clases.Programa11
Hay 2 objetos
Hay 3 objetos
Hay 4 objetos
Objeto que finaliza: Clase09 [numero=1, nombre=objeto 1] Quedan 3
Hay 4 objetos
Objeto que finaliza: Clase09 [numero=2, nombre=objeto 2] Quedan 3
Hay 4 objetos
Objeto que finaliza: Clase09 [numero=3, nombre=objeto 3] Quedan 3
Hay 4 objetos
Objeto que finaliza: Clase09 [numero=4, nombre=objeto 4] Quedan 3
Hay 4 objetos
Objeto que finaliza: Clase09 [numero=5, nombre=objeto 5] Quedan 3
Hay 4 objetos
Objeto que finaliza: Clase09 [numero=6, nombre=objeto 6] Quedan 3
Hay 4 objetos
Objeto que finaliza: Clase09 [numero=7, nombre=objeto 7] Quedan 3
Objeto que finaliza: Clase09 [numero=8, nombre=objeto 8] Quedan 2
Hay 2 objetos
C:\Juan\JavaTelefonica\Clases\bin>
```

FIGURA 5.15: EJECUCIÓN DEL MÉTODO FINALIZE.



12. Clases anidadas e internas

En Java se pueden definir clases dentro de otras clases, esto se conoce como **clases anidadas**. Una clase anidada no existe fuera de su clase contenedora y puede acceder a todos sus miembros, incluso a los privados, pero no al contrario.

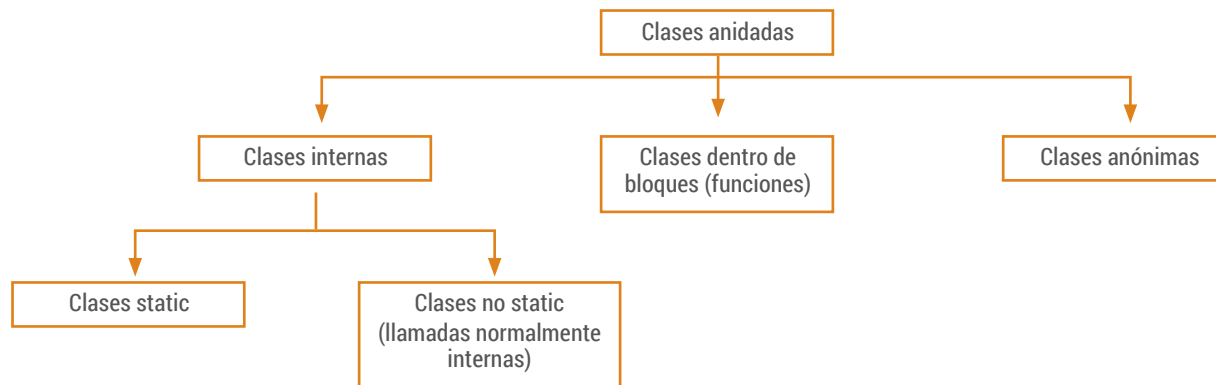


FIGURA 5.16: TIPOS DE CLASES ANIDADAS.

Una clase declarada dentro de otra es miembro de esta. Las clases anidadas pueden ser static o no. Las que normalmente se llaman **clases** internas son las clases anidadas que no van precedidas de static

También se pueden declarar clases anidadas dentro de bloques de la clase contenedora, por ejemplo dentro de una función.

Además se pueden crear clases anidadas sin nombre, son las llamadas **anónimas**.

En general se puede decir que si una clase es interna a otra es porque sirve a un propósito muy específico y particular de la clase contenedora y se opta por poner esta funcionalidad muy cerca de quien la utiliza (la clase contenedora).

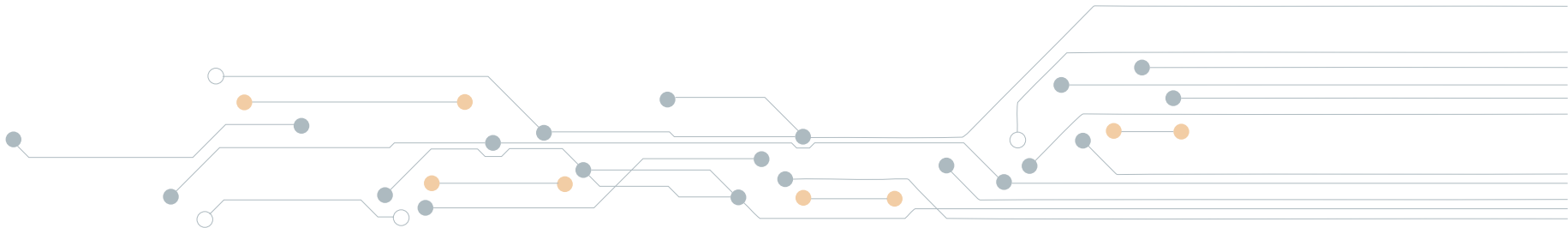
Suelen ser clases sencillas con uno o dos métodos, pero complejas en la sintaxis de utilización.

Como ejemplo vamos a utilizar una clase interna dentro de otra, una clase que implementa una lista de Integer con operaciones "add", "get" y un "iterador" que será la clase interna.

```
public class ListaNumerosIterador {  
    private Integer [] lista;  
    private int numElementos;  
  
    public ListaNumerosIterador(int maxElementos){  
        lista = new Integer[maxElementos];  
    }  
  
    public int get(int i){  
        return lista[i];  
    }  
  
    public void add(int i){  
        lista[numElementos++]=i;  
    }  
  
    public Iterador getIterador(){  
        return new Iterador();  
    }  
}
```

La clase interna tiene un método "siguiente" que retorna el Integer que corresponda según se recorre la lista de inicio a fin, o null si se ha finalizado el recorrido. Es la funcionalidad de muchos iteradores en Java que se utilizan para recorrer listas con métodos "hasNext" o similares- Esta clase interna es la siguiente:

```
class Iterador{  
    int indice;  
    Integer siguiente(){  
        if(indice < numElementos)  
            return lista[indice++];  
        else  
            return (Integer) null;  
        //fin de la clase interna  
    }  
    //fin de la clase contenedora
```



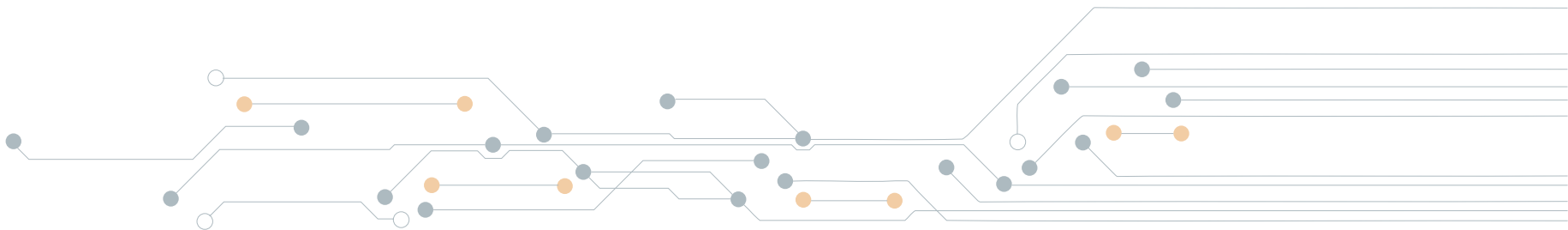
Un código que prueba la utilización de esta clase interna es el siguiente:

```
package com.juan.clasesinternas;
import java.util.Random;
public class programaListaNumeros {
    public static void main(String[] args) {
        Random aleatorio = new Random();
        int cant_numeros= aleatorio.nextInt(10)+1;
        ListaNumerosIterador milista =
                                new
        ListaNumerosIterador(cant_numeros);

        for(int i=0; i<cant_numeros; i++)
            milista.add(aleatorio.nextInt());

        ListaNumerosIterador.Iterador
            iterador= milista.getIterador();
        Integer numero;
        while( (numero= iterador.
siguiente())!=null )
            System.out.print(numero+" ");
    }
}
```

En realidad ningún tipo de clase anidada es necesario, todo se puede resolver con clases no anidadas que tiene algún tipo de relación, pero repitiendo lo dicho anteriormente se incluyeron en Java para poder implementar soluciones orientadas a objetos en las que una funcionalidad representada en una clase, sólo es utilizada únicamente por otra clase y se anida dentro de esta para poder tener acceso cómodo y seguro, pero obligando a introducir complicidad en la sintaxis de su utilización.



Telefonica

EDUCACIÓN DIGITAL