



DIPLOMADO VIRTUAL EN **PROGRAMACIÓN EN JAVA**

Guía didáctica 3: Programación orientada a objetos (POO)



Formación Virtual

.....educación sin límites



Competencia específica

Se espera que con los temas abordados en la guía didáctica del módulo 3: Programación orientada a objetos (POO), el estudiante logre la siguiente competencia específica:

- Comprender la estructuración de la programación orientada a objetos y sus componentes, aplicando los conocimientos básicos del lenguaje para formular estructuras de datos orientadas a objetos.



Contenidos temáticos

El contenido temático para desarrollar en la guía didáctica del módulo 3: Programación orientada a objetos (POO) es:

```
container">
<div class="row">
  <div class="col-md-6 col-lg-8"> <!--
  <nav id="nav" role="navigation">
    <ul>
      <li><a href="index.html">Home</a>
      <li><a href="home-events.html">Home
      <li><a href="multi-col-menu.html">
      <li class="has-children"> <a href=
        <ul>
          <li><a href="tall-button-h
          <li><a href="image-logo.ht
          <li class="active"><a href=
        </ul>
      </li>
      <li class="has-children"> <a href=
        <ul>
          <li><a href="variable-width
```

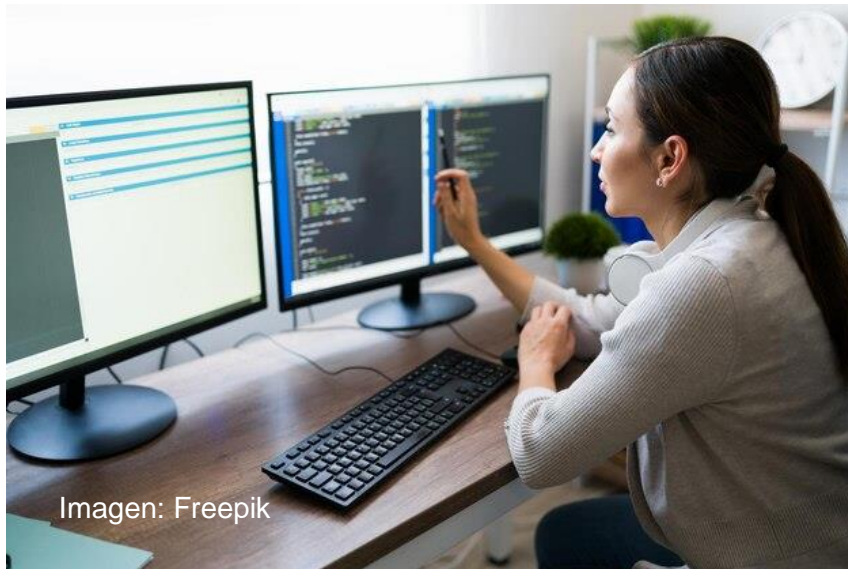
Programación orientada a objetos

Ilustración 1: caracterización de la guía didáctica.

Fuente: autor.

Tema: Programación orientada a objetos

La programación orientada a objetos es un paradigma de la programación, y ¿qué es un paradigma en programación? «Es una propuesta tecnológica adoptada por una comunidad de programadores y desarrolladores cuyo núcleo central es incuestionable en cuanto a que únicamente trata de resolver uno o varios problemas claramente delimitados» (Aprendiendo Arduino, 2019). Es una forma que los programadores emplean para resolver problemas a partir de clases y objetos, es una forma especial de programar, más cercana a como se expresan las cosas en la vida real.



Aclaramos que la programación orientada a objetos no se trata de nuevas características que adquiere el lenguaje de programación. Con la programación orientada a objetos se hace referencia a una nueva forma de pensar. Lo que se acostumbra a ver es la programación estructurada (Wikipedia, 2023a), donde se tiene un problema y se descompone en distintos subproblemas para tener soluciones más pequeñas y simples.

Como mencionamos anteriormente, la programación orientada a objetos busca ver la programación como la vida real, donde los objetos son similares a los objetos en la realidad.

En la programación orientada a objetos, constantemente aludiremos a los conceptos de clases y objetos, pero ¿qué es una clase y qué es un objeto?

Las clases son declaraciones de objetos, también se podrían definir como **abstracciones de objetos o moldes**. Esto quiere decir que la definición de un objeto es una clase. Cuando se programa un objeto y se definen sus características y funcionalidades, en realidad lo que se hace es programar una clase (Poolhjc, 2001).

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

    public void Persona()
    {
        nombre = "Diego";
        pais = "Colombia";
        edad = 22;
    }

    public void Saludar()
    {
        System.out.println("Hola");
    }
}
```

Ilustración 2: ejemplo de clase en Java.

Fuente: Eclipse.

Las clases en Java son básicamente una plantilla que sirve para crear un objeto. Si se imaginan las clases en el mundo en el que se vive, se podría decir que la clase **persona** es una plantilla sobre cómo es un humano. Juan, Andrés, Evelin y Nicol son objetos de la clase **persona**, ya que todos son **personas**. La clase **persona** contiene la definición de un ser humano, mientras que cada ser humano es una instancia u objeto de dicha clase.

En el ejemplo de la clase **persona**, hay varias características que debemos tener en cuenta al momento de trabajar con clases y programación orientada a objetos, dado que entran en juego términos como atributos, métodos, objetos y constructores, todos a partir de una clase.


```
public class Perro
{
    String raza;
    String nombre;
    int edad;
    String encargada;

    public Perro()
    {
        raza = "Perro";
        nombre = "Bruno";
        edad = 3;
        encargada = "Evelin";
    }
    public void ladrar()
    {
        System.out.println("Gau!, gau!");
    }

    public void pasear()
    {
        System.out.println("Ir al parque");
    }
}
```

En este ejemplo, la clase **perro** está compuesta de cuatro atributos: **raza**, **nombre**, **edad** y **encargada** o **dueña**, que son inicializados por medio del constructor de la clase **perro**, además de contener dos métodos *void* sin retorno que describen dos acciones: **ladrar** y **pasear**.

Hay conceptos nuevos que se desarrollarán en el transcurso de la guía, como los siguientes:

Características de una clase

Existe una estructura fundamental para todas las clases definidas en Java que debemos tener en cuenta al momento de construirlas:

Nombre: identifica la clase de forma única en nuestro proyecto.

Atributos: referencia los campos y variables de la clase que permiten definir las características de una clase.

Los atributos hacen el papel de variables en las clases, tienen las mismas características y restricciones, pero en algunos casos cuentan con variantes y complementos.

En el ejemplo anterior, los atributos que componen la clase son:

```
String nombre;  
String pais;  
int edad;
```

Las características recomendadas para la implementación de los atributos en Java son las siguientes:

- Una clase puede contener N atributos.
- Se recomienda el uso de los modificadores de acceso para limitar el alcance de este, en especial *private*.
- Nombres claros en relación con la clase.
- En la mayoría de los casos no se inicializan los atributos.
- Si se inicializa un atributo, se recomienda el uso de la palabra reservada **final** para determinarse como constante.

```
final private String raza = "Perro";  
private String nombre;  
private int edad;  
private String encargada;
```

Ilustración 3: atributos de una clase en Java.

Fuente: Eclipse.

Getters y setters

Los *setters* y *getters* son métodos de acceso, generalmente son una interfaz pública para cambiar atributos privados de las clases, dado que, cuando se determinan atributos privados, no hay manera de acceder a ellos sin un método de acceso:

Setters: hacen referencia a la acción de «establecer», sirven para asignar un valor inicial a un atributo, pero de forma explícita; además, el *setter* nunca retorna nada (siempre asigna) y solo permite dar acceso público a ciertos atributos que el usuario pueda modificar.

```
public void setNombre(String nombre) {  
    this.nombre = nombre;  
}
```

Los métodos *setters* se componen de las siguientes características:

- Suelen ser públicos, puesto que dan acceso a los atributos privados.
- La palabra *void* indica que el método no tendrá retorno, solo realizará la acción de asignar valores.
- El nombre suele estar compuesto anteponiendo la palabra *set* y el nombre del atributo.
- Contienen un parámetro del mismo tipo del método que se pasa argumentado desde la instancia de la clase (*string* nombre).
- La palabra *this* ayuda a diferenciar el atributo de la clase y el parámetro, dado que sin este se estaría operando únicamente con el parámetro, dejando a un lado los atributos.

```
public void setNombre(String nombre) {  
    nombre = nombre;  
}
```

La implementación con base en un objeto creado de la clase perro sería:

```
public static void main(String[] args)  
{  
    Perro firulais = new Perro();  
    firulais.setNombre("firulais");  
}
```

De esta forma, el **atributo** nombre del **objeto** firulais toma el valor de "firulais".

Getters: hacen referencia a la acción de obtener, sirven para obtener (recuperar o acceder) el valor ya asignado a un atributo y ser utilizado.

```
public String getNombre() {  
    return nombre;  
}
```

Los métodos *getters* se componen de las siguientes características:

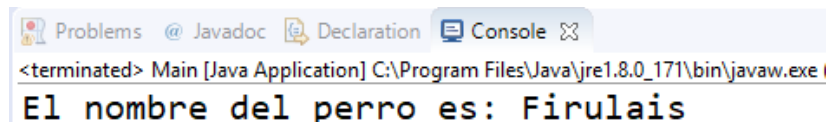
- Suelen ser públicos, puesto que dan acceso a los atributos privados.

- Deben tener un tipo de dato asignado al tipo de atributo que representan. Si el atributo es de tipo *string*, el método *get* debe ser del mismo tipo.
- El nombre suele estar compuesto anteponiendo la palabra *get* y el nombre del atributo.
- No reciben parámetros, dado que únicamente cumplen la función de retornar valores ya establecidos en la clase (los atributos por medio de *set* o ya declarados como constantes).
- Retornan el valor del atributo que representa.
- No hace falta usar *this*, dado que no se reciben parámetros que se relacionen con argumentos.

```
public String getNombre() {  
    return nombre;  
}
```

La implementación con base en un objeto creado de la clase perro sería:

```
public static void main(String[] args)  
{  
    Perro Firulais = new Perro();  
  
    Firulais.setNombre("Firulais");  
    System.out.println("El nombre del perro es: " + Firulais.getNombre());  
}
```



Problems @ Javadoc Declaration Console

<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe

El nombre del perro es: Firulais

De esta forma, el **atributo** nombre del **objeto** Firulais retornará el valor que fue seteado (*set*) previamente.

Es necesario tener en cuenta que deben existir dos clases para el funcionamiento, en este caso, la clase perro con todas las propiedades y otra clase con un método principal para poder realizar todos los procesos a partir de esta.


```
public class Perro
{
    final private String raza = "Perro";
    private String nombre;
    private int edad;
    private String encargada;

    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public int getEdad() {
        return edad;
    }
    public void setEdad(int edad) {
        this.edad = edad;
    }
    public String getEncargada() {
        return encargada;
    }
    public void setEncargada(String encargada) {
        this.encargada = encargada;
    }
    public String getRaza() {
        return raza;
    }
}
```

Ilustración 4: métodos *getters* y *setters* de la clase perro.

Fuente: Eclipse.

This

Sirve para hacer referencia a un método, propiedad o atributo del objeto actual. Se utiliza principalmente cuando existe sobrecarga de nombres. La sobrecarga de nombres se da cuando hay una variable local de un método o constructor, o un parámetro formal de un método o constructor, con un nombre idéntico al que está presente en la clase en el momento de relacionarse.

```
public Perro(String nombre, int edad, String encargada) {
    this.nombre = nombre;
    this.edad = edad;
    this.encargada = encargada;
}
```

Ilustración 5: *this* en Java.

Fuente: Eclipse.

En el constructor de la clase perro, **this** ayuda a diferenciar los parámetros de los atributos, dado que estos contienen el mismo nombre.

Parámetros y argumentos

Los parámetros o argumentos son una forma de intercambiar información con el método. Pueden servir para introducir datos para ejecutar el método (entrada) o para obtener o modificar datos tras su ejecución (salida).

Parámetros: uso en la declaración del método.

```
public Carro(String marca, String modelo, String color, boolean enVenta) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.color = color;  
    this.enVenta = enVenta;  
}
```

Argumentos: uso en el paso de datos a métodos desde un objeto.

```
public static void main(String[] args)  
{  
    Carro Tracker = new Carro("Chevrolet", "Negro", "Tracker LT", false);  
}
```

Parámetros: son los valores que un método recibe desde un objeto.

```
//Constructor Parametros que debe recibir el método constructor  
public Carro(String marca, String modelo, String color, boolean enVenta) {  
    this.marca = marca;  
    this.modelo = modelo;  
    this.color = color;  
    this.enVenta = enVenta;  
}
```

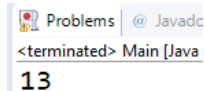
Argumentos: son los valores que un objeto recibe para operar un método.

```
public static void main(String[] args)  
{  
    //Clase Objeto                                Argumentos al método constructor  
    Carro Tracker = new Carro("Chevrolet", "Negro", "Tracker LT", false);  
}
```

Paso por valor: son argumentos de tipo primitivo que contienen el valor exacto de estos.

```
public static void main(String[] args)
{
    Calculadora Operaciones = new Calculadora();
    int resultado = Operaciones.sumar(9, 4);
    System.out.println(resultado);
}

public int sumar(int numero1, int numero2)
{
    return numero1+numero2;
}
```



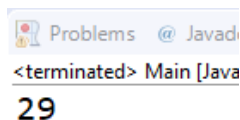
Problems @ Javadc
<terminated> Main [Java]
13

El paso por valor se caracteriza por el uso de tipos primitivos que permiten el paso de valores exactos a los métodos.

Paso por referencia: son argumentos de tipo objeto, arreglo, entre otros, que no contienen los valores de estos, sino su respectiva dirección en memoria.

```
public static void main(String[] args)
{
    Calculadora Operaciones = new Calculadora();
    int numeros[] = {9,2,8,1,0,9};
    int resultado = Operaciones.sumarVector(numeros);
    System.out.println(resultado);
}

public int sumarVector(int numeros[])
{
    int suma = 0;
    for(int i = 0; i < numeros.length; i++)
    {
        suma = suma + numeros[i];
    }
    return suma;
}
```



Problems @ Javadc
<terminated> Main [Java]
29

El paso por referencia se caracteriza por el uso de tipos datos por referencia que permiten el paso de la dirección de memoria del valor original, mas no la copia del valor.

Constructores

Es un método que contiene las acciones que se realizarán por defecto al crear un objeto, en la mayoría de los casos, se inicializan los valores de los atributos en el constructor.

- No es obligatoria su creación.
- El nombre debe ser el mismo de la clase.
- No retorna ningún valor.
- Pueden existir varios constructores (solo se deben diferenciar en sus parámetros).
- Dado el caso de existir varios constructores, solo se ejecutará uno de estos.
- Se recomienda el uso de modificadores de accesos.

```
public Perro(String nombre, int edad, String encargada) {  
    this.nombre = nombre;  
    this.edad = edad;  
    this.encargada = encargada;  
}
```

Ilustración 6: ejemplo de un constructor.

Fuente: Eclipse.

El constructor recibe todos los atributos de la clase en su inicialización. Desde la declaración de la instancia u objeto, sería de la siguiente forma:

```
public static void main(String[] args)  
{  
    Perro Firulais = new Perro("Firulas", 9, "Ana");  
}
```

El objeto debe recibir tres argumentos del mismo tipo y en el mismo orden conforme se declara dentro del constructor, la palabra **perro** hace las veces de tipo de dato de la clase **perro** para declarar el objeto, la instrucción **new Perro(..)** se encarga de enviar los argumentos al constructor para ser inicializados los atributos.

Para el uso de correcto de los constructores, debemos entender el uso y la aplicación de los **argumentos** y **parámetros** que ayudan inicializar esos valores.

Veamos:

```
public class Calculadora
{
    private int numero1;
    private String operador;
    private int numero2;

    public Calculadora(int numero1, String operador, int numero2)
    {
        this.numero1 = numero1;
        this.operador = operador;
        this.numero2 = numero2;
    }
}
```

La clase calculadora es un ejemplo básico que permitirá describir adecuadamente el funcionamiento del constructor. La clase cuenta con tres atributos, todos privados y sin inicializar sus respectivos valores, dado que eso se hará dentro del constructor.

Este, para ser un constructor, debe cumplir con unas características básicas para el correcto funcionamiento:

- Es público.
- El nombre es el mismo de la clase donde está contenido (calculadora).
- Puede o no recibir parámetros.
- No cuenta con retorno.

Entendamos el funcionamiento dentro del constructor: este recibe unos parámetros desde la instancia u objeto que lo invoca que permiten asignar a los atributos privados unos valores en tiempo de ejecución.

Veamos los parámetros y argumentos:

```
public class Main {

    public static void main(String[] args)
    {
        Calculadora Operaciones = new Calculadora(9, "+", 8);
    }

}
```

En la clase *main* se creará la respectiva instancia de la clase operaciones, por lo que Java exigirá unos argumentos.

Recordemos que los argumentos son los valores que se le asignan al método llamado por el objeto y los parámetros son los valores que recibe el método en su declaración (más adelante profundizaremos en esto).

Estos argumentos son transformados en parámetros en el método de forma de objeto, y el constructor de la clase cuenta ahora con los siguientes valores:

```
private int numero1; // 9
private String operador; // +
private int numero2; // 8
```

Ilustración 7: asignación de valores a parámetros.

Fuente: Eclipse.

Puesto que el constructor les asignó los valores a los atributos del presente objeto por medio de la siguiente operación:

```
                //Parámetros enviados desde la clase Main
public Calculadora(int numero1, String operador, int numero2)
{
    //Atributos    //Parámetros
    this.numero1 = numero1;
    //Atributos    //Parámetros
    this.operador = operador;
    //Atributos    //Parámetros
    this.numero2 = numero2;
}
```

En esta operación de asignación convergen los argumentos y los parámetros, dado que los valores de los parámetros los deben tomar dichos argumentos que se desean inicializar.

This cumple el papel de diferenciar un atributo de un parámetro, puesto que por convención es ideal mantener los nombres, *this* permite hacer la referencia al objeto actual (es decir, la clase).

En el ejercicio anterior, el constructor recibía todos los parámetros, ¿cómo funcionan los constructores vacíos?

```
public class Calculadora
{
    private int numero1;
    private String operador;
    private int numero2;

    public Calculadora()
    {

    }
}
```

Ilustración 8: constructor vacío.

Fuente: Eclipse.

Es admisible la declaración de un constructor vacío, pero ¿cómo se inicializarán los valores? Veamos el ejercicio aplicando *getters* y *setters*.

```
public int getNumero1() {
    return numero1;
}

public void setNumero1(int numero1) {
    this.numero1 = numero1;
}

public String getOperador() {
    return operador;
}

public void setOperador(String operador) {
    this.operador = operador;
}

public int getNumero2() {
    return numero2;
}

public void setNumero2(int numero2) {
    this.numero2 = numero2;
}

public String operacion()
{
    return this.getNumero1() + " " + this.getOperador() + " " + this.getNumero2();
}
```

Se implementan los métodos *getter* (obtener) y *setter* (asignar) para cada atributo de la clase. Estos permiten interactuar con atributos primitivos.

```
public static void main(String[] args)
{
    Calculadora Operaciones = new Calculadora();
    Operaciones.setNumero1(9);
    Operaciones.setOperador("+");
    Operaciones.setNumero2(8);
    System.out.println(Operaciones.operacion());
}
```

Ilustración 9: asignación de setters.

Fuente: Eclipse.

Por medio del objeto se realiza la asignación de los valores (*notemos que en la creación del objeto no asignamos argumentos al constructor, puesto que este está definido como vacío*).

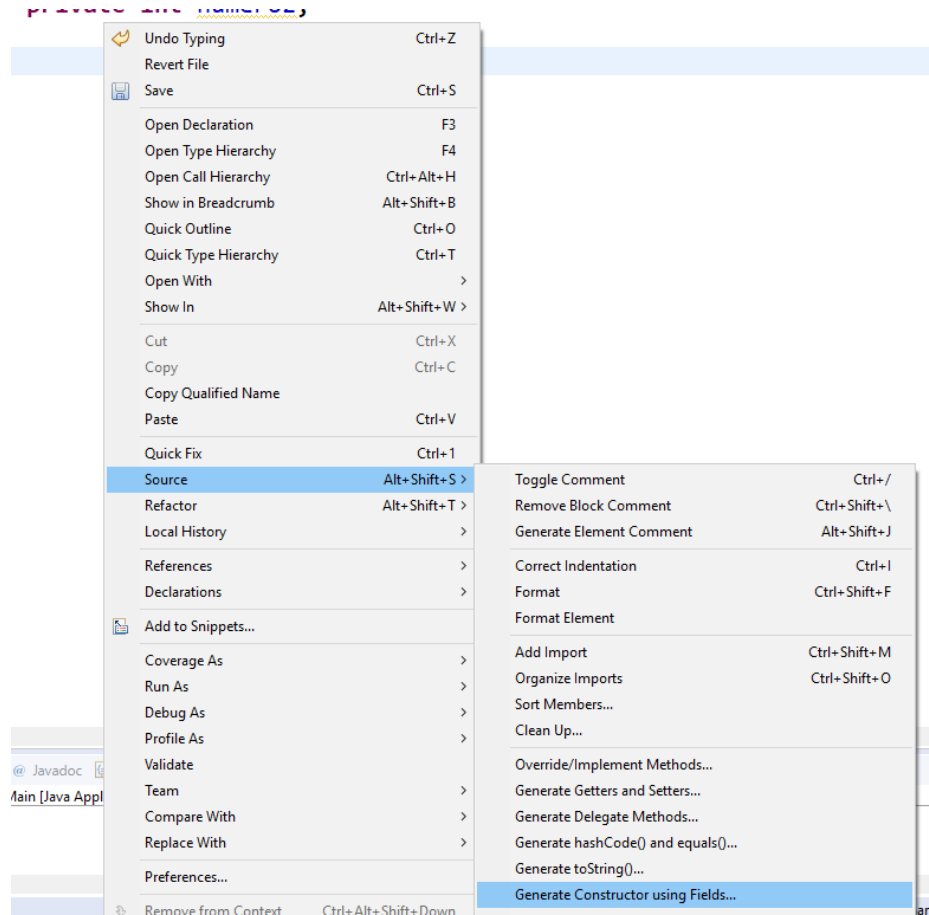
Java tiene atajos que permiten ahorrar tiempo a la hora de generar algunos bloques de código: *getters*, *setters* y constructores, entre otros. Veamos cómo generar un constructor.

Partiendo de los atributos del ejercicio anterior, generar un constructor con todos sus atributos.

```
public class Calculadora
{
    private int numero1;
    private String operador;
    private int numero2;
}
```

Los pasos para llevar a cabo son los siguientes:

- Clic derecho sobre el editor de la clase.
- Opción: *source*.
- Opción: *Generate Constructor Using Fields*.



```
public class Calculadora
{
    private int numero1;
    private String operador;
    private int numero2;

    public Calculadora(int numero1, String operador, int numero2) {
        this.numero1 = numero1;
        this.operador = operador;
        this.numero2 = numero2;
    }
}
```

Ilustración 10: constructor automático.
Fuente: Eclipse.

Sobrecarga de constructores

Permite definir más de un constructor con el mismo nombre, con la condición de que no puede haber dos de ellos con el mismo número y tipo de parámetros.

```
public Perro()
{

}

public Perro(String nombre, int edad, String encargada) {
    this.nombre = nombre;
    this.edad = edad;
    this.encargada = encargada;
}

public Perro(String nombre)
{
    this.nombre = nombre;
}

public static void main(String[] args)
{
    Perro Firulais = new Perro();
    Perro Peluche = new Perro("Peluche", 9, "Yulied");
    Perro Poseidon = new Perro("Poseidon");
}
```

Ilustración 11: ejemplo de sobrecarga de constructores.

Fuente: Eclipse.

Métodos

Procesos o acciones disponibles para el objeto creados a partir de una clase.

«Un método es una abstracción de una operación que puede hacer o realizarse con un objeto. Una clase puede declarar cualquier número de métodos que lleven a cabo operaciones de todo tipo con los objetos». (Garro, 2014a)

Algunas características que se encuentran dentro de los métodos son:

- Permiten reutilizar código.
- Pueden o no retornar valores, los métodos que no retornan valores son conocidos como métodos de tipo *void*, a diferencia que los métodos que sí retornan y son de un tipo en específico: *int*, *string*, entre otros.
- Pueden existir N cantidades de métodos dentro de una clase.
- Pueden contener o no parámetros.
- Un método puede contener N parámetros, aunque se recomienda no sobrecargar los métodos.
- Los métodos deben retornar un tipo de dato del mismo tipo que está diseñado el método, es decir, un método de tipo entero debe retornar un entero.

- Los nombres deben ser muy claros y descriptivos con la acción que van a realizar.
- Se recomienda el uso de los modificadores de acceso, en especial, diseñar los métodos con el modificador de acceso *private*.

La estructura general para implementar los métodos dentro de las clases es la siguiente:

- Modificador de acceso.
- Tipo de dato.
- Nombre de método.
- Parámetros.

```
public class Casa {  
  
    private String color;  
    private int cuartos;  
    private int habitantes;  
    private String ciudad;  
    private int precio;  
    private String propietario;  
}
```

Ilustración 12: atributos de la clase casa de ejemplo.
Fuente: Eclipse.

Métodos *void*

La utilidad de los métodos *void* radica en que son métodos que no cuentan con ningún tipo de retorno.

Algunas características de este tipo de método son:

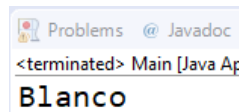
- Se centra en realizar acciones que no requieren retornar un valor en específico, también suele ser usado para mostrar mensajes.
- Se caracteriza por no tener un tipo de dato asociado.
- Siempre contiene la palabra *void*.
- El modificador de acceso más común es *public*.
- Puede o no recibir parámetros.
- Los nombres deben ser muy claros y descriptivos con la acción que van a realizar.

Observemos algunos ejemplos de métodos *void*:

El método `pintarDeBlanco ()` cumple la función de asignar un valor al atributo `color` de la clase `Casa` a partir del objeto `miCasa`.

```
public void pintarDeBlanco()
{
    color = "Blanco";
}

public static void main(String[] args)
{
    Casa miCasa = new Casa();
    miCasa.setColor("Verde");
    miCasa.pintarDeBlanco();
    System.out.println(miCasa.getColor());
}
```



Problems @ Javadoc
<terminated> Main [Java Ap
Blanco

El método `cambiarDePropietario ()` cumple la función de asignar un nuevo valor al atributo `propietario` de la clase `Casa` por medio del parámetro que recibe (recordemos el uso de *this*).

```
public void cambiarPropietario(String propietario)
{
    this.propietario = propietario;
}

public static void main(String[] args)
{
    Casa miCasa = new Casa();
    miCasa.setPropietario("Juan");
    miCasa.cambiarPropietario("Diego");
    System.out.println(miCasa.getPropietario());
}
```



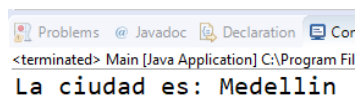
Problems @ Javac
<terminated> Main [Jav
Diego

Ilustración 13: métodos casa.
Fuente: Eclipse.

El método `mostrarCiudad ()` cumple la función de mostrar un mensaje de la ciudad de la casa con base en el atributo de ciudad.

```
public void mostrarCiudad()
{
    System.out.println("La ciudad es: " + ciudad);
}

public static void main(String[] args)
{
    Casa miCasa = new Casa();
    miCasa.mostrarCiudad();
}
```



Problems Javadoc Declaration Cor
<terminated> Main [Java Application] C:\Program Fil
La ciudad es: Medellin

Métodos de tipo

La utilidad de los métodos de tipo radica en que son métodos que cuentan un retorno con base en el tipo de dato que fue declarado.

Algunas características de este tipo de método son:

- Se centra en realizar acciones que deben contar con un retorno obligatorio de acuerdo con el tipo de dato.
- Se caracteriza por tener un tipo de dato asociado.
- El modificador de acceso más común es *public*.
- Puede o no recibir parámetros.
- Los métodos *get* son un ejemplo de métodos de tipo.
- Se debe tener en cuenta la correcta identificación de los nombres de los métodos.
- Los nombres deben ser muy claros y descriptivos con la acción que van a realizar.

Observemos algunos ejemplos de métodos de tipo:

El método `aumentarPrecio ()` cumple la función de incrementar y retornar el atributo del precio de la clase casa partiendo del parámetro que recibe (recordemos el uso de *this*).

```
public int aumentarPrecio(int precio)
{
    return this.precio = this.precio + precio;
}

public static void main(String[] args)
{
    Casa miCasa = new Casa();
    miCasa.setPrecio(100000);
    miCasa.aumentarPrecio(150000);
    System.out.println(miCasa.getPrecio());
}
```

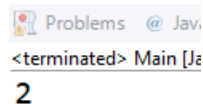


Problems @ Javadoc
<terminated> Main [Java Applic
250000

El método cantidadDeCuartosPorHabitante () cumple la función de retornar la cantidad de habitaciones que hay por cada habitante de la casa, basándose en los atributos de cuartos y habitantes.

```
public int cantidadDeCuartosPorHabitante()
{
    return cuartos/habitantes;
}

public static void main(String[] args)
{
    Casa miCasa = new Casa();
    miCasa.setCuartos(4);
    miCasa.setHabitantes(2);
    System.out.println(miCasa.cantidadDeCuartosPorHabitante());
}
```




Problems @ Jav.
<terminated> Main [J
2

El método valorCasa () cumple la función de retornar el precio de la casa con base en el atributo de la clase casa.

```
public int valorCasa()
{
    return precio;
}
```

```
public static void main(String[] args)
{
    Casa miCasa = new Casa();
    miCasa.setPrecio(25000000);
    System.out.println(miCasa.valorCasa());
}
```



Problems @ Javadoc
<terminated> Main [Java Applica
25000000

Modificadores de acceso

Los modificadores de acceso introducen el concepto de encapsulamiento. El encapsulamiento busca controlar el acceso a los datos que conforman un objeto o instancia; de este modo, una clase y, por ende, sus objetos, que hacen uso de modificadores de acceso (especialmente privados), son objetos encapsulados.

Los modificadores de acceso permiten dar un nivel de seguridad mayor al restringir el acceso a diferentes atributos, métodos, constructores, de manera que aseguran que el usuario deba seguir una «ruta» especificada para acceder a la información.

Implementando el uso de los modificadores de acceso se podrá asegurar que un valor no será modificado incorrectamente. Generalmente el acceso a los atributos se consigue por medio de los métodos *get* y *set*, es estrictamente necesario que los atributos de una clase sean privados.

- Siempre se recomienda que los atributos de una clase sean privados y, por tanto, cada atributo debe tener sus propios métodos *get* y *set* para obtener y establecer respectivamente el valor del atributo.

```
public class Carro
{
    private String marca;
    private String modelo;
    private String color;
    private boolean enVenta;
}
```



```
public class Carro
{

    private String marca;
    private String modelo;
    private String color;
    private boolean enVenta;

    public String getMarca() {}
    public void setMarca(String marca) {}

    public String getModelo() {}
    public void setModelo(String modelo) {}

    public String getColor() {}
    public void setColor(String color) {}

    public boolean isEnVenta() {}
    public void setEnVenta(boolean enVenta) {}

}
```

- Siempre que se use una clase de otro paquete se debe importar usando *import*. Cuando dos clases se encuentran en el mismo paquete, no es necesario hacer el *import*, pero esto no significa que se pueda acceder a sus componentes directamente. (Meza, 2018b).

```
public class Main {

    public static void main(String[] args)
    {
        Carro Tracker = new Carro();
    }

}
```

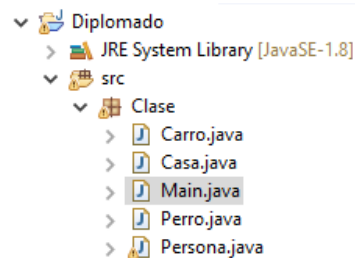


Ilustración 14: orden del proyecto.
Fuente: Eclipse.

En este primer ejemplo, al coincidir ambas clases dentro del mismo paquete, el uso de importar la clase *carro* dentro de la clase *main* sobra, dado que se encuentran al mismo nivel. Veamos un caso en el que ambas clases se encuentran en diferentes paquetes y el uso del *import* se hace obligatorio.

```
package OtroPaquete;

import Clase.Carro;

public class Main {

    public static void main(String[] args)
    {
        Carro Tracker = new Carro();
    }
}
```

Ilustración 15: objeto tipo carro.

Fuente: Eclipse.

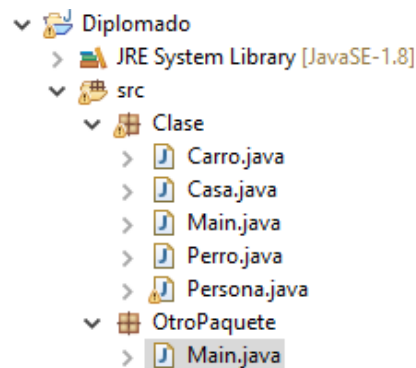


Ilustración 16: distribución por paquetes.

Fuente: Eclipse.

Existen dos paquetes (*Clase* y *OtroPaquete*); *clase* contiene a *carro* y otro paquete contiene la clase *main*, donde se utiliza la clase *carro*, por lo que para hacer uso de esta se debe recurrir a *import* para referenciar dicha clase.

Existen cuatro modificadores de acceso:

- **Private:** es el modificador más restrictivo y especifica que los elementos que lo utilizan solo pueden ser accedidos desde la misma clase en la que se encuentran. Este modificador únicamente puede utilizarse sobre los

miembros de una clase y sobre interfaces y clases internas, no sobre clases o interfaces de primer nivel, dado que esto no tendría sentido.

- **Protected:** indica que los elementos solo pueden ser accedidos desde su mismo paquete y desde cualquier clase que extienda la clase en que se encuentra, independientemente de si esta se encuentra en el mismo paquete o no. Este modificador, como *private*, no tiene sentido a nivel de clases o interfaces no internas.
- **Public:** este nivel de acceso permite acceder al elemento desde cualquier clase, independientemente de que esta pertenezca o no al paquete en que se encuentra el elemento.
- **Default:** si no se determina ningún modificador, se usa el de por defecto, que solo puede ser accedido por clases que están en el mismo paquete.

Visibilidad	<i>Public</i>	<i>Private</i>	<i>Protected</i>	<i>Default</i>
Desde la misma clase	Sí	Sí	Sí	Sí
Desde cualquier clase del mismo paquete	Sí	No	Sí	Sí
Desde una subclase del mismo paquete	Sí	No	Sí	Sí
Desde cualquier clase fuera del paquete	Sí	No	Sí, a través de herencia	No
Desde cualquier subclase fuera del paquete	Sí	No	No	No

Tabla 1: modificadores de acceso.

Fuente: autor.

Return

La palabra reservada *return* permite retornar valores dentro de los métodos, además de detener la ejecución del mismo.

- Cualquier instrucción que se encuentre después de la ejecución de *return* NO será ejecutada. Es común encontrar métodos con múltiples sentencias *return* dentro de condicionales, pero, una vez que el código ejecuta una sentencia *return*, lo que haya de allí hacia abajo no se ejecutará.

- El tipo del valor que se retorna en un método debe coincidir con el del tipo declarado del método, es decir, si se declara *int* el método, el valor retornado debe ser un número entero.
- En el caso de los métodos *void* (sin retorno), se pueden usar la sentencia *return*, pero sin ningún tipo de valor, solo se usaría como una manera de terminar la ejecución del método.

Sobrecarga de métodos

Permite definir más de un método con el mismo nombre, con la condición de que no puede haber dos de ellos con el mismo número y tipo de parámetros.

El uso de la sobrecarga de métodos se ve sujeto a las siguientes condiciones:

- Debe tener el mismo nombre en los métodos.
- No puede haber dos métodos con el mismo nombre y el mismo tipo con igual número de parámetros; se permite el mismo nombre y mismo tipo, siempre y cuando el número de parámetros sea diferente.
- Suele ser utilizada para sobrecarga de métodos con métodos de tipo.
- Java desde la instancia, a la hora de usar los métodos, los diferenciará basándose en los parámetros que este reciba.

```
public class Calculadora
{
    public int sumar(int numero1, int numero2)
    {
        return numero1+numero2;
    }

    public double sumar(double numero1, double numero2)
    {
        return numero1+numero2;
    }

    public float sumar(float numero1, float numero2)
    {
        return numero1+numero2;
    }

    public int sumar()
    {
        return 0;
    }

    public int sumar(int numero1, int numero2, int numero3)
    {
        return numero1+numero2+numero3;
    }
}
```

```
public static void main(String[] args)
{
    Calculadora Operaciones = new Calculadora();

    Operaciones.sumar();
    Operaciones.sumar(9.2, 3.7);
    Operaciones.sumar(9f, 9.5f);
    Operaciones.sumar(5, 2);
    Operaciones.sumar(2, 3, 4);
}
```

Ilustración 17: clase calculadora.

Fuente: Eclipse.

Objetos

Los objetos son ejemplares de una clase. Cuando se crea un objeto se debe especificar la clase a partir de la cual se creará. Esta acción de crear un objeto a partir de una clase se llama **instanciar**.

```
package Clase;

public class Main {

    public static void main(String[] args)
    {
        Persona Diego = new Persona();
    }
}
```

Ilustración 18: objetos.

Fuente: Eclipse.

Los objetos poseen características fundamentales que nos permiten conocerlos mediante la observación, la identificación y el estudio posterior de su comportamiento. Veamos estas características:

- **Identidad:** es la propiedad que permite diferenciar un objeto y distinguirse de otros. Por lo general, esta propiedad es tal que da nombre al objeto. Por ejemplo, el «verde» como un objeto concreto de una clase color; la propiedad que da identidad única a este objeto es precisamente su «color» verde. Tanto es así que no tiene sentido usar otro nombre para el objeto que no sea el valor de la propiedad que lo identifica.

En programación, la identidad de todos los objetos es útil para comparar si dos objetos son iguales o no. Es común encontrar que en

muchos lenguajes de programación la identidad de un objeto esté determinada por la dirección de memoria de la computadora en la que se encuentra el objeto, pero este comportamiento puede ser variado redefiniendo la identidad del objeto a otra propiedad (Wikipedia, 2023b).

- **Comportamiento:** el comportamiento de un objeto está directamente relacionado con su funcionalidad y determina las operaciones que este puede realizar. La funcionalidad de un objeto está definida por su responsabilidad. **Una de las ventajas fundamentales de la programación orientada a objetos es la reusabilidad del código;** un objeto es más fácil de reutilizarse en tanto su responsabilidad sea mejor definida y más concreta.

Una tarea fundamental al diseñar una aplicación informática es determinar el comportamiento que tendrán los objetos de las clases involucradas en la aplicación, asociando la funcionalidad requerida con la aplicación a las clases adecuadas (Wikipedia, 2023b).

- **Estado:** el estado de un objeto se refiere al conjunto de atributos y sus valores en un instante de tiempo dado. El comportamiento de un objeto puede modificar su estado (Wikipedia, 2023).

Para declarar un objeto de una clase específica, el proceso es el siguiente:

```
public static void main(String[] args)
{
    //Variable de tipo persona
    Persona Diego;
    //Creación del objeto de la clase persona
    Diego = new Persona();
}
```

Con la declaración **Persona Diego** se está reservando un espacio de memoria para almacenar una referencia (dirección de memoria) a un objeto de la clase **persona**. Al respecto, es importante comprender que **Diego** no es un objeto, sino una variable que almacenará la referencia a un objeto (de la clase **persona**) que todavía no existe. Seguidamente, mediante la sentencia **Diego = new Persona** (**()**), el operador *new* creará un objeto de la clase **persona**, reservando memoria para

guardar sus atributos. Finalmente, con el operador de asignación (=), la dirección de memoria donde esté creado el objeto es asignada a **Diego**.

Otros aspectos para tener en cuenta en POO con los objetos:

- Se pueden instanciar N cantidad de objetos de una casa, siempre y cuando tengan nombres diferentes.
- Un objeto puede acceder a todos métodos definidos en la clase de la cual está generada la instancia a partir de los modificadores de acceso.

En la clase persona que se utiliza para ilustrar el ejemplo del objeto **Diego**, se encuentra la siguiente estructura que se compone por los atributos y métodos que se encargan de definir toda la estructura funcional de la misma.

```
public class Persona
{
    private String nombre;
    private String pais;
    private int edad;

    public Persona()
    {

    }

    public String getNombre() { }

    public void setNombre(String nombre) { }

    public String getPais() { }

    public void setPais(String pais) { }

    public int getEdad() { }

    public void setEdad(int edad) { }

    public void Comer()
    {
        System.out.println("Comer");
    }

    public void Cantar()
    {
        System.out.println("Cantar");
    }
}
```

Ilustración 19: declaración de métodos.

Fuente: Eclipse.

Para realizar el proceso de instanciar un objeto de tipo persona, se presenta de la siguiente forma:

```
public class Main
{
    public static void main(String[] args)
    {
        Persona Diego = new Persona();
        Persona Katt = new Persona();
    }
}
```

Ilustración 20: declaración de objetos.

Fuente: Eclipse.

Con ambos objetos instanciados de la clase persona, se puede ahora operar con los métodos y atributos establecidos en la clase.

```
public static void main(String[] args)
{
    Persona Diego = new Persona();

    Diego.setNombre("Diego Alejandro");
    Diego.setEdad(22);
    Diego.setPais("Colombia");

    Persona Katt = new Persona();

    Katt.setNombre("Katt");
    Katt.setEdad(19);
    Katt.setPais("España");
}
```

Ilustración 21: uso de objetos.

Fuente: Eclipse.

Se establecen valores a los atributos por medio del método *set*. Debemos considerar que los atributos de un objeto serán diferentes al del otro, dado que cada uno es independiente del otro.

```
public static void main(String[] args)
{
    Persona Diego = new Persona();

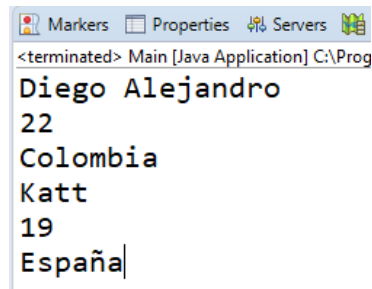
    Diego.setNombre("Diego Alejandro");
    Diego.setEdad(22);
    Diego.setPais("Colombia");

    System.out.println(Diego.getNombre());
    System.out.println(Diego.getEdad());
    System.out.println(Diego.getPais());

    Persona Katt = new Persona();

    Katt.setNombre("Katt");
    Katt.setEdad(19);
    Katt.setPais("España");

    System.out.println(Katt.getNombre());
    System.out.println(Katt.getEdad());
    System.out.println(Katt.getPais());
}
```



Markers Properties Servers
<terminated> Main [Java Application] C:\Prog
Diego Alejandro
22
Colombia
Katt
19
España

Ilustración 22: *getters* + objeto.

Fuente: Eclipse.

Se obtienen los valores de los atributos por medio del método *get*, se debe contemplar que los atributos de un objeto serán diferentes al del otro, dado que cada uno es independiente del otro.

```
public static void main(String[] args)
{
    Persona Diego = new Persona();

    Diego.Cantar();
    Diego.Comer();
}
```

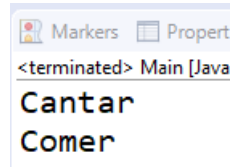


Ilustración 23: métodos + objeto.

Fuente: Eclipse.

Por último, se operan los métodos descritos en la clase **persona** para realizar acciones, en este caso, **cantar** y **comer**.

En programación orientada a objetos, una instancia de programa (por ejemplo, un programa ejecutándose en una computadora) es tratada como un conjunto dinámico de objetos interactuando entre sí. Los objetos en la POO extienden la noción más general descrita en secciones anteriores para modelar un tipo muy específico que está definido fundamentalmente por:

- **Atributos:** representan los datos asociados con el objeto o, lo que es lo mismo, sus propiedades o características. Los atributos y sus valores en un momento dado determinan el estado de un objeto.
- **Métodos:** acceden a los atributos de una manera predefinida e implementan el comportamiento del objeto.

Los atributos y métodos de un objeto están definidos por su clase, aunque una instancia puede poseer atributos que no fueron definidos en su clase. Algo similar ocurre con los métodos: una instancia puede contener métodos que no estén definidos en su clase, de la misma manera que una clase puede declarar ciertos métodos como «métodos de clase», y estos (en dependencia del lenguaje) podrán estar o no presentes en la instancia.

En el caso de la mayoría de los objetos, los atributos solo pueden ser accedidos a través de los métodos; de esta manera es más fácil garantizar que los datos permanecerán siempre en un estado bien definido (invariante de clase).

En un lenguaje en el que cada objeto es creado a partir de una clase, un objeto es llamado una instancia de esa clase. Cada objeto pertenece a un tipo y dos objetos que pertenezcan a la misma clase tendrán el mismo tipo

de dato. Crear una instancia de una clase es entonces referido como instanciar la clase.

En casi todos los lenguajes de programación orientados a objeto, el operador «punto» (.) es usado para referirse o «llamar» a un método particular de un objeto. Un ejemplo de lenguaje que no siempre usa este operador es C++, ya que para referirse a los métodos de un objeto a través de un puntero al objeto se utiliza el operador (->) (Wikipedia, 2023b).

Consideremos como ejemplo una clase aritmética llamada aritmética. Esta clase contiene métodos como sumar, restar, multiplicar, dividir, etc., que calculan el resultado de realizar estas operaciones sobre dos números. Un objeto de esta clase puede usarse para calcular el producto de dos números, pero primero es necesario definir dicha clase y crear un objeto (Wikipedia, 2023b).

Declaración de la clase **aritmetica**:

```
public class Aritmetica
{
    public int suma(int numero1, int numero2)
    {
        return numero1+numero2;
    }

    public int resta(int numero1, int numero2)
    {
        return numero1-numero2;
    }

    public int multiplicacion(int numero1, int numero2)
    {
        return numero1*numero2;
    }

    public double division(int numero1, int numero2)
    {
        return numero1/numero2;
    }
}
```

Creación del objeto **operaciones**:


```
public static void main(String[] args)
{
    Aritmetica Operaciones = new Aritmetica();
}
```

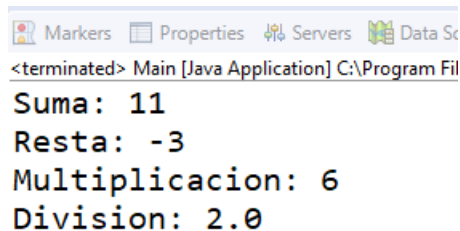
Ejecución de los métodos de la clase **aritmetica**, a partir del objeto **operaciones**:

```
public static void main(String[] args)
{
    Aritmetica Operaciones = new Aritmetica();

    int suma = Operaciones.suma(9, 2);
    int resta = Operaciones.resta(9, 12);
    int multiplicacion = Operaciones.multiplicacion(3, 2);
    double division = Operaciones.division(10, 5);

    System.out.println("Suma: " + suma);
    System.out.println("Resta: " + resta);
    System.out.println("Multiplicacion: " + multiplicacion);
    System.out.println("Division: " + division);
}
```

Resultados de la ejecución de los métodos con base en los argumentos:



```
<terminated> Main [Java Application] C:\Program Fil
Suma: 11
Resta: -3
Multiplicacion: 6
Division: 2.0
```

Static

Los elementos estáticos (o miembros de clase) son aquellos que pertenecen a la clase, en lugar de pertenecer a un objeto en particular.

Veamos un ejemplo para entender un poco su definición: el número **Pi**. Este permite realizar cálculos con circunferencias. Podría tener la clase circunferencia y definir como atributo el número Pi. Sin embargo, ese número deberá ser usado para otras operaciones, como pasar ángulos de grados a radianes. En ese caso, en condiciones normales sin atributos de clase, necesitaría instanciar cualquier círculo para luego preguntarle por el valor de Pi. Lo cual es

una mala práctica, ya que bien se podría tener *pi* como un atributo *static* y sin necesidad de crear un objeto, poder usarlo. Veamos:

A través de la clase, por medio de la palabra reservada *static*, accedemos directamente en este caso a los atributos estáticos de la misma sin necesidad de crear una instancia.

```
public class Circunferencia
{
    private static float pi = 3.1415926535f;
    private float radio = 0f;

    public Circunferencia(float radio)
    {
        this.radio = radio;
    }

    public float area()
    {
        return pi * (radio * radio);
    }
}

public static void main(String[] args)
{
    System.out.println(Circunferencia.pi);
}
```



Markers Properties Servers
<terminated> Main [Java Application] C
3.1415927

Ilustración 24: uso de *static*.

Fuente: Eclipse.

Entre los aspectos para considerar en el uso de la palabra reservada *static* encontramos los siguientes:

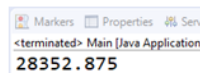
- Las instancias también pueden hacer uso de los elementos *static*.
- Se recomienda en los atributos el uso de modificadores de acceso cuando contienen características estáticas.
- Los atributos estáticos pueden ser constantes.
- En los métodos estáticos solo tienen acceso los atributos estáticos.
- Se deben usar parámetros para poder operar.

En el uso de *static* dentro de un método, debemos tener presente que la sintaxis cambia considerablemente, dado que los atributos pierden funcionalidad dentro de estos métodos. Veamos.

Método estático: se hace uso de los parámetros y únicamente de los atributos estáticos.

```
public static float area(float radio)
{
    return pi * (radio * radio);
}

public class Main
{
    public static void main(String[] args)
    {
        System.out.println(Circunferencia.area(95f));
    }
}
```



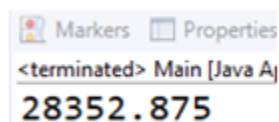
Markers Properties Run
<terminated> Main [Java Application]
28352.875

Método no estático: se hace uso de los atributos.

```
public float area()
{
    return pi * (radio * radio);
}

public class Main
{
    public static void main(String[] args)
    {
        Circunferencia Circulo = new Circunferencia(95f);

        System.out.println(Circulo.area());
    }
}
```



Markers Properties Run
<terminated> Main [Java Application]
28352.875

Herencia

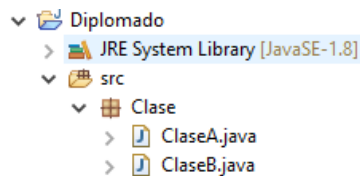
La herencia es un mecanismo que facilita la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases y objetos.

«Esto proporciona una de las ventajas principales de la programación orientada a objetos: la reutilización de código previamente desarrollado, ya que permite a una clase más específica incorporar la estructura y comportamiento de una clase más general». (Garro, 2014b)

Cuando una clase B se construye a partir de otra A mediante la herencia, la clase B hereda todos los atributos, métodos y clases internas de la clase A. Además, la clase B puede redefinir los componentes heredados y añadir atributos, métodos y clases internas específicas.

Para indicar que la clase B (clase descendiente, derivada, hija o subclase) hereda de la clase A (clase ascendiente, heredada, padre, base o superclase), se emplea la palabra reservada *extends* en la cabecera de la declaración de la clase descendiente (Garro, 2014b). La sintaxis es la siguiente:

```
package Clase;  
  
public class ClaseB extends ClaseA  
{  
  
}
```



Un ejemplo más claro de la aplicación de la herencia es el siguiente:

Un programa en el que se encuentre una clase **taxi** y **autobús** con características similares como: encendido, apagado, matrícula, modelo y potencia. Podrían resumirse esos elementos en común en una clase padre (**vehículo**) y las clases hijos (**taxi** y **autobús**) heredadas de la clase padre.

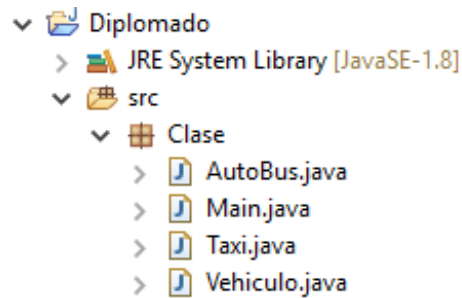


Ilustración 25: distribución de clases.

Fuente: Eclipse.

```
public class Vehiculo
{
    private String matricula;
    private String modelo;
    private int potencia;

    public String getMatricula() {
    }
    public void setMatricula(String matricula) {
    }
    public String getModelo() {
    }
    public void setModelo(String modelo) {
    }
    public int getPotencia() {
    }
    public void setPotencia(int potencia) {
    }

    public void encenderVehiculo()
    {
        System.out.println("El vehiculo está encendido");
    }

    public void apagarVehiculo()
    {
        System.out.println("El vehiculo está apagado");
    }
}
```

Ilustración 26: clase vehículo – getters - setters - métodos.

Fuente: Eclipse.

La clase padre se encarga de englobar todas las características que ambas clases hijas tengan en común para su posterior herencia. Las clases hijas tendrían la siguiente estructura:

Estructura de la clase autobús:

```
public class AutoBus extends Vehiculo
{
    private int puestos;

    public int getPuestos() {
        return puestos;
    }

    public void setPuestos(int puestos) {
        this.puestos = puestos;
    }
}
```

Estructura de la clase taxi:

```
public class Taxi extends Vehiculo
{
    private String licencia;

    public String getLicencia() {
        return licencia;
    }

    public void setLicencia(String licencia) {
        this.licencia = licencia;
    }
}
```

Con la estructura jerárquica de las tres clases descritas, ahora se puede operar entre ellas de la manera correcta y reutilizando código.

```
public class Main {

    public static void main(String[] args)
    {
        //Objeto de la Clase Taxi
        Taxi taxiAmarillo = new Taxi();

        //Uso de los atributos de la Clase Vehiculo
        taxiAmarillo.setMatricula("USR 192");
        taxiAmarillo.setModelo("1995");
        taxiAmarillo.setPotencia(800);

        //Uso del atributo de la Clase Taxi
        taxiAmarillo.setLicencia("12987894");

        //Uso de los métodos de la Clase Vehiculo
        taxiAmarillo.encenderVehiculo();
        taxiAmarillo.apagarVehiculo();
    }
}
```

```
public class Main {  
  
    public static void main(String[] args)  
    {  
        //Objeto de la Clase AutoBus  
        AutoBus autoCol = new AutoBus();  
  
        //Uso de los atributos de la Clase Vehiculo  
        autoCol.setMatricula("AHC 359");  
        autoCol.setModelo("2004");  
        autoCol.setPotencia(2000);  
  
        //Uso del atributo de la Clase AutoBus  
        autoCol.setPuestos(34);  
  
        //Uso de los métodos de la Clase Vehiculo  
        autoCol.encenderVehiculo();  
        autoCol.apagarVehiculo();  
    }  
}
```

Ilustración 27: uso de herencia.

Fuente: Eclipse.

Por medio de la clase padre, no hay necesidad de crear métodos y atributos independientes en cada clase, dado que la herencia permite englobar aquellas características que se comparten en cuanto a la estructura y simplificarlas para ahorrar código posteriormente.

«Java permite múltiples niveles de herencia, pero no la herencia *múltiple*, es decir, una clase solo puede heredar directamente de una clase ascendiente. Por otro lado, una clase puede ser ascendiente de tantas clases descendiente como se desee (*un único padre, multitud de hijos*)» (Garro, 2014b).

En la siguiente figura se muestra un ejemplo de jerarquía entre diferentes clases relacionadas mediante la herencia.

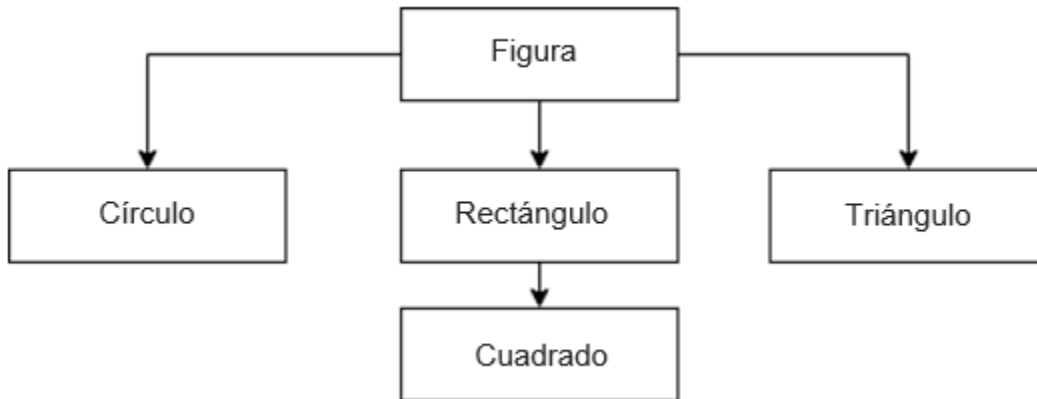


Ilustración 28: jerarquía entre diferentes clases relacionadas por herencia.

Fuente: Eclipse.

Como se ha comentado anteriormente, la clase descendiente puede añadir sus propios atributos y métodos, y asimismo puede sustituir u ocultar los heredados. En suma:

- Se puede declarar un nuevo **atributo** con el mismo identificador que uno heredado, quedando este atributo **oculto**. Esta técnica no es recomendable.
- Se puede declarar un nuevo **método de instancia** con la misma cabecera que el de la clase ascendiente, lo que supone su **sobreescritura**. Por lo tanto, la sobreescritura o redefinición consiste en que métodos adicionales declarados en la clase descendiente con el mismo nombre, tipo de dato devuelto y número y tipo de parámetros sustituyen a los heredados.
- Se puede declarar un nuevo **método de clase** con la misma cabecera que el de la clase ascendiente, lo que hace que este quede **oculto**. Por lo tanto, los métodos de clase o estáticos (declarados como **static**) no pueden ser redefinidos.
- Un método declarado con el modificador **final** tampoco puede ser redefinido por una clase derivada.
- Se puede declarar un **constructor** de la subclase que llame al de la superclase de forma implícita o de mediante la palabra reservada **super**.

En general, puede accederse a los métodos de la clase ascendiente que han sido redefinidos empleando la palabra reservada **super** delante del identificador del

método. Este mecanismo solo permite acceder al método perteneciente a la clase en el nivel inmediatamente superior de la jerarquía de clases (Garro, 2014b).

Sobreescritura de métodos - polimorfismo

«Es la forma por la cual una clase que hereda puede redefinir los métodos de su clase padre, de esta manera puede crear nuevos métodos con el mismo nombre de su superclase». (Henao, 2014).

Es decir, si existe una clase padre con el método saludar (), se puede crear en la clase hija un método que también se llame saludar (), pero con la implementación según la necesidad.

Estas son algunas características para tener muy presentes al trabajar con sobreescritura de métodos:

- La estructura de los métodos debe ser igual en ambas clases: mismos parámetros, mismo tipo de retorno e implementación del mismo modificador de acceso.
- Los métodos *static* y *final* no se pueden sobreescribir.
- Se recomienda el uso de anotaciones cuando se trabaja con sobreescritura de métodos, en especial `@Override` (en clases abstractas o interfaces se usará más claramente).

Observemos el siguiente ejemplo para entender el funcionamiento de la sobreescritura de métodos:

Clase padre instrumento:

```
public class Instrumento
{
    public void tocar()
    {
        System.out.println("Tocar el instrumento");
    }
}
```

Clase hija clarinete, heredada de la clase padre instrumento:

```
public class Clarinete extends Instrumento
{
    @Override
    public void tocar()
    {
        System.out.println("Tocando el Clarinete");
    }
}
```

La clase padre implementa el método tocar; la clase hija lo sobrescribe, por lo que ahora son dos métodos diferentes (recordemos el uso de la anotación). La estructura de ejecución sería la siguiente:

```
public class Main {
    public static void main(String[] args)
    {
        Clarinete clarineteNegro = new Clarinete();

        clarineteNegro.tocar();
    }
}
```

Ilustración 29: uso del polimorfismo.
Fuente: Eclipse.

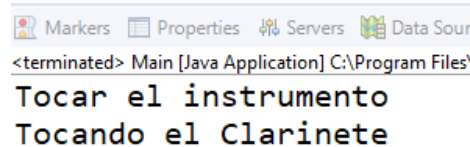
Partiendo del objeto de tipo clarinete, se accede al método tocar de la misma clase (no la padre, dado que ha sido sobrescrita). Observemos la salida:



Markers Properties Servers Data
<terminated> Main [Java Application] C:\Program
Tocando el Clarinete

Ya el objeto no accede al método de la **superclase** o clase **padre**, sino que directamente accede al declarado en su propia estructura de clase. Si queremos obtener la ejecución de ambos métodos, tanto el de la clase **padre** como el de **hija**, simplemente se tendría que cambiar la siguiente estructura:

```
public class Clarinete extends Instrumento
{
    @Override
    public void tocar()
    {
        super.tocar();
        System.out.println("Tocando el Clarinete");
    }
}
```



Markers Properties Servers Data Sour
<terminated> Main [Java Application] C:\Program Files\
Tocar el instrumento
Tocando el Clarinete

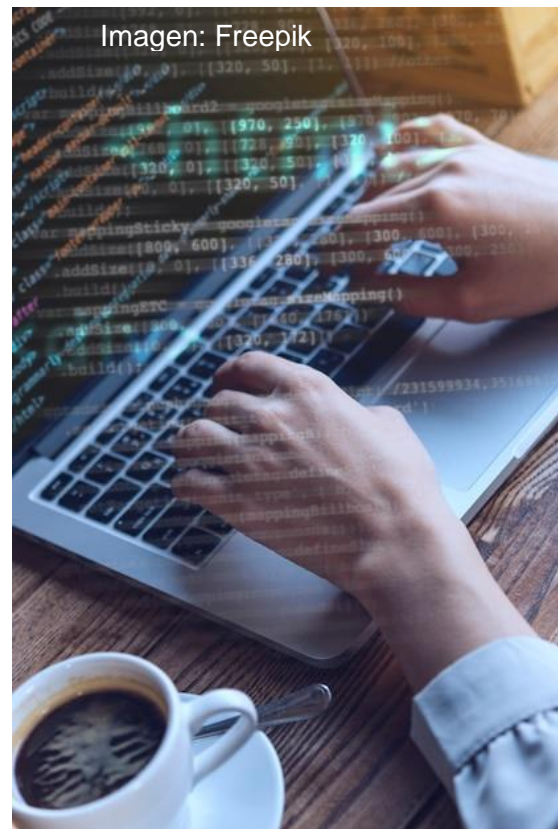
La palabra **super** hace el llamado al método **tocar** de la superclase, hace las veces de **this**, pero ya no hace referencia al objeto o clase actual, sino a su clase padre, en este caso, instrumento.

Clases anidadas

Es una clase que se declara dentro de otra clase. Están definidas de forma normal, pero dentro de otra clase, o definidas dentro de un método, o dentro de una clase asignadas a un atributo, o pasadas como parámetro, o asignadas a una variable local.

Una clase anidada no existe independientemente de su clase adjunta. Por lo tanto, el alcance de una clase anidada está limitado por su clase externa.

- Una clase anidada también es miembro de su clase adjunta. También es posible declarar una clase anidada que es local a un bloque.



- Como miembro de su clase adjunta, una clase anidada se puede declarar *private*, *public*, *protected*, o *default*.
- Una clase anidada tiene acceso a los miembros, incluidos los miembros privados, de la clase en la que está anidado. Sin embargo, lo inverso no es verdadero, es decir, la clase adjunta no tiene acceso a los miembros de la clase anidada. (Web 2020, s.f.)

Definición de la clase usuario – definición de la clase anidada administrador:

```
public class Usuario
{
    public String usuario;

    public Usuario(String usuario)
    {
        this.usuario = usuario;
    }

    public void establecerRoles()
    {
        Administrador admin = new Administrador();
        admin.trabajar();
    }

    public class Administrador
    {
        public void trabajar()
        {
            System.out.println("El administrador se encuentra trabajando");
        }
    }
}
```

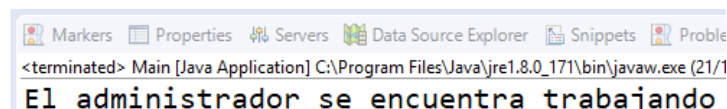
Ilustración 30: clases anidadas.

Fuente: Eclipse.

Ejecución de la clase anidada administrador a partir del objeto de tipo usuario:

```
public class Main {

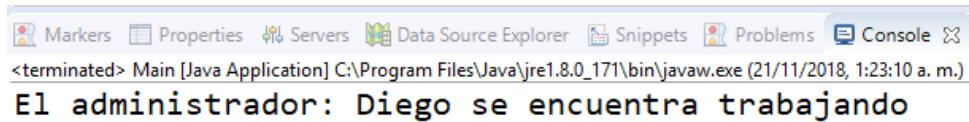
    public static void main(String[] args)
    {
        Usuario usuario = new Usuario("Diego");
        usuario.establecerRoles();
    }
}
```



Markers Properties Servers Data Source Explorer Snippets Problems
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (21/1)
El administrador se encuentra trabajando

También podemos usar atributos y métodos de la clase en que se anida:

```
public class Administrador
{
    public void trabajar()
    {
        System.out.println("El administrador: " + usuario + " se encuentra trabajando");
    }
}
```



Markers Properties Servers Data Source Explorer Snippets Problems Console
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (21/11/2018, 1:23:10 a. m.)
El administrador: Diego se encuentra trabajando

Clases abstractas

En Java las clases abstractas son aquellas clases base (superclases) de las cuales no se permite crear objetos. Para ello, se utiliza la palabra clave *abstract*.

En una clase abstracta pueden definirse métodos abstractos, que se caracterizan por el hecho de que no pueden ser implementados en la clase base. De ellos solo se escribe su signature en la superclase, y su funcionalidad (polimórfica) tiene que indicarse en las clases derivadas (subclases).

```
package Clase;

public abstract class Figura
{
    private String color;

    public Figura(String color)
    {
        this.color = color;
    }

    public abstract double calcularArea();

    public String getColor()
    {
        return color;
    }
}
```

Ilustración 31: clase abstracta.

Fuente: Eclipse.

En la clase **figura** se ha definido un atributo (**color**), un constructor y dos métodos (**calcularArea** y **getColor**), **calcularArea** es abstracto, por lo que no cuenta con implementación.

```
package Clase;

public class Cuadrado extends Figura
{
    private double lado;

    public Cuadrado(String color, double lado)
    {
        super(color);
        this.lado = lado;
    }

    public double calcularArea()
    {
        return lado * lado;
    }
}
```

Ilustración 32: uso de clase abstracta.

Fuente: Eclipse.

En la clase **cuadrado** se ha definido un atributo (**lado**), un constructor y un método (**calcularArea**).

```
package Clase;

public class Triangulo extends Figura
{
    private double base;
    private double altura;

    public Triangulo(String color, double base, double altura)
    {
        super(color);
        this.base = base;
        this.altura = altura;
    }

    public double calcularArea()
    {
        return (base * altura) / 2;
    }
}
```

En la clase **triángulo** se han definido dos atributos (**base** y **altura**), un constructor y un método (**calcularArea**).

Como observamos, el método **calcularArea** ha sido definido abstracto (*abstract*) en la superclase abstracta figura, indicándose solamente su signatura:

```
public abstract double calcularArea();
```

Por otro lado, veamos que en cada una de las subclases (cuadrado y triángulo) se ha implementado dicho método:

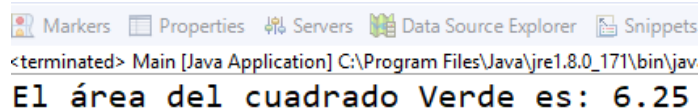
```
package Clase;

public class Main {

    public static void main(String[] args)
    {
        String colorDelCuadrado = "Verde";
        double ladoDelCuadrado = 2.5;

        Cuadrado cuadrado = new Cuadrado(colorDelCuadrado, ladoDelCuadrado);

        System.out.printf("El área del cuadrado " + cuadrado.getColor() + " es: " + cuadrado.calcularArea());
    }
}
```

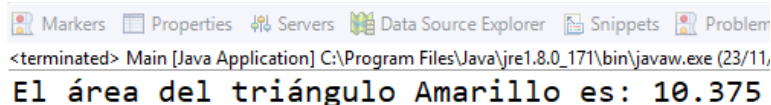


```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\jav
El área del cuadrado Verde es: 6.25
```

```
public static void main(String[] args)
{
    String colorDelTriangulo = "Amarillo";
    double baseDelTriangulo = 8.3;
    double alturaDelTriangulo = 2.5;

    Triangulo triangulo1 = new Triangulo(colorDelTriangulo, baseDelTriangulo, alturaDelTriangulo);

    System.out.printf("El área del triángulo " + triangulo1.getColor() + " es: " + triangulo1.calcularArea());
}
```



```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin\javaw.exe (23/11,
El área del triángulo Amarillo es: 10.375
```

Fijémonos en que, en la ejecución de ambas clases, se ha invocado al método getColor definido e implementado en la superclase **figura**. Sin embargo, el método abstracto **calcularArea** únicamente se implementa en las subclases (**cuadrado** y **triángulo**).

Interfaces

Una interfaz en Java es una colección de métodos abstractos y propiedades constantes. En las interfaces se especifica qué se debe hacer, pero no su implementación. Serán las clases que implementen estas interfaces y describan la lógica del comportamiento de los métodos.

La principal diferencia entre *interface* y *abstract* es que una interfaz proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia (Wikipedia, 2023c).

Una interfaz es una especie de plantilla para la construcción de clases. Normalmente una interfaz se compone de un conjunto de declaraciones de cabeceras de métodos (sin implementar, de forma similar a un método abstracto) que especifican un protocolo de comportamiento para una o varias clases. Además, una clase puede implementar una o varias interfaces: en ese caso, la clase debe proporcionar la declaración y definición de todos los métodos de cada una de las interfaces, o bien declararse como clase *abstract*. Por otro lado, una interfaz puede emplearse también para declarar constantes que luego puedan ser utilizadas por otras clases (Garro, 2014c)

Una interfaz puede parecer similar a una clase abstracta, pero existen diferencias entre una interfaz y una clase abstracta:

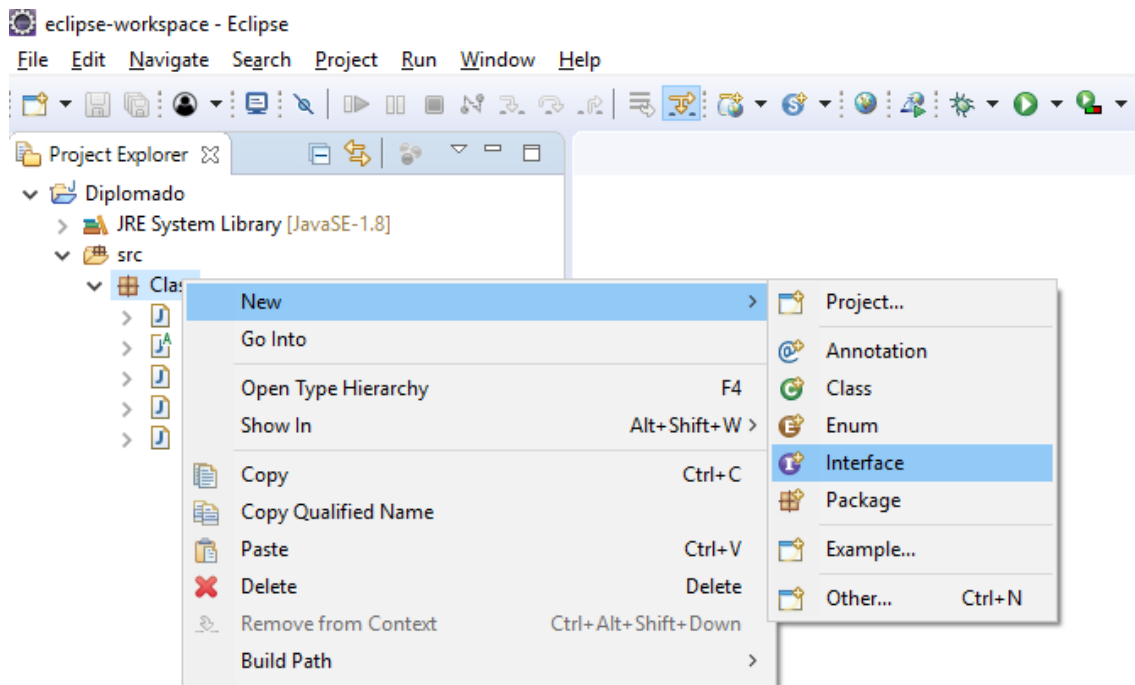
- Las interfaces especifican lo que debe hacer una clase y no cómo; es el plano de la clase.
- Si una clase implementa una interfaz y no proporciona cuerpos de métodos para todas las funciones especificadas en la interfaz, la clase debe declararse abstracta.
- Permiten declarar constantes que van a estar disponibles para todas las clases (implementando esa interfaz)
- Obligar a que ciertas clases utilicen los mismos métodos (nombres y parámetros).
- Establecer relaciones entre clases que no estén relacionadas.
- Todos los métodos de una interfaz se declaran implícitamente como abstractos y públicos.

- Una clase abstracta no puede implementar los métodos declarados como abstractos, una interfaz no puede implementar ningún método (ya que todos son abstractos).
- Una interfaz no declara variables de instancia.
- Una clase puede implementar varias interfaces, pero solo puede tener una clase ascendiente directa. (Garro, 2014c)

Creación de una interfaz

La declaración de una interfaz es similar a una clase, aunque emplea la palabra reservada **interface** en lugar de **class** y no incluye ni la declaración de variables de instancia ni la implementación del cuerpo de los métodos (solo las cabeceras). (Garro, 2014c)

La sintaxis de declaración de una interfaz es la siguiente:



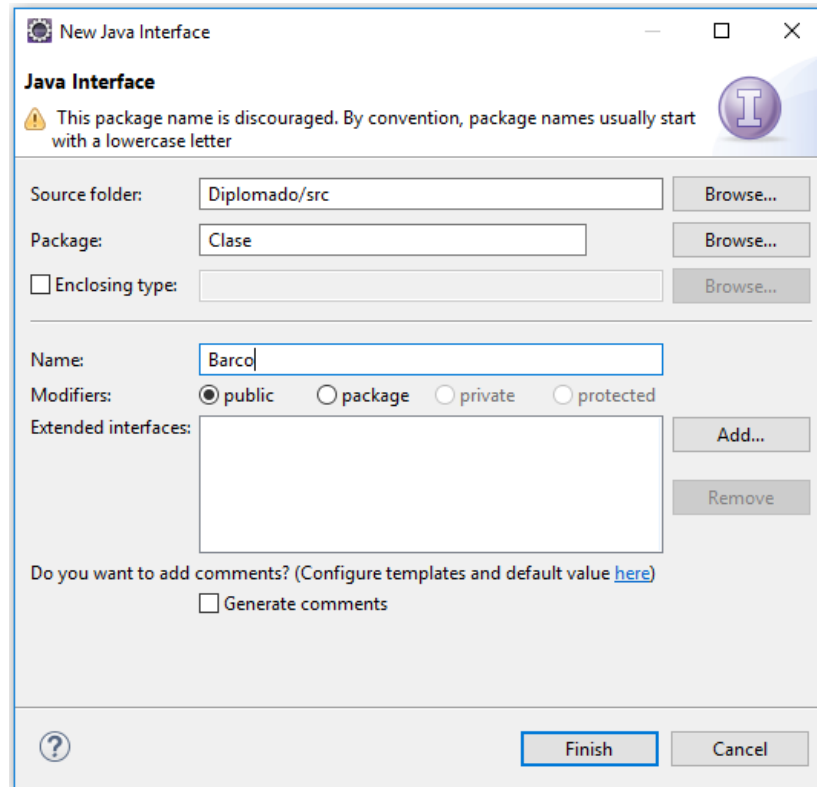


Ilustración 33: creación de interfaces.

Fuente: Eclipse.

```
package Clase;  
  
public interface Barco  
{  
  
}
```

Ilustración 34: resultado de creación de interfaces.

Fuente: Eclipse.

Las cabeceras de los métodos declarados en el cuerpo de la interfaz se separan entre sí por caracteres de punto y coma, y todos son declarados implícitamente como *public* y *abstract* (se pueden omitir). Por su parte, todas las constantes incluidas en una interfaz se declaran implícitamente como *public*, *static*

y final (también se pueden omitir) y es necesario inicializarlas en la misma sentencia de declaración.

```
package Clase;

public interface Barco
{
    void moverPosicion(int x, int y);
    void disparar();
}
```

En la interface **barco** se han definido dos métodos (**moverPosicion** y **Disparar**).

```
public class BarcoPirata implements Barco {

    @Override
    public void moverPosicion(int x, int y) {

    }

    @Override
    public void disparar() {

    }

}
```

En la clase **BarcoPirata** se implementa la interfaz por medio de la palabra reservada **implements** y el nombre de la clase. Esta automáticamente exige e implementa los métodos declarados en la interfaz para su posterior definición.

```
public class BarcoPirata implements Barco {

    private int x;
    private int y;

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void conocerPosicion(){
        System.out.println("La posición actual es: " + x + " - " + y);
    }

    @Override
    public void moverPosicion(int x, int y) {
        this.x = this.x - x;
        this.y = this.y - y;
    }

    @Override
    public void disparar() {
        System.out.println("Disparar cañones");
    }
}
```

Clase **BarcoPirata**, con algunas modificaciones como **atributos** para realizar cambios en la posición, un nuevo método llamado **conocerPosicion**, **getters** y **setters** y la implementación de los métodos de la clase **barco** realizada.

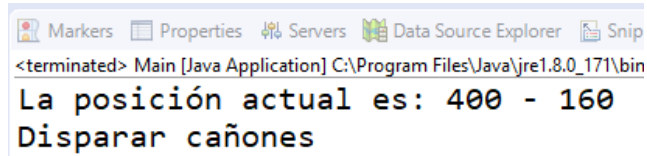
```
public static void main(String[] args)
{
    BarcoPirata sunny = new BarcoPirata();

    sunny.setX(450);
    sunny.setY(180);

    sunny.moverPosicion(50, 20);

    sunny.conocerPosicion();

    sunny.disparar();
}
```




```
Markers Properties Servers Data Source Explorer Snip
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_171\bin
La posición actual es: 400 - 160
Disparar cañones
```

Observemos que, en la ejecución de la clase **BarcoPirata**, se han invocado los métodos definidos por la interfaz **barco** y se ha hecho uso de estos sin problema, al igual que los nuevos atributos y métodos de la clase que los implementa.



Referencias bibliográficas

- Aprendiendo Arduino. (2019, 1 de septiembre). *Qué es la programación*. Aprendiendo Arduino. <https://tinyurl.com/p9hcpp>
- Byspel Tech. (s.f.). Qué son las clases en Java y para qué sirven, modelado de objetos en Java. *Byspel Tech*. <https://tinyurl.com/ympe8c3>
- Desarrollo Web. (2021, 18 de septiembre). *Qué es la programación orientada a objetos*. Desarrollo Web. <https://desarrolloweb.com/articulos/499.php>
- Eclipse Foundation. (2023). *Eclipse*. Eclipse Foundation. <https://www.eclipse.org/>
- Garro, A. (2014a, 15 de abril). Capítulo 14. Tipos de métodos. En *Java*. Arkaitz Garro. <https://www.arkaitzgarro.com/java/capitulo-14.html>
- Garro, A. (2014b, 15 de abril). Capítulo 16. Herencia. En *Java*. Arkaitz Garro. <https://www.arkaitzgarro.com/java/capitulo-16.html>
- Garro, A. (2014c, 15 de abril). Capítulo 18. Interfaces. En *Java*. Arkaitz Garro. <https://www.arkaitzgarro.com/java/capitulo-18.html>
- Henao, C. (2014, 12 de mayo). Sobreescritura y sobrecarga de métodos en Java (overriding y overloading). *Codejavu*. <https://tinyurl.com/525dc8sx>
- Meza, J. (2018a). *Métodos en Java, funciones y procedimientos. Cómo hacerlos y usarlos*. Programar Ya. <https://www.programarya.com/Cursos/Java/Funciones>
- Meza, J. (2018b). *Modificadores de acceso public, protected, default y private en Java. Encapsulamiento en Java*. Programar Ya. <https://www.programarya.com/Cursos/Java/Modificadores-de-Acceso>
- Poolhjc. (2008, 21 de julio). Programación orientada a objetos. *WordPress*. <https://poolhjc.wordpress.com/clases/>
- Web 2020. (s.f.). Clases anidadas. En *Java*. Web 2020. <http://web2020.es/java-clasesanidadas.html>
- Wikipedia. (2023a, 20 de enero). Programación estructurada. En *Wikipedia*. https://es.wikipedia.org/wiki/Programaci%C3%B3n_estructurada
- Wikipedia. (2023b, 16 de mayo). Objeto (programación). [https://es.wikipedia.org/wiki/Objeto_\(programaci%C3%B3n\)](https://es.wikipedia.org/wiki/Objeto_(programaci%C3%B3n))
- Wikipedia. (2023c, 24 de julio). Interfaz (Java). En *Wikipedia*. [https://es.wikipedia.org/wiki/Interfaz_\(Java\)](https://es.wikipedia.org/wiki/Interfaz_(Java))



Esta guía fue elaborada para ser utilizada con fines didácticos como material de consulta de los participantes en el Diplomado Virtual en PROGRAMACIÓN EN JAVA del Politécnico de Colombia, y solo podrá ser reproducida con esos fines. Por lo tanto, se agradece a los usuarios referirla en los escritos donde se utilice la información que aquí se presenta.

GUÍA DIDÁCTICA 3

M2-DV59-GU03

MÓDULO 3: PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

© DERECHOS RESERVADOS - POLITÉCNICO DE COLOMBIA, 2023
Medellín, Colombia

Proceso: Gestión Académica Virtual
Realización del texto: Diego Palacio, docente
Revisión del texto: Comité de Revisión
Diseño: Comunicaciones

Editado por el Politécnico de Colombia