

## UNIDAD DE TRABAJO 5. INTRODUCCIÓN A LA ORIENTACIÓN A OBJETOS

1. Los tres principios de la programación
  1. Encapsulado
  2. Herencia
  3. Polimorfismo
  4. Encapsulado, herencia, polimorfismo
2. Modificadores de visibilidad.
3. Objetos, estado, comportamiento e identidad.
  1. Instanciación de los objetos. Declaración y creación de objetos.
  2. Utilización de métodos y atributos.
4. Propiedades
  1. Declaración
  2. Inicialización
5. Métodos.
  1. Definición y creación.
  2. Llamada al método.
  3. Ámbito, alcance y tiempo de almacenamiento de un identificador.
  4. Variables locales y globales.
  5. Tipo de almacenamiento de una variable local
  6. Valor de retorno
  7. Paso de parámetros
    1. Paso por valor
    2. Paso por referencia
  8. Tipos de almacenamiento.
    1. Variables estáticas
    2. Propiedades estáticas
    3. Métodos estáticos
6. Constructores
7. Sobrecarga de métodos
8. La referencia this.
9. El método finalize()
10. Parámetros de main().
11. Métodos recursivos.
12. Controles de acceso.
13. Clases Internas.

La programación orientada a objetos es la base de Java. De hecho, todos los programas Java son orientados a objetos.

El paradigma orientado a objetos nació en 1969 de la mano del doctor noruego Kristin Nygaard.

Todos los programas están formados por dos elementos: código y datos. La P.O.O. organiza programa entorno a sus datos (objetos) y a un conjunto de interfaces bien definidas a esos datos. Un programa orientado a objetos se puede interpretar como *datos que controlan el acceso al código*.

Un elemento esencial de la P.O.O. es la **abstracción**. Por ejemplo, no pensamos que nuestro coche sea un conjunto de decenas de miles de partes individuales. Pensamos que es un objeto bien definido con un comportamiento propio único, como ir al supermercado, ir a la playa, etc., sin ser abrumados por la complejidad de las partes que forman el coche. Podemos ignorar los detalles acerca de cómo funciona el motor, la transmisión y el sistema de frenos, utilizando el coche en su conjunto.

Una forma poderosa de gestionar la abstracción es utilizando clasificaciones jerárquicas. Esto nos permite dividir en niveles la semántica de los sistemas complejos. Desde el exterior, el coche es un objeto sencillo. Una vez dentro podemos ver que el coche está compuesto de varios subsistemas: la dirección, los frenos, el equipo de sonido, los cinturones de seguridad, la calefacción y el teléfono móvil. Por ejemplo, el equipo de sonido está compuesto de radio, el CD y el cassette.

### 1. Los tres principios de la programación

Los lenguajes de programación orientada a objetos proporcionan mecanismos que ayudan a implementar el modelo orientado a objetos, éstos son: encapsulado, herencia y polimorfismo.

#### Encapsulado

El encapsulado es el mecanismo que permite juntar el código y los datos que manipula, y que mantiene a ambos alejados de posibles interferencias o usos indebidos. El encapsulado es como un envoltorio protector que evita que otro código que está fuera pueda acceder arbitrariamente al código o a los datos. El acceso al código y a los datos se realiza de forma controlada a través de una interfaz bien definida.

Para relacionar esto con el mundo real, consideremos la transmisión de un automóvil. La transmisión automática encapsula información acerca del motor, como cuánto se está acelerando, el terreno sobre el que está y la posición de la palanca de cambios. Nosotros, como usuarios, solo tenemos un método para actuar sobre este encapsulado complejo: mover la palanca de cambio. Por lo tanto, no puede afectar a la transmisión utilizando el intermitente o el limpiaparabrisas. La palanca de cambio es una interfaz bien definida para la transmisión. Es más, lo que ocurra dentro de la transmisión no afecta a los objetos que están fuera de ella. Como la transmisión automática está encapsulada, los fabricantes de coches pueden implementar este mecanismo de la manera que les parezca bien.

En la programación, el poder del código encapsulado es que todo el mundo conoce cómo acceder a él y pueden utilizarlo independientemente de los detalles de implementación.

En Java la base del encapsulado es la **clase**. Una **clase** define la estructura y el comportamiento (datos y código) que serán compartidos por un conjunto de objetos. Cada objeto de una clase dada contiene la estructura y el comportamiento definidos por la clase, como si fuera grabado por un molde con la forma de la clase. Una clase es una construcción lógica; un **objeto** tiene realidad física.

Cuando se crea una clase, hay que especificar el código y los datos que constituyen esa clase. En su conjunto, estos elementos son **miembros** de la clase. Los *datos* definidos por la clase son las **variables** o propiedades. El *código* que opera sobre esos datos se conoce como **método o función**.

#### **Formato**

```
class NombreClase
{
    lista_de_miembros
}
```

Donde lista de miembros son los datos y los métodos (operaciones que se pueden realizar sobre dichos datos).

Por ejemplo:

```
Class Alumno{
    DATOS:
        String nombre;
        int edad;
        ....
    OPERACIONES:
        void iniciarDatos(String n, int e);
        void pedirNombre();
        void pedirEdad();
        void visualDatosAlumno();
        .....
};
```

Por ello, una clase está formada por:

- Los **atributos** o **propiedades**: son los datos que contiene la clase. Una clase puede tener cualquier número de atributos o no tener ninguno. Se declaran con un nombre y el tipo de dato correspondiente.
- Los **métodos**: definen el comportamiento de la clase, ya que está representado por las operaciones que puede realizar. Permite cambiar los datos de los atributos que forman la clase.

Cada método o variable de una clase se puede marcar como **privado** o **público**. La interfaz **pública** de una clase representa todo lo que los usuarios externos de la clase necesitan conocer, o que pueden conocer. Los métodos y variables **privadas** solo pueden ser utilizados por código de la clase.

### Herencia

La herencia es el proceso mediante el cual un objeto adquiere las propiedades de otro. Esto es importante ya que así se consigue la clasificación jerárquica.

Por ejemplo, un perro perdiguero es una parte de la clasificación de perro, que a su vez, es parte de la clase mamífero, que es debajo de la clase animal. Si no se utilizaran jerarquías, cada objeto debería definir todas sus características. Sin embargo, con la herencia, un objeto solo necesita definir aquellas cualidades que lo hacen único dentro de su clase. Este objeto puede heredar sus atributos generales de su padre.

Si se deseara describir los animales de una manera abstracta (clase animales), diría que tienen atributos (variables o propiedades), como tamaño, inteligencia y tipo de esqueleto. Los animales también tienen ciertos aspectos de comportamiento: comen, respiran y duermen.

Si se deseara describir a los mamíferos (subclase mamíferos pertenecientes a la superclase animales), tendrían atributos más específicos, como el tipo de dientes y las glándulas mamarias.

La herencia interactúa también con el encapsulado. Si una clase dada encapsula algunos atributos, entonces cualquier clase tendrá los mismos atributos más cualquiera que añada como parte de su especialización.

### Polimorfismo

El polimorfismo es una característica que le permite a una interfaz ser utilizada por una clase general de acciones.

El concepto general de polimorfismo se puede expresar con la frase “una interfaz, varios métodos”. Esto significa que es posible diseñar una interfaz genérica para un grupo de actividades relacionadas.

Continuando con el ejemplo del perro perdiguero, el sentido del olfato del perro es polimórfico. Si el perro huele a un gato, ladrará y correrá detrás de él. Si el perro huele su comida, segrega saliva y correrá hacia su cuenco. La diferencia está en lo que huele, es decir, en el tipo de dato que opera sobre la nariz del perro.

### Polimorfismo, encapsulado y herencia

Cuando se aplican adecuadamente, el polimorfismo, el encapsulado y la herencia pueden producir un entorno de programación que admite el desarrollo de programas mucho más robustos y fáciles de ampliar que los modelos tradicionales de diseño orientado a proceso.

El automóvil muestra de manera más completa la potencia del diseño orientado a objetos. Todos los conductores se basan en la herencia para conducir diferentes tipos (subclases) de vehículos. Independientemente de si el vehículo es un autobús de colegio, un Mercedes, un Porsche o un coche familiar, todos los conductores pueden más o menos encontrar y utilizar el volante, los frenos y el acelerador. Después de un poco de práctica, la mayoría de nosotros podemos incluso superar la diferencia entre un cambio manual y uno automático, porque básicamente entendemos el fundamento de su superclase común, la transmisión.

El polimorfismo se refleja con más claridad en la capacidad de los fabricantes de coches de ofrecer una amplia variedad de opciones sobre el mismo vehículo. Por ejemplo, puede elegir frenos ABS o tradicionales, dirección asistida o normal, motor de 4, 6 u 8 cilindros. En cada uno de estos modelos, sigue siendo necesario pisar el freno para parar, girar el volante para cambiar de dirección y pisar el acelerador para andar. La misma interfaz se puede utilizar para controlar estos modelos de coches.

## 2. Modificadores de visibilidad.

**Para controlar el acceso a los miembros de la clase se utilizan cuatro especificadores de acceso:** `public`, `private`, `package` (no poner nada) y `protected`. Cada miembro de la clase está precedido del especificador de acceso que le corresponde.

- **public** → cualquier clase puede acceder a los miembros de la clase actual.
- **private** → solo podrán acceder a ellos los métodos de la propia clase.
- **protected** → cualquier clase que esté dentro del mismo paquete, así como cualquier clase que herede dicha clase, podrán acceder a dichos elementos.
- **package** (no poner nada o `friendly`) → podrán ser utilizados por los métodos de otras clases, siempre que pertenezcan al mismo paquete.

Sobre las propias clases también se pueden establecer permisos de visibilidad. De esta forma, si la clase es pública, se podrá usar fuera del paquete donde pertenece; pero si no se pone ningún modificador de visibilidad, significa que es `package`, solo es visible dentro del mismo paquete.

## 3. Objetos: estado, comportamiento e identidad.

Un **objeto** es una unidad básica de programación orientada a objetos. Es la declaración de una variable donde el tipo es una clase.

Están formados por:

El **estado** son los valores concretos que tiene un objeto en cada uno de sus propiedades.

El **comportamiento** está formado por todos los métodos que forman la clase ya que, dichos métodos, son los que fijan las operaciones que puede realizar un objeto.

La **identidad** es el nombre que se le ha dado a un objeto (su identificador).

```
package paquete_Taxi;
public class Taxi //Nombre de la clase
{
    private String matricula; //Matrícula de cada objeto taxi
    private String taxista; //Nombre del dueño de cada objeto taxi
    public int tipoMotor; //Tipo de motor asignado a cada objeto taxi. 0 = desconocido, 1 = gasolina,
```

2 = diesel

```
//Método para inicializar los datos o propiedades
public void iniciaDatos (String valorMatricula, String valorTaxista, int valorTipoMotor)
{
    this.matricula=valorMatricula;
    this.taxista=valorTaxista;
    this.tipoMotor=valorTipoMotor;
} //Cierre del método

//Método que devuelve una cadena con Taxista y el nombre del taxista
public String visualTaxista () { return "\nTaxista: "+this.taxista; } //Cierre del método

//Método para obtener la matrícula del objeto taxi
public String getMatricula () { return this.matricula; } //Cierre del método

//Método que devuelve una cadena con los valores del taxi actual
public String toString ()
{
    String cadena="Datos del Taxi:\nMatrícula:"+this.matricula+this.visualTaxista()+"\nTipo del
motor:"+ this.tipoMotor;
    return cadena;
} //EL método visualTaxista() se puede llamar desde cualquier punto de la clase Taxi
}
```

```
package paquete_Taxi;
public class Principal //Nombre de la clase donde va estar el método main()
{
    public static void main(Strings []args)
    {
        Taxi t1, t2; //Declaración de dos objetos de tipo Taxi
        t1=new Taxi(); //Creación del objeto t1 asignándole memoria
        t2=new Taxi();
        t1.iniciaDatos("9898-HKJ","Pepe López", 1); //Guardamos datos para el primer taxi
        t2.iniciaDatos("2345-GFH","Amador González", 2); //Guardamos datos para el segundo taxi
        System.out.println("Taxi t1 :"+t1.toString()); //Mostramos los datos del primer taxi
        System.out.println("La matrícula del segundo taxi es: "+t2.getMatricula());
        System.out.println("El tipo de motor del primer taxi es : +t1.tipoMotor);
    }
}
```

Identidad: los objetos se llaman t1 y t2

Estado:

Los valores de las propiedades de t1 son Matrícula: "9898-HKJ", taxista:"Pepe López" y el tipo de motor: 1 (diesel).


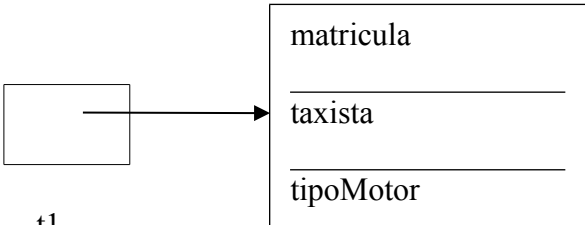
### 3.1. Instanciación de objetos. Declaración y creación.

La instanciación de un objeto consiste, precisamente, en poder asignar memoria necesaria para poder guardar datos. Para poder realizar ésto, hay que llevar a cabo estos dos pasos:

1. Declarar el objeto Clase objeto; Por ejemplo, Taxi t1;
2. Creación del objeto donde se reserva memoria, y se hace con el operador **new** y a partir de ahí se pueden guardar datos.

objeto=new Clase(); Por ejemplo, t1=new Taxi();

También puede ser: Clase objeto = new Clase(); Por ejemplo, Taxi t1=new Taxi();

Orden	Efecto
Taxi t1;	 <p>t1</p>
t1=new Taxi();	 <p>objeto Taxi</p>

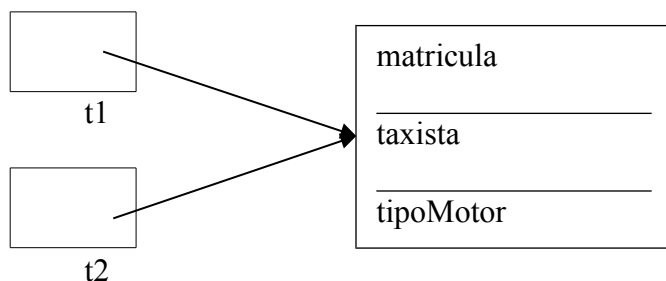
Cuando se asigna una variable referencia a objeto a otra, no se está creando una copia del objeto, sino que sólo se está haciendo una copia de la referencia.

Por ejemplo,

Taxi t1=new Taxi();

Taxi t2=t1;

Sería:



Aunque **t1** y **t2** referencian al mismo objeto, no están relacionadas de ninguna otra forma. Por ejemplo, cualquier asignación posterior de **t1** *desenganchara* **t1** del objeto original sin afectar al propio objeto **t2**.

Por ejemplo,

```
Taxi t1=new Taxi();
```

```
Taxi t2=t1;
```

```
.....
```

```
t1=null;
```

Aquí, **t1** ha sido asignado a **null**, pero **t2** todavía apunta al objeto original.

### **3.2. Utilización de métodos y propiedades.**

Para poder utilizar los métodos o atributos de una clase hay que distinguir si se quiere hacer uso de ellos fuera o dentro de la clase:

- Para hacer uso de los métodos o los atributos de una clase fuera de ella, solo hay que declarar y crear un objeto; a partir de aquí, podremos usar los métodos o atributos de dicho objeto. **Siempre y cuando el modificador de visibilidad lo permita.** Para hacerlo se pone:

*nombre\_objeto.nombre\_del\_metodo(lista\_datos\_entrada)*

*nombre\_objeto.atributo*

Por ejemplo,

```
t2.iniciaDatos("2345-GFH","Amador González", 2); //Guardamos datos para el segundo taxi
System.out.println("El tipo de motor del primer taxi es : +t1.tipoMotor);
```

- Para hacer uso de los métodos o atributos dentro de la propia clase, se hace indicando el nombre de dicho método sin indicar el nombre del objeto.

Por ejemplo,

```
public void iniciaDatos (String valorMatricula, String valorTaxista, int valorTipoMotor)
{
    this.matricula=valorMatricula;
    this. taxista=valorTaxista;
    this.tipoMotor=valorTipoMotor;
} //Cierre del método
```

```
public String toString ()
{
    String cadena="Datos del Taxi:\nMatrícula:"+this.matricula+this.visualTaxista()+"\nTipo del
motor:"+this. tipoMotor;
    return cadena;
} //Cierre del método
```

#### **4. Atributos, propiedades o variables miembro**

Una clase puede no tener atributos.

##### **4.1. Declaración**

La sintaxis para declarar un dato dentro de una clase es la siguiente:

Ambito tipo identificador;

- Ámbito → visibilidad del atributo (package, public, protected, private)
- Tipo → tipo de dato del atributo (int, double ...)
- Identificador → nombre identificativo del atributo (edad, nombre, etc)

##### **4.2. Inicialización**

Los atributos de una clase se pueden inicializar en la misma declaración; y se le dan valores iniciales, pudiendo cambiar su valor en cualquier momento a través de sus métodos o funciones.

Si a los atributos de una clase no se le dan valores iniciales, el compilador se los da. Si son de tipo numérico, toman el valor 0; si son objetos, les da el valor null; si es boolean, el valor false, y si es char, le asigna el carácter que corresponde al código ASCII 0.

Por ejemplo, `private String ciudad="Murcia";`

#### **5. Métodos**

##### **5.1. Definición y creación**

Un método se define de forma similar al método `main()`, y se hace una sola vez a lo largo del programa o de la clase.

Formato:

```
TipoBaseReturn    nombreMetodo    (Tipo_parámetro1    nombre_parámetro1,    Tipo_parámetro2
    nombre_parámetro2,    Tipo_parámetro3    nombre_parámetro3,.....,    Tipo_parámetroN
    nombre_parámetroN)
{
    Cuerpo del método;
}
```

Ejemplo

```
static int multiplica(int a, int b){
    return (a*b);
} //Es obligatorio es uso de return si el método devuelve un valor
```

`static int multiplica(int a,b);` //NO ESTA PERMITIDO en el prototipo del método

Ejemplo

```
public void iniciaDatos (String valorMatricula, String valorTaxista, int valorTipoMotor)
{
    this.matricula=valorMatricula;
    this. taxista=valorTaxista;
    this. tipoMotor=valorTipoMotor;
} //Cierre del método
```



## 5.2. Llamada al método

Es el punto del programa donde se invoca al método. Se realiza mediante el nombre del método y una posible lista de parámetros. Se pueden distinguir dos situaciones:

- Llamada desde la clase a la que pertenece: nombreMetodo(lista\_parametros)
- Llamada desde otra clase: objeto.nombreMetodo(lista\_parametros)

Con respecto al valor de retorno de los métodos, se pueden distinguir dos situaciones también:

- No retorna ningún valor (void) , en este caso se haría la llamada sin más.

Ejemplo:

```
mostrar_menu(); //desde la clase a la que pertenece  
ventana.mostrar_menu(); //desde fuera
```

- Retorna un valor, en este caso el método debe estar incluido en una expresión que recoja el valor retornado.

Ejemplo:

```
x=multiplica(5,7); //desde la clase a la que pertenece.  
x=miObjeto.multiplica(5,7); //desde fuera
```

La lista de parámetros que se pasan en la llamada al método se llama **parámetros actuales** y la lista de parámetros declarados en la declaración se llama **parámetros formales**. Cuando se realiza una llamada a un método **la lista de parámetros actuales ha de coincidir en número, orden y tipo con la lista de parámetros formales**.

## 5.3. Ámbito, alcance y tiempo de almacenamiento de un identificador

El ámbito de un identificador es el bloque en el que es definido.

El alcance de un identificador es toda aquella parte de la aplicación en que es posible referirse a él y que la aplicación lo reconozca.

El **tiempo de almacenamiento** es el tiempo de vida o de duración de la variable en memoria.

El ámbito de una propiedad de una clase es el bloque donde se define esa clase, los métodos la conocen.

También una propiedad se puede utilizar fuera de la clase a través del objeto y siempre que los modificadores de visibilidad le den acceso a ellos.

## 5.4. Variables locales

Una **variable local** está definida en un bloque de sentencias de un método, normalmente todo el método. Su alcance será ese bloque, fuera de él la variable no existe. De igual forma su tiempo de almacenamiento es desde que la ejecución comienza hasta que termina el bloque de código donde está definida. Por tanto, la variable local se crea al entrar la ejecución en el bloque y se destruye cuando sale.

El compilador no da valores iniciales a las variables locales de un método. Por eso es obligatorio darle un valor inicial antes de trabajar con ellas.

```
public class Principal{ //Nombre de la clase donde va estar el método main()  
    public static void main(Strings []args) {  
        int vLocal=2; //Variable local al método main()  
        .....  
        {  
            int p; variable local al bloque de código
```

```
        System.out.println(vLocal2); //dentro de este bloque vLocal se conoce pq se creó en un
bloque padre
    }
    .....
    int x=multiplica(vLocal,5); //llamada al método multiplica, x es local a main()
}
public static int multiplica (int a, int b){ //Parámetros formales, a y b son locales a multiplica
    int m; //Variable local al método multiplica
        m=a*b;
        return m;
    }
}
```

### 5.5. Parámetros formales

Un **parámetro formal** es una variable de enlace entre métodos y está definida dentro de los paréntesis del encabezamiento de la definición del método. Su alcance y tiempo de almacenamiento son los mismos que los expresados para una variable local de un método.

### 5.6. Retorno de un método

Cuando un método termina su tarea devuelve el control de la ejecución al método que lo llamó. Para finalizar el método puede utilizar la palabra clave **return**. Cuando el método está definido con tipo de retorno distinto de **void**, entonces es obligatorio finalizar el método con la palabra clave **return** seguida de una expresión del tipo base que retorna.

Ejemplo

```
void menu(){
    System.out.print("MENU PRINCIPAL\n");
    System.out.print("1-Altas\n");
    System.out.print("2-Bajas\n");
    System.out.print("3-Modificaciones\n");
    System.out.print("0-Terminar\n");
}

int divide(int a, int b){
    if (b==0)
        return 0;
    else
        return (a/b);
}
```

### 5.7. Paso de parámetros.

Los parámetros constituyen la vía de interconexión entre métodos.

#### **Paso por valor.**

Los parámetros formales reciben el valor de los parámetros actuales, creándose una copia del valor almacenado en los parámetros actuales sobre los parámetros formales. Por tanto, los cambios que se produzcan sobre los parámetros formales no afectarán a los parámetros actuales. Solo se aplica a parámetros de tipo primitivo int, float, double, char, etc., y también la clase String.

```
import java.util.Scanner;
public class Principal
{
```

```
public static void main(String []largs){
    Scanner teclado=new Scanner(System.in);
    int n1,n2,n3;
    n1=teclado.nextInt();
    n2=teclado.nextInt();
    n3=potencia(n1,n2);//paso por valor
    System.out.println("El valor de "+n1+" elevado a "+n2+" es: "+n3);
}

public static int potencia(int x, int n){
    int i,resultado=1;
    for(i=1;i<=n;i++)
        resultado *= x;
    return resultado;
}
}
```

### Paso por referencia

Los parámetros formales que sean objetos o arrays, reciben la dirección de memoria de los parámetros actuales y, por tanto, los cambios producidos sobre los valores a los que apuntan los parámetros formales se verán también reflejados en los parámetros actuales, ya que ambos apuntan a la misma dirección de memoria.

```
import java.util.Scanner;
public class Principal
{
    public static void main(String []largs){
        Scanner teclado=new Scanner(System.in);
        int n[]={1,2,3,4};

        incremento1(n);//paso por referencia
        muestraVector(n);//paso por referencia
    }

    public static void incremento1(int x[]){
        for(i=0;i<x.length;i++)
            x[i]++;
    }
    public static void muestraVector(int x[])
    {
        System.out.println("Valores del vector");
        for(i=0;i<x.length;i++) System.out.println(x[i]);
    }
}
```

En cambio, en el siguiente ejemplo, al tratarse de variables de tipo primitivo, las variables no intercambian sus valores.

```
import java.util.Scanner;
public class Principal
{
    public static void main(String []largs){
        Scanner teclado=new Scanner(System.in);
        int n1,n2;
        n1=teclado.nextInt();
        n2=teclado.nextInt();
        intercambio(n1,n2);//paso por valor
        System.out.println("El valor de n1: "+n1+" y de n2: "+n2);
    }
}
```

```

}

public static void intercambio(int x, int n){
    int aux;
    aux=a;
    a=b;
    b=aux;
}
}

```

## 5.8. Tipo de almacenamiento.

### 5.8.1. Variables estáticas.

En Java es posible indicar al compilador además del tipo de variable y su identificador dónde queremos que se almacene la variable. Los modificadores de almacenamiento son:

**auto.** Es el modificador por defecto y no se pone en las variables.

**static.** La variable tiene un tiempo de almacenamiento limitado al tiempo que dura la ejecución del programa, por tanto, se crea cuando se define y se destruye al terminar el programa.

```

public class Principal //Nombre de la clase donde va estar el método main()
{
    public static void main(Strings []args)
    {
        int vLocal=2; //Variable local al método main()
        .....
        muestra(vLocal);
        muestra(vLocal+8);
    }
    public static void muestra (int a) //Parámetro formal, a es local a muestra
    {
        int m; //Variable local al método multiplica
        static int n=0;
        n++;
        System.out.println(n),
        System.out.print(a);
    }
}

```

En la primera llamada a muestra(vLocal), se muestra por pantalla:

0

2

En la segunda llamada a muestra(vLocal+8), se muestra por pantalla:

1

10

### 5.8.2. Propiedades estáticas y Métodos estáticos

Cada objeto tiene su propio espacio de memoria donde guarda sus datos (propiedades); en cambio, de las funciones miembros (métodos) solo existe una copia para todos los objetos que pertenecen a una determinada clase.

Sin embargo, si un dato miembro de una determinada clase se declara **static**, significa que solo existirá una copia de ese dato, el cual será compartido por todos los objetos de esa clase. Además, tiene la cualidad de que este dato existe, aunque no se haya declarado ningún objeto de dicho tipo. Dicho de otra forma, una propiedad declarada **static** es una variable asociada a la clase y no al

objeto.

Un método declarado **static** no es un método del objeto, sino de la clase. Esto permite que se pueda usar dicho método sin tener declarado ningún objeto.

Por ello, la llamada a un método estático se puede hacer poniendo:

*nombreClase.metodoEstatico();*

Esta forma de invocar al método **static** puede resultar engañosa, ya que su actuación no es sobre el objeto en particular, sino sobre todos los objetos de la clase.

Dentro de un método estático solo se puede usar los miembros o propiedades de la clase que sean estáticas.

Clase		
Atributos y métodos static		
Objeto 1	Objeto 2	Objeto 3
Atributos y métodos dinámicos	Atributos y métodos dinámicos	Atributos y métodos dinámicos

```
import java.util.Scanner;
public class Autor {
    private String nombre;
    private double ganancias;
    private static float IRPF;

    public String getNombre(){ return nombre;}
    public void setNombre(String n){nombre=n;}
    public double getGanancias(){ return ganancias;}
    public void setGanancias(double d){ganancias=d;}
    public static float getIRPF(){ return IRPF;}
    public static void setIRPF(float f){IRPF=f;}
    public void pedirDatos()
    {
        Scanner teclado=new Scanner(System.in);
        System.out.print("Nombre del autor: ");
        nombre=teclado.nextLine();
        System.out.print("Ganancias del autor: ");
        ganancias=teclado.nextDouble();
    }
    public static void pedirIRPF()
    {
        System.out.print("IRPF de los autores: ");
        IRPF=Lectura.pedirFloat();
    }
    public void visualDatosAutor()
    {
        System.out.println("Nombre: "+nombre+"\nGanancias: "+ganancias+"\nIRPF: "+IRPF);
    }
}
```

```
package tema4;

public class Principal {

    public static void main(String[] args) {
        Autor a1, a2;
        a1=new Autor();
        a2=new Autor();
        Autor.pedirIRPF();// También a1.pedirIRPF();
        a1.pedirDatos();
        a2.pedirDatos();
        visualizarDosAutores(a1,a2);
        visual("\nVamos a pedir de nuevo el IRPF");
        Autor.pedirIRPF();
        visualizarDosAutores(a1,a2);
    }

    static void visual(String t)
    {
        System.out.println(t);
    }
    static void visualizarDosAutores(Autor a, Autor b)
    {
        visual("Datos del primer autor: ");
        a.visualDatosAutor();
        visual("Datos del segundo autor: ");
        b.visualDatosAutor();
    }
}
```

## 6. Constructores

Los constructores son métodos especiales que puede tener una clase. Son métodos especiales porque se llaman automáticamente cada vez que se instancia un objeto de esa clase, es decir, cuando se hace un **new**.

Su nombre es el mismo que el de la clase y no puede retornar ningún valor, ni siquiera se puede especificar la palabra reservada **void**. Así, un constructor tiene esta cabecera:

*modif\_visibilidad nombreClase(lista\_parámetros);*

La principal utilidad de los constructores es dar valores iniciales a los datos de un objeto cuando se reserva memoria para dicho objeto

Cuando en una clase no hay un constructor, Java asume uno por defecto, da el valor 0 a los atributos numéricos, null a los alfanuméricos y a los objetos.

```
package tema4;

public class Cuenta {
    private String titular;
    private String numero;
    private float saldo;

    public Cuenta(String nombre, String number, float s)
    {
        this.titular=nombre;
        this.numero=number;
        this.saldo=s;
    }

    public Cuenta(Cuenta c)
    {
        this.titular=c.titular;
        this.numero=c.numero;
        this.saldo=c.saldo;
    }
    //Otra forma del constructor anterior
    public Cuenta(Cuenta c)
    {
        this(c.titular,c.numero,c.saldo);
    }
}
```

En el anterior ejemplo, dará un error de codificación porque se han diseñado 2 constructores con el mismo número de parámetros de entrada y del mismo tipo. Es decir, solo se puede definir uno de ellos.

#### ▣ Sobrecarga de métodos.

La sobrecarga hace referencia a la posibilidad de que pueda haber en una misma clase dos o más métodos con el mismo nombre. La única condición es que cada método tenga diferentes parámetros con el fin de que el compilador sepa a qué método tiene que llamar en función de los argumentos que se envíen en la llamada. **Si los métodos se diferencian solo en el tipo de valor devuelto, se producirá un error de codificación y por lo tanto, el compilador no lo admitirá.**

Como el constructor es un método más, también se puede sobrecargar.

```
public class Cuenta {
    String titular;
    String numero;
    float saldo;

    public Cuenta(String nombre, String number, float s)
    {
        this.titular=nombre;
        this.numero=number;
        this.saldo=s;
    }

    public Cuenta(Cuenta c)
    {
        this(c.titular,c.numero,c.saldo);
    }
    public void setCuenta(String nombre, String number, float s)
    {
        titular=nombre;
        numero=number;
        saldo=s;
    }
    public void setCuenta(Cuenta c)
    {
        setCuenta(c.titular,c.numero,c.saldo);
    }

    public String toString()
    {
        return "\nTitular: "+titular+"\nNúmero: "+numero+"\nSaldo: "+saldo;
    }
}
```

```
package tema4;

public class Ppal {
    public static void main(String[] args) {
        Cuenta c1=new Cuenta("Pedro Gómez","12345-34-455",1500);
        Cuenta c2=new Cuenta(c1);
        System.out.print(c1.toString());
        System.out.print(c2.toString());
        c1.setCuenta("David Bustamante","23432-90",4500);
        System.out.print(c1.toString());
        c2.setCuenta(c1);
        System.out.print(c2.toString());
    }
}
```



### 8. La referencia **this**

La palabra **this** es una referencia al propio objeto en el que estamos. Ejemplo:

```
public class Punto {  
    int x, y; // posición del punto  
  
    public Punto(x,y)  
    {  
        this.x=x;  
        this.y=y;  
    }  
  
    Public Punto()  
    {  
        this(0,0);  
    }  
  
    public Punto clone()  
    {  
        Punto p=new Punto(this);  
        return p;  
    }  
  
    public void setX(int x){  
        this.x=x;  
    }  
    public void setY(int y){  
        this.y=y;  
    }  
  
    public void setPunto(int x, int y)  
    {  
        this.x=this.setX(x);  
        this.y=this.setY(y);  
    }  
  
    public void setPunto(Punto p)  
    {  
        this.setPunto(p.x,p.y);  
    }  
}
```

En el ejemplo del primer constructor, hace falta la referencia **this** para clarificar cuando se usan las propiedades x e y, y cuando los argumentos con el mismo nombre.

Los posibles usos de **this** son:

- ✓ **this**. Referencia al objeto actual. Se usa por ejemplo pasarle como parámetro a un método cuando es llamado desde la propia clase.
- ✓ **this.atributo**. Para acceder a una propiedad del objeto actual.
- ✓ **this.método(parámetros)**. Permite llamar a un método del objeto actual con los parámetros indicados.
- ✓ **this(parámetros)**. Permite llamar a un constructor del objeto actual. Esta llamada sólo puede ser empleada en la primera línea de un constructor.

### 9. El método `finalize()`

Cuando no hay ninguna referencia a un objeto determinado, se asume que el objeto no se va a utilizar más, y la memoria ocupada por el objeto se libera.

Pero, algunas veces un objeto necesita realizar una serie de acciones cuando es destruido. Para gestionar estas situaciones, Java proporciona un mecanismo llamado *finalización*. Para ello es necesario definir un método **`finalize()`**.

Formato:

```
protected void finalize(){
```

```
.....  
}
```

Se declara *protected* para evitar que se pueda acceder a `finalize()` desde código definido desde fuera de su clase.

```
public class Hijo {  
    private String nombre;  
    private int anioNac;  
    public int getAnioNac(){ return anioNac;}  
    public Hijo()  
    {...}  
    public void visualDatosHijo()  
    {...}  
    public void visualNombre()  
    {...}  
    public void visualAnioNac()  
    {...}  
    void pedirNombre()  
    {...}  
    void pedirAnioNac()  
    {...}  
    protected void finalize()  
    {  
        Escritura.writeln("Hemos liberado un objeto de la clase Hijo. Sus datos son:");  
        visualNombre();  
        visualAnioNac();  
    }  
}
```

### 10. Parámetros del método `main()`

Es posible pasar parámetros desde la línea de comandos del sistema operativo a la función `main`. Los parámetros que se le pueden indicar son:

**`args`** es una tabla de punteros a cadena de caracteres que contiene desde el nombre del programa y todos los parámetros escritos a continuación separados por un espacio o un tabulador.

Ejemplo,

```
public class NewMain {  
  
    public static void main(String[] args) {  
        for(int i=0; i<args.length; i++)  
            System.out.println("args["+i+"]:"+args[i]);  
    }  
}
```

Al intentar ejecutar este programa introduciendo la siguiente línea de órdenes:

**`java NewMain esto es una prueba 100`**

La salida será:

arg[0]: esto

arg[1]: es

arg[2]: una

arg[3]: prueba

arg[4]: 100

### 11. Funciones miembro recursivas

Se trata de un tipo de funciones miembro que se llaman así mismas. Este tipo de funciones sustituyen a una estructura iterativa. A veces ofrecen una solución más elegante de acuerdo a la definición matemática del problema, pero son más lentas de ejecutar.

En este tipo de métodos es muy importante identificar la condición de finalización de las llamadas recursivas, de lo contrario entraríamos en el temido bucle infinito.

Ejemplo.

```
int factorial(int x){
    if (x==0) //condición de terminación
        return 1;
    else
        return (x*factorial(x-1)); //llamada recursiva
}
```

### 12. Control de acceso

En la siguiente tabla se puede observar la visibilidad de cada especificador:

Zona	private (privado)	Sin modificador (friendly)	protected (protegido)	public (público)
Misma clase	X	X	X	X
Subclase en el mismo paquete		X	X	X
Clase (no subclase) en el mismo paquete		X		X
Subclase en otro paquete			X	X
No subclase en otro paquete				X

### 13. Clases anidadas

El lenguaje de programación Java permite definir una clase dentro de otra clase. Esta clase se denomina *clase anidada* y se ilustra aquí:

```
class ClaseExterna {
    ...
    class ClaseAnidada {
        ...
    }
}
```

**Terminología:** Las clases anidadas se dividen en dos categorías: estáticas y no estáticas. Las clases anidadas que se declaran static se llaman *clases anidadas estáticas*. Las clases anidadas no estáticas se llaman *clases internas*.

```
class ClaseExterna {  
    ...  
    class static ClaseStatica{  
        ...  
    }  
    class ClaseInterna {  
        ...  
    }  
}
```

Una clase anidada es un miembro de su clase envolvente. Las clases anidadas (clases internas) tienen acceso a otros miembros de la clase envolvente, incluso si se declaran `private`.

Como miembro de la `ClaseExterna`, una clase anidada puede ser declarado `private`, `public`, `protected` o *paquete*.

Normalmente se realiza esta práctica para crear objetos internos a una clase (el motor de un coche por ejemplo), de modo que esos objetos pasan a ser atributos de la clase.

Por ejemplo:

```
public class Coche  
{  
    public int velocidad;  
    public Motor motor;  
  
    public Coche(int cil)  
    {  
        motor=new Motor(cil);  
        velocidad=0;  
    }  
  
    public class Motor  
    { //Clase interna  
        public int cilindrada;  
        public Motor(int cil)  
        {  
            cilindrada=cil;  
        }  
    }  
}
```

El objeto motor es un objeto de la clase Motor que es interna a Coche. Si quisiéramos acceder al objeto motor de un coche sería:

```
Coche c=new Coche(1200);  
System.out.println(c.motor.cilindrada); //Saldrá 1200
```

Las clases internas fuera de la clase contenedora no pueden crear objetos (sólo se pueden crear *motores* dentro de un *coche*).

Las clases anidadas estáticas no tienen acceso a otros miembros de la clase que lo contiene. Son accedidas usando:

`ClaseExterna.ClaseAnidadaStatica`

Por ejemplo, para crear un objeto para la clase anidada estática (sí se puede), se usa la sintaxis:

ClaseExterna.ClaseAnidadaStatica objetoAnidado=new ClaseExterna.ClaseAnidadaStatica();

Por ejemplo,

```
//suponiendo que la declaración del Motor dentro de Coche es
// public class static Motor{....
Coche.Motor m=new Coche.Motor(1200);
```

Pero eso sólo tiene sentido si todos los Coches tuvieran el mismo motor.

Y se crearía el objeto: Coche.Motor motorcito=new Coche.Motor();

Cuando se use **this** dentro de una clase interna, this se refiere al objeto de la clase interna (es decir **this** dentro de *Motor* se refiere al objeto *Motor*). Para poder referirse al objeto contenedor (al coche) se usa *Clase.this* (*Coche.this*). Ejemplo:

```
public class Coche
{
    public int velocidad;
    public Motor motor;
    public int cilindrada;

    public Coche(int cil)
    {...}

    public class Motor
    { //Clase interna
        public int cilindrada;
        public Motor(int cil)
        {
            Coche.this.cilindrada=cil;//Coche
            this.cilindrada=cil;//Motor
        }
    }
}
```

Por último las clases internas pueden ser anónimas (se verán más adelante al estar más relacionadas con interfaces y adaptadores).