

Introducción a la P00



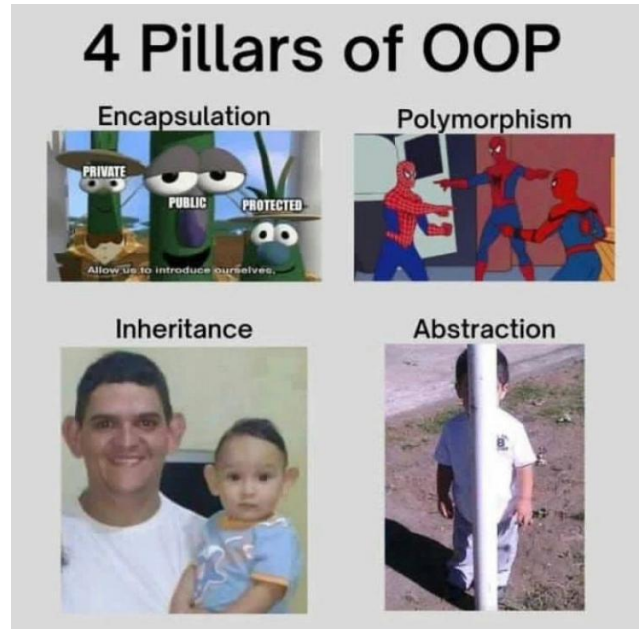
Curso 2023/24

Java

Los cuatro pilares de la POO

Los cuatro pilares de la POO

Estos son los cuatros pilares de la POO:

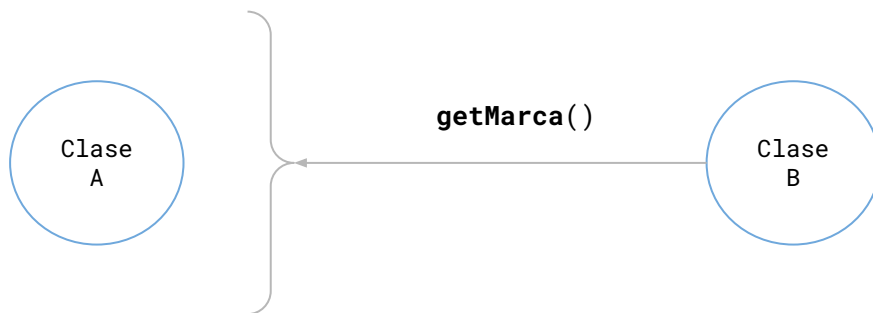


Abstracción

Se refiere a la capacidad de representar objetos del mundo real en un programa de manera simplificada, enfocándose solo en los detalles esenciales y **ocultando los detalles innecesarios**.

Sólo es necesario conocer las **cabeceras** de los métodos para saber cómo comunicarse con el objeto.

```
private String marca;  
private String modelo;  
  
public getMarca() {  
    ...  
}
```



Abstracción

Recuerda, en la POO la abstracción de datos es **muy importante y nunca** debe romperse.

No prestar atención a la abstracción de la información implicará, potencialmente, accesos **no deseados** y **brechas** en nuestras aplicaciones (además de demostrar un nivel de competencia nulo).

La implementación del tipo de datos es **privada**. Por esa razón, los **métodos** y **procedimientos** para manipular el tipo de datos son **públicos**.

Tarea: observa y analiza el código proporcionado por el profesor, y separa las clases en ficheros. Si no lo entiendes, trata de dibujar su diagrama de clases para saber cómo funciona. Además, fíjate en la abstracción del código.

Encapsulación

Estrechamente relacionado con la abstracción, la encapsulación es uno de los conceptos fundamentales de la programación orientada a objetos que se refiere a la **ocultación de los detalles internos de un objeto** y la protección de sus datos y métodos.

En Java, la encapsulación se logra mediante la declaración de variables como **privadas** y proporcionando métodos **públicos** (llamados "métodos accesorios" o "getters" y "setters") para acceder y modificar esos datos de manera controlada.

Encapsulación

Tarea: observa y analiza el código proporcionado por el profesor, y separa las clases en ficheros. Si no lo entiendes, trata de dibujar su diagrama de clases para saber cómo funciona. Además, fíjate en la encapsulación del código.

En este ejemplo, la clase `Persona` encapsula los datos de una persona (nombre y edad) al declarar las variables como privadas y proporcionar métodos públicos (`getNombre()`, `setNombre()`, `getEdad()`, `setEdad()`) para acceder y modificar estos datos. El uso de **setters** también permite realizar validaciones, como asegurarse de que la edad no sea negativa.

Herencia

La herencia permite la creación de una nueva clase **basada** en una clase existente. En Java, la herencia se utiliza para modelar una relación "es-un" entre clases (como lo que vimos en el modelo EER de Bases de Datos), donde una clase derivada (subclase o clase hija) **hereda** características y comportamientos de una clase base (superclase o clase padre).

Tarea: observa y analiza el código proporcionado por el profesor, y separa las clases en ficheros. Si no lo entiendes, trata de dibujar su diagrama de clases para saber cómo funciona. Además, añade dos animales más e inventa alguno de sus atributos y métodos.

Polimorfismo

El polimorfismo se refiere a la capacidad de diferentes clases de objetos de **responder al mismo mensaje de manera distinta**. En Java, el polimorfismo se logra mediante la herencia y la implementación de **interfaces**, y **permite que un objeto se comporte de manera flexible según el contexto** en el que se utilice.

Tarea: observa y analiza el código proporcionado por el profesor, y separa las clases en ficheros. Si no lo entiendes, trata de dibujar su diagrama de clases para saber cómo funciona. Ejecuta el programa para ver su funcionamiento y trata de entender el polimorfismo.

Java

Conceptos generales

Sintaxis de una clase Java

Mantener una organización común y seguir determinados protocolos **ayuda** a los desarrolladores a elaborar proyectos bajo un mismo **marco** de trabajo.

```
Public class NombreClase {  
    // Atributos  
    // Constructoras  
    // Métodos  
}
```

Atributos

Definen las propiedades de una clase.

```
ÁMBITO TIPO nombre;
```

El ámbito determina la visibilidad del atributo.

- `public`: accesible desde cualquier sitio.
- `private`: accesible únicamente en la implementación de la clase.
- `protected`: igual que el público para clases que hereden y privado para el resto.

Atributos

Los atributos se pueden inicializar a la vez que se declaran.

```
private int x = 2;  
private String cadena = "cadena vacía";  
private boolean fin = true;
```

Como norma general, todos los atributos serán privados o protegidos cuando estemos en presencia de herencia.

Atributos de objeto

Siempre nos referimos a ellos anteponiendo la palabra `this`.

```
public class A {  
    private int x;  
    private int y;  
  
    public int suma() {  
        return this.x + this.y;  
    }  
  
    public incr(int z) {  
        this.x = this.x + z;  
        this.y = this.y + z;  
    }  
}
```

Atributos constantes

Son atributos cuyo valor no se modifica **nunca**, es decir son **constantes**.

Este tipo de atributos habitualmente son compartidos por **todos** los objetos de la clase, es decir son **estáticos**.

Es conveniente además que dichas constantes estén **accesibles** para todos los objetos de la clase, con lo cual conviene que sean **públicos**.

¿Cómo se crean los objetos?

Los objetos se crean **invocando** a sus constructoras.

Una constructora en una clase **inicializa** los valores de los **atributos**.

¿Cómo se crean los objetos?

Los objetos se crean **invocando** a sus constructoras.

Una constructora en una clase **inicializa** los valores de los **atributos**.

Puede haber **muchas** constructoras.

```
private String marca;  
private String modelo;  
  
public Coche(String _marca) {  
    ...  
}
```

```
private String marca;  
private String modelo;  
  
public Coche(String _modelo) {  
    ...  
}
```

Constructoras

Sirven para **inicializar** los atributos de un objeto.

```
public NombreClase (tipo1 arg1, ....., tipon argn) {  
    ...  
}
```

Las constructoras **tienen 3 características** que las distinguen del resto de los métodos:

- I. Se llaman igual que la clase.
- II. No tienen tipo de salida (ni siquiera void).
- III. Sólo se llaman una vez por objeto, cuando este se crea con new. No pueden llamarse explícitamente (sin new).

Constructoras

A la hora de diseñar una clase es **muy importante** definir de qué constructoras dispondrá la clase.

Aunque no incluyamos ninguna constructora, **toda** clase tiene una constructora sin argumentos por **defecto**.

Una vez incluida una constructora con argumentos, la constructora sin argumentos **deja** de existir, por lo que si queremos seguir usándola hay que **incluirla** explícitamente.

Constructoras

```
public class A {  
    private int x;  
  
    public A (int y) {  
        this.x = y;  
    }  
}
```

```
public class B {  
    private A a;  
  
    public B (A aa) {  
        this.a = aa;  
    }  
}
```

Constructoras

```
public class A {  
    private int x;  
  
    public A (int y) {  
        this.x = y;  
    }  
}
```

```
public class B {  
    private A a;  
  
    public B (A aa) {  
        this.a = aa;  
    }  
}
```

```
A aa = new A(2);  
B bb = new B(aa);
```

Constructoras

```
public class A {  
    private int x;  
  
    public A (int y) {  
        this.x = y;  
    }  
}
```

```
public class B {  
    private A a;  
  
    public B (A aa) {  
        this.a = aa;  
    }  
}
```

```
A aa = new A(2);  
B bb = new B(aa);
```



Compartición

Declaración de métodos

Los métodos **definen** la funcionalidad de un objeto, es decir las operaciones que el objeto **puede** hacer.

Los métodos pueden definirse como privados, públicos o protegidos. Si forman parte de la interfaz deben ser públicos. Si son auxiliares para definir internamente a otros métodos, deben ser privados. El concepto de protegido aparecerá con la herencia.

```
ÁMBITO TIPO nombre (lista_parámetros) {  
    // Cuerpo del método  
}
```

En Java hay **sobrecarga** de métodos: puede haber varios métodos con el mismo tipo de retorno (o void) pero que difieren en el número y/o tipo de los parámetros.

Declaración de métodos

```
public class A {  
    private int x;  
  
    A () {  
        this.x = 0;  
    }  
    int incr(int z){  
        return this.x + z;  
    }  
    int incr() {  
        return this.x + 1;  
    }  
    int incr(double z){  
        return 1;  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        A a = new A();  
        a.incr(2);  
        a.incr(2.0);  
    }  
}
```


Declaración de métodos

```
public class A {
    private int x;

    A () {
        this.x = 0;
    }
    int incr(int z){
        return this.x + z;
    }
    int incr(){
        return this.x + 1;
    }
    int incr(double z){
        return 1;
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new A();
        a.incr(2); // Ejecuta primer método
        a.incr(2.0); // Ejecuta segundo método
    }
}
```

Llamadas a métodos

`objeto.método(parámetros)`

`objeto` debe haber sido creado previamente invocando a alguna de las constructoras de la clase a la que pertenece.

Llamadas a métodos

`objeto.método(parámetros)`

objeto debe haber sido creado previamente invocando a alguna de las constructoras de la clase a la que pertenece.

```
// method definition
public static void printArea(int x, int y) {
    System.out.println(x * y);
}

public static void main(String[] args) {
    // calling method
    printArea(2, 4);
}
```

Parameters

Arguments

Método Main

```
Public static void main (String[] args) {  
    // Cuerpo del método  
}
```

Es el punto de **inicio** de toda aplicación en Java.

Toda clase puede tener un método `main`, que puede utilizarse para comprobar el funcionamiento correcto de la misma.

En una aplicación en Java siempre habrá un **único** método `main`. Como convenio general, este método lo implementaremos en una clase llamada `Main`.

Es un método estático. Se aplica por tanto a la **clase** y no a una instancia en particular.

Precaución con métodos mutadores/accesores

```
public class A {  
    private int x;  
  
    public A (int y) {  
        this.x = y;  
    }  
  
    public void set(int _num) {  
        this.x = _num;  
    }  
}
```

```
public class B {  
    private A a;  
  
    public B (A _aa){  
        this.a = _aa;  
    }  
  
    public A get(){  
        return this.a;  
    }  
}
```

Precaución con métodos mutadores/accesores

```
public class A {  
    private int x;  
  
    public A (int y) {  
        this.x = y;  
    }  
  
    public void set(int _num) {  
        this.x = _num;  
    }  
}
```

```
public class B {  
    private A a;  
  
    public B (A _aa){  
        this.a = _aa;  
    }  
  
    public A get(){  
        return this.a;  
    }  
}
```

```
A a1 = new A(3);  
B b = new B(a1);  
A a2 = b.get();  
a2.set(7);
```

Precaución con métodos mutadores/accesores

```
public class A {  
    private int x;  
  
    public A (int y) {  
        this.x = y;  
    }  
  
    public void set(int _num) {  
        this.x = _num;  
    }  
}
```

```
public class B {  
    private A a;  
  
    public B (A _aa){  
        this.a = _aa;  
    }  
  
    public A get(){  
        return this.a;  
    }  
}
```

```
A a1 = new A(3);  
B b = new B(a1);  
A a2 = b.get();  
a2.set(7);
```

→ ¿Qué salida produce este código?
¿Cuál es el problema si a continuación
ejecutamos b.get() ;?

Visibilidad

public: los métodos o atributos públicos son accesibles desde la implementación de **cualquier** clase.



private: los atributos o métodos privados sólo son accesibles desde la implementación de la **propia** clase.

Pueden omitirse los **public/private**. En tal caso:

- El elemento es accesible dentro de la propia clase.
- El elemento es accesible dentro del mismo paquete.
- El elemento no es accesible para el resto (clases de otros paquetes).

Igualdad entre objetos

Dos objetos se pueden comparar con "==" o el método `equals` de la clase `Object`. La comparación "==" compara las direcciones de memoria de **ambos** objetos, es decir, si ambos objetos apuntan a la **misma** dirección.

`equals` existe por defecto y se comporta como "==" . Pero toda clase puede **redefinirlo** para establecer la igualdad deseada.

Si no se implementa el método `equals` en una clase entonces es equivalente a "==" .

El método `equals` pertenece a la clase `Object`.

Todas las clases heredan de `Object`.

Igualdad entre objetos, ejemplo 1

Definamos la clase A con un par de atributos e implementemos el método `equals`.

```
public class A {  
    private int x;  
    private int y;  
  
    public A (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public boolean equals (A a) {  
        return (  
            this.x == a.x &&  
            this.y == a.y  
        );  
    }  
}
```

Igualdad entre objetos, ejemplo 1

Definamos la clase A con un par de atributos e implementemos el método equals.

```
public class Main {  
  
    public static void main (String[] args) {  
        A a1 = new A(2,2);  
        A a2 = new A(2,2);  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

```
public class Main {  
  
    public static void main (String[] args) {  
        A a1 = new A(2,2);  
        A a2 = a1;  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

Igualdad entre objetos, ejemplo 1

Definamos la clase A con un par de atributos e implementemos el método equals.

```
public class Main {  
  
    public static void main (String[] args) {  
        A a1 = new A(2,2);  
        A a2 = new A(2,2);  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

```
public class Main {  
  
    public static void main (String[] args) {  
        A a1 = new A(2,2);  
        A a2 = a1;  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

¿Qué muestra por pantalla?

Igualdad entre objetos, ejemplo 1

Definamos la clase A con un par de atributos e implementemos el método equals.

```
public class Main {  
  
    public static void main (String[] args) {  
        A a1 = new A(2,2);  
        A a2 = new A(2,2);  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

¿Qué muestra por pantalla?

```
public class Main {  
  
    public static void main (String[] args) {  
        A a1 = new A(2,2);  
        A a2 = a1;  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

¿Y ahora? ¿Por qué?

Igualdad entre objetos, ejemplo 2

Definamos las siguientes clases B y A1 con sus respectivos atributos.

```
public class B {  
    private int z;  
  
    public B (int z) {  
        this.z = z;  
    }  
}
```

```
public class A1 {  
    private int x;  
    private B b;  
  
    public A1 (int x, B b) {  
        this.x = x;  
        this.b = b;  
    }  
  
    public boolean equals (A1 a) {  
        return (  
            this.x == a.x &&  
            this.b == a.b  
        );  
    }  
}
```

Igualdad entre objetos, ejemplo 2

```
public class Main {  
    public static void main(String[] args) {  
        B b1 = new B(2);  
        B b2 = new B(2);  
        A1 a1 = new A1(3,b1);  
        A1 a2 = new A1(3,b2);  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

Igualdad entre objetos, ejemplo 2

```
public class Main {  
    public static void main(String[] args) {  
        B b1 = new B(2);  
        B b2 = new B(2);  
        A1 a1 = new A1(3,b1);  
        A1 a2 = new A1(3,b2);  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

¿Qué muestra por pantalla?

Igualdad entre objetos

Cuando tenemos atributos que son objetos (punteros), si se desea definir el método `equals` hay definirlo de forma sintáctica en todas las clases.

```
// En la Clase A1 añade:
```

```
public boolean equals (A1 a) {  
    return (  
        this.x == a.x &&  
        this.b.equals(a.b)  
    );  
}
```

```
// En la Clase B añade:
```

```
public boolean equals (B b) {  
    return (  
        this.z == b.z  
    );  
}
```

Igualdad entre objetos

```
public class Main {  
    public static void main(String[] args) {  
        B b1 = new B(2);  
        B b2 = new B(2);  
        A1 a1 = new A1(3,b1);  
        A1 a2 = new A1(3,b2);  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

Igualdad entre objetos

```
public class Main {  
    public static void main(String[] args) {  
        B b1 = new B(2);  
        B b2 = new B(2);  
        A1 a1 = new A1(3,b1);  
        A1 a2 = new A1(3,b2);  
  
        if (a1.equals(a2))  
            System.out.println("Cierto");  
        else  
            System.out.println("Falso");  
    }  
}
```

Y ahora, ¿qué muestra por pantalla?

Clonación de objetos

Para clonar un objeto existe el método `clone` de la clase `Object`.

El método `clone` debe implementarse en **todas** las clases para que realice una copia del objeto.

La copia debe realizarse a **todos** los niveles, es decir, si hay atributos que son objetos, estos a su vez **deben** clonarse.

Clonación de objetos

```
public class A1 {  
    private int x;  
    private B b;  
  
    public A1 (int x, B b) {  
        this.x = x;  
        this.b = b;  
    }  
  
    public boolean equals (A1 a) {  
        return (  
            this.x == a.x &&  
            this.b.equals(a.b)  
        );  
    }  
  
    public A1 clone () {  
        return new A1(  
            this.x,  
            this.b.clone()  
        );  
    }  
}
```

```
public class B {  
    private int z;  
  
    public B (int z) {  
        this.z = z;  
    }  
  
    public boolean equals (B b) {  
        return (this.z == b.z);  
    }  
  
    public B clone () {  
        return new B(this.z);  
    }  
}
```

Atributos y métodos estáticos

Los atributos y métodos vistos hasta ahora son atributos y métodos de **objeto** (o instancia).

Existen, además, otro tipo de atributos y métodos denominados **estáticos** o de **clase**.

Los atributos y métodos de un objeto son **propios** del objeto.

Los atributos y métodos de clase son **compartidos** por todos los objetos de la clase.

Atributos y métodos estáticos

Un método o atributo estático **no necesita** para su ejecución un objeto al que acceder. Se invocan con el nombre de la clase aunque también se pueden invocar **a través** de objetos.

Un método estático **no puede** modificar ningún atributo no estático del objeto `this`, ya que son métodos que no pertenecen a **ningún** objeto sino a la clase.

Atributos y métodos estáticos

```
public class A {  
    private static int x = 5;  
    private int y;  
  
    public A (int y) {  
        this.y = y;  
    }  
  
    static public void incrementa() {  
        A.x++;  
    }  
  
    public String toString() {  
        return "x = " + A.x +  
            "y = " + this.y;  
    }  
}
```


Atributos y métodos estáticos

```
public class A {  
    private static int x = 5;  
    private int y;  
  
    public A (int y) {  
        this.y = y;  
    }  
  
    static public void incrementa() {  
        A.x++;  
    }  
  
    public String toString() {  
        return "x = " + A.x +  
            "y = " + this.y;  
    }  
}
```

```
static public void incrementa() {  
    A.x++;  
    this.y++; // Incorrecto  
}
```

Atributos y métodos estáticos

```
public class A {  
    private static int x = 5;  
    private int y;  
  
    public A (int y) {  
        this.y = y;  
    }  
  
    static public void incrementa() {  
        A.x++;  
    }  
  
    public String toString() {  
        return "x = " + A.x +  
            "y = " + this.y;  
    }  
}
```

```
static public void incrementa() {  
    A.x++;  
    this.y++; // Incorrecto  
}
```

```
static public void incrementa() {  
    A.x++;  
    A a = new A(3);  
    a.y++;  
}
```

Atributos y métodos estáticos

```
public class A {  
    private static int x = 5;  
    private int y;  
  
    public A (int y) {  
        this.y = y;  
    }  
  
    static public void incrementa() {  
        A.x++;  
    }  
  
    public String toString() {  
        return "x = " + A.x +  
            "y = " + this.y;  
    }  
}
```

```
static public void incrementa() {  
    A.x++;  
    this.y++; // Incorrecto  
}
```

```
static public void incrementa() {  
    A.x++;  
    A a = new A(3);  
    a.y++;  
}
```

→ ¿Correcto o incorrecto?

Paso de parámetros en Java

En Java distinguimos los tipos primitivos y los tipos referencia (punteros).

Los tipos primitivos son `int`, `float`, `char`, `boolean`, etc., es decir, empiezan por minúsculas.

Los tipos primitivos, al no ser objetos, **no disponen** de método `equals`. Su comparación se hace con `=="`.

Los objetos y los arrays se consideran tipos **referencia**, es decir internamente son punteros.

Paso de parámetros en Java

Cuando un objeto hace de parámetro en un método, la dirección a la que apunta el objeto **nunca** cambia, aunque sí puede cambiar el contenido de lo apuntado.

Cuando se asignan dos objetos implica que **ambos** objetos apuntan a la **misma** dirección. Además, una vez asignados los objetos, **cualquier** cambio en cualquiera de los objetos **afecta** a ambos.

Paso de parámetros en Java

No sólo ocurre con la asignación sino también con el paso de parámetros que es por **valor**. Recordemos que el paso de parámetros en Java **siempre** es por valor.

Paso de parámetros en Java

No sólo ocurre con la asignación sino también con el paso de parámetros que es por **valor**. Recordemos que el paso de parámetros en Java **siempre** es por valor.


Tarea: ¿qué es exactamente el paso de parámetros por **valor**? ¿Qué diferencia hay con el paso de parámetros por **referencia**?

Paso de parámetros en Java

```
public class A {  
    private int x;  
  
    public A(int x) {  
        this.x = x;  
    }  
  
    public void set(int x) {  
        this.x = x;  
    }  
  
    public String toString() {  
        return "x = " + this.x;  
    }  
}
```


Paso de parámetros en Java

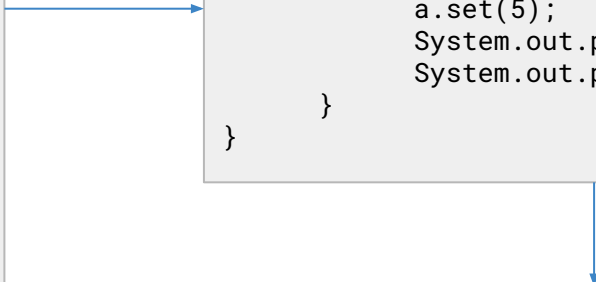
```
public class A {  
    private int x;  
  
    public A(int x) {  
        this.x = x;  
    }  
  
    public void set(int x) {  
        this.x = x;  
    }  
  
    public String toString() {  
        return "x = " + this.x;  
    }  
}
```



```
public class Main {  
    public static void main(String[] args) {  
        A a = new A(3);  
        A b = a;  
        a.set(5);  
        System.out.println(a);  
        System.out.println(b);  
    }  
}
```

Paso de parámetros en Java

```
public class A {  
    private int x;  
  
    public A(int x) {  
        this.x = x;  
    }  
  
    public void set(int x) {  
        this.x = x;  
    }  
  
    public String toString() {  
        return "x = " + this.x;  
    }  
}
```



```
public class Main {  
    public static void main(String[] args) {  
        A a = new A(3);  
        A b = a;  
        a.set(5);  
        System.out.println(a);  
        System.out.println(b);  
    }  
}
```

¿Qué muestra por pantalla?

Tipos primitivos

En java, los tipos primitivos son un tipo de dato heredado de lenguajes de programación **no orientada** a objetos, como puede ser C.

Tienen ciertas particularidades, que no comparten con otras clases más complejas, como por ejemplo que no necesitan ser declarados.

Tipos primitivos

En java, los tipos primitivos son un tipo de dato heredado de lenguajes de programación **no orientada** a objetos, como puede ser C.

Tienen ciertas particularidades, que no comparten con otras clases más complejas, como por ejemplo que no necesitan ser declarados.

int

double

boolean

char

...



Clases Wrapper

Entender la diferencia entre `int` e `Integer` puede ayudarnos mucho en el desarrollo de aplicaciones.

```
public class Cliente {  
    private int edad;  
    private String nombre;  
}
```

Clases Wrapper

Entender la diferencia entre `int` e `Integer` puede ayudarnos mucho en el desarrollo de aplicaciones.

```
public class Cliente {  
    private int edad;  
    private String nombre;  
}
```

Supongamos que en esta clase el atributo `edad` es opcional. ¿Qué ocurre si solicitamos la `edad` no habiendo asignado ningún valor?

```
public static void main(String[] args) {  
    Cliente cliente = new Cliente();  
    System.out.println(cliente.getEdad());  
    System.out.println(cliente.getNombre());  
}
```

Clases Wrapper

Entender la diferencia entre `int` e `Integer` puede ayudarnos mucho en el desarrollo de aplicaciones.

```
public class Cliente {  
    private int edad;  
    private String nombre;  
}
```

Supongamos que en esta clase el atributo `edad` es opcional. ¿Qué ocurre si solicitamos la `edad` no habiendo asignado ningún valor?

```
public static void main(String[] args) {  
    Cliente cliente = new Cliente();  
    System.out.println(cliente.getEdad());  
    System.out.println(cliente.getNombre());  
}
```



0
null

Clases Wrapper

¿Por qué la edad es 0? En ningún momento hemos pasado ese valor. Además, ¿“edad 0”? Eso no existe en la práctica.

Cuando declaramos que un atributo es de algún tipo primitivo, ese atributo **tiene** un valor predeterminado. Pero probablemente esto no es lo que queramos... Si el cliente registra la edad, queremos guardar la edad. De lo contrario, no queremos guardar **ningún** valor.

Un wrapper es una clase que **representa** un tipo primitivo.

Clases Wrapper

Una de las ventajas es que podemos asignar **valores nulos**.

Pero asignar valores nulos sin ningún criterio puede conllevar errores en tiempo de ejecución de tipo `NullPointerException`.

Clases Wrapper

Una de las ventajas es que podemos asignar **valores nulos**. Pero asignar valores nulos sin ningún criterio puede conllevar errores en tiempo de ejecución de tipo `NullPointerException`.

También podemos hablar de los **métodos disponibles**. Veamos los métodos disponibles para edad de tipo `int`.

```
private int edad;  
This.edad. // ...?
```

Exacto, ninguno.

Clases Wrapper

Ahora, veamos los métodos disponibles para edad de tipo Integer.

```
private Integer edad;  
this.edad.
```

Clases Wrapper

Ahora, veamos los métodos disponibles para edad de tipo Integer.

```
private Integer edad;  
this.edad.
```



Todos estos:

<https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html>

Clases Wrapper

Los wrappers, como `Integer`, son útiles cuando necesitamos usar nuestra variable en colecciones o queremos dejar algún atributo opcional con un **valor nulo**.

Los tipos primitivos son excelentes para cuando no queremos nulos y para operaciones matemáticas, ya que ocupan **poco espacio** en la memoria, mejorando el **rendimiento** de su aplicación.

Clases Wrapper

Los wrappers, como Integer, son útiles cuando necesitamos usar nuestra variable en colecciones o queremos dejar algún atributo opcional con un **valor nulo**.

Los tipos primitivos son excelentes para cuando no queremos nulos y para operaciones matemáticas, ya que ocupan **poco espacio** en la memoria, mejorando el **rendimiento** de su aplicación.

Tarea: elabora una tabla para ver las diferencias entre el tipo `int` e `Integer` en cuanto a: Type, Purpose, Flexibility, Memory allocation, Casting y Allowed operations.

Boxing y Unboxing

Dado que los tipos primitivos **no son** objetos, existe la posibilidad de transformarlos a objetos. Cada tipo primitivo tiene **asociada** una clase:

- `int` : clase `java.lang.Integer`
- `float`: clase `java.lang.Float`
- `char`: clase `java.lang.Character`

- Boxing: un tipo primitivo se **mete** en su clase equivalente.
- Unboxing: un objeto de una clase **asociada** a un tipo primitivo se saca de la clase.

Boxing y Unboxing

```
public class Main {  
    public static void main(String[] args){  
  
        // Boxing  
        Integer a;  
        int b = 15;  
        Character c;  
        char d = 'z';  
        a = 15; // a = new Integer(15);  
        a = b; // a = new Integer(b);  
        c = d; // c = new Character(d);  
  
        // Unboxing  
        b = a; // b = a.intValue();  
        d = c; // d = c.charValue();  
    }  
}
```


Diagramas de flujo

Símbolos utilizados y ejercicios

Definición

Los diagramas de flujo son **representaciones gráficas** de un algoritmo o un proceso. Se utilizan en multitud de disciplinas tales como programación, economía, procesos industriales y psicología cognitiva.

Estos diagramas se construyen a través de una **serie de símbolos** predefinidos, los cuales nos ayudan a realizar tareas siguiendo un mismo marco de trabajo.

A continuación veremos el conjunto de símbolos más usados y realizaremos algunos ejercicios con pseudocódigo y diagramas de flujo.

Símbolos I



Representa el
inicio y final
de un programa



Representa entrada
y salida de datos



Símbolo de entrada
por teclado



Salida por
pantalla



Entrada y salida
de la impresora



Símbolo de decisión
verdadero o falso



Representa un paso
dentro de un
proceso



Indica que el flujo
continúa donde se ha
colocado un símbolo
idéntico

Símbolos II



Llamada a
subrutina o
procedimiento



Añadir un
comentario



Representa
entrada/salida de
disco magnético
(almacenamiento)

Para construir los diagramas puedes usar la herramienta online **Smart Draw**. Si necesitas un correo temporal, puedes usar **Temp Mail**.



Ejercicios I



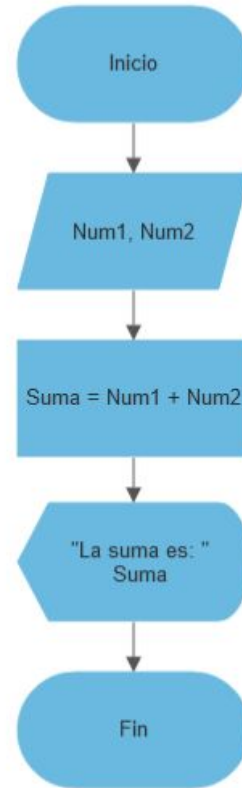
Inicio

Leer Num1, Num2

Suma = Num1 + Num2

Imprimir "La suma es: " Suma

Fin



Ejercicios II

Inicio

```
Leer A, B
if A > B then
    Imprimir "El mayor es: " A
if not
    if A = B then
        Imprimir "Son iguales"
    if not
        Imprimir "El mayor es: " B
    end if
end if
```

Fin

Ejercicios III

```
Inicio  
  do  
    Imprimir "Escribe dos números"  
    Leer A, B  
  while A = B  
Fin
```

Ejercicios IV

Inicio

Declarar Cuenta = 0

Declarar Suma = 0

do

Imprimir "Escribe un número"

Leer A

Cuenta = Cuenta + 1

Suma = Suma + A

while Cuenta = 10

Imprimir "La suma es: " Suma

Fin

Ejercicios V

Inicio

```
    Declarar Cuenta = 0
    Declarar Suma = 0
    Declarar Media
    Abrir fichero Notas
    Leer registro (Nombre, Curso, Nota)
    while no EOF do
        Imprimir Nombre, Curso, Nota
        Cuenta = Cuenta + 1
        Suma = Suma + Nota
        Leer registro (Nombre, Curso, Nota)
    end while
    Media = Suma / Cuenta
    Imprimir "Nota media: " Media
    Cerrar fichero Notas
```

Fin

Hacer cuando se vea el
tema de lectura y
escritura de ficheros

Ejercicios VI

Inicio

Declarar Cuenta = 0

Declarar Suma = 0

do

Imprimir "Escribe un número"

Leer A

Cuenta = Cuenta + 1

Suma = Suma + A

while Cuenta = 10

Imprimir "La suma es: " Suma

Fin

Ejercicios VII

Inicio

```
Leer N
switch N do
case 1
    Imprimir "Lunes"
case 2
    Imprimir "Martes"
case 3
    Imprimir "Miércoles"
case 4
    Imprimir "Jueves"
case 5
    Imprimir "Viernes"
case 6
    Imprimir "Sábado"
case 7
    Imprimir "Domingo"
default
    Imprimir "No válido"
end switch
```

Fin

Ejercicios VIII, usando lógica en lenguaje natural

Traducir problemas de lenguaje natural a condiciones lógicas en programación es uno de los mayores dolores de cabeza que suelen tener los programadores.

Tarea: Construye un diagrama de flujo que permita averiguar cuándo un año es bisiesto. En segundo lugar, obtén el pseudocódigo de dicho diagrama. Por último, haz un programa en Java que realice esa tarea.

Introducción a la P00



Curso 2023/24