



UNIVERSIDAD  
**NACIONAL**  
DE COLOMBIA  
SEDE BOGOTÁ

# PROGRAMACIÓN ORIENTADA A OBJETOS

SEBASTIÁN DAVID MORENO BERNAL  
CRISTIAN CAMILO ORJUELA VELANDIA

# CONTENIDO


1. Introducción
2. Historia
3. Filosofía del paradigma
4. Conceptos clave
5. Principios de la POO
6. Ventajas y desventajas
7. Lenguajes de programación
8. Aplicaciones
9. Referencias y bibliografía



# 1. INTRODUCCIÓN

## 2. HISTORIA



- 
- La *Programación orientada a Objetos* es un paradigma de programación que tiene como objetivo la implementación basada en una colección de objetos que están estructurados en clases.
  - Aparece como parte de la evolución de la programación y se establece como un enfoque diferente al momento de obtener resultados.



1. INTRODUCCIÓN

## **2. HISTORIA**

3. FILOSOFÍA DEL PARADIGMA



- Surge en un Centro de Computación Noruego en los años 60's con la implementación un lenguaje llamado Simula 67 por Krinsten Nygaard y Ole-Johan Dahl.
- Simula 67 inicia implementando los conceptos de clases, subclases y rutinas.



1. GitHub Pages – Paradigmas

# Línea del tiempo

Programación de computadores

Simula 67

C++  
Objective C  
Ada

Python  
Ruby

Java

HTML

Javascript  
C#

ABAP

Oz

Maya

50

67

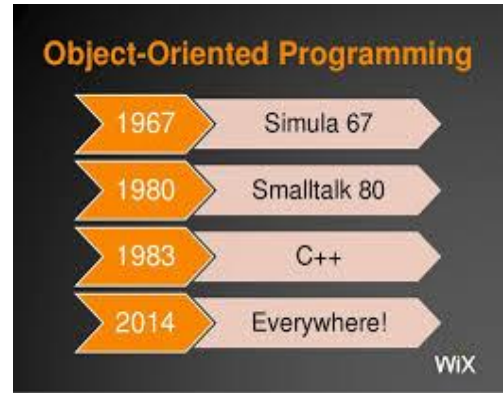
80's

96

2000

2010

Hoy





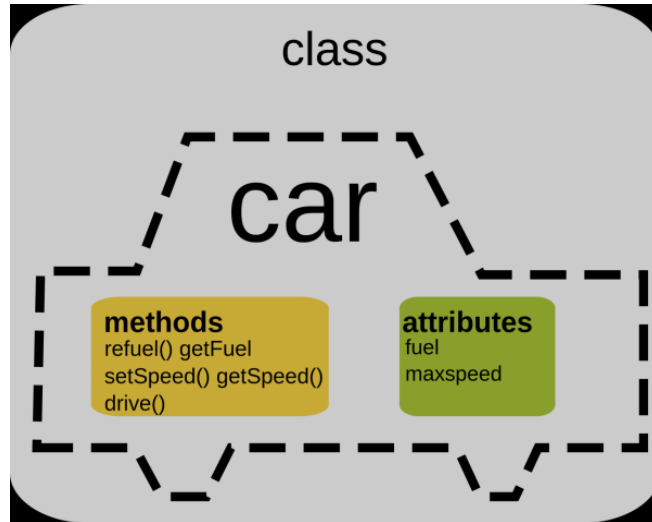
2. HISTORIA

## **3. FILOSOFÍA DEL PARADIGMA**

4. CONCEPTOS CLAVE







3. Car Class Example

- La implementación y desarrollo del paradigma está fundamentado en los **objetos**.
- Dar prioridad a los objetos y su abstracción como una parte fundamental en la solución de problemas.
- Definir los métodos, propiedades y características de los objetos así como su relación(interacción).




3. FILOSOFÍA DEL PARADIGMA

## **4. CONCEPTOS CLAVE**

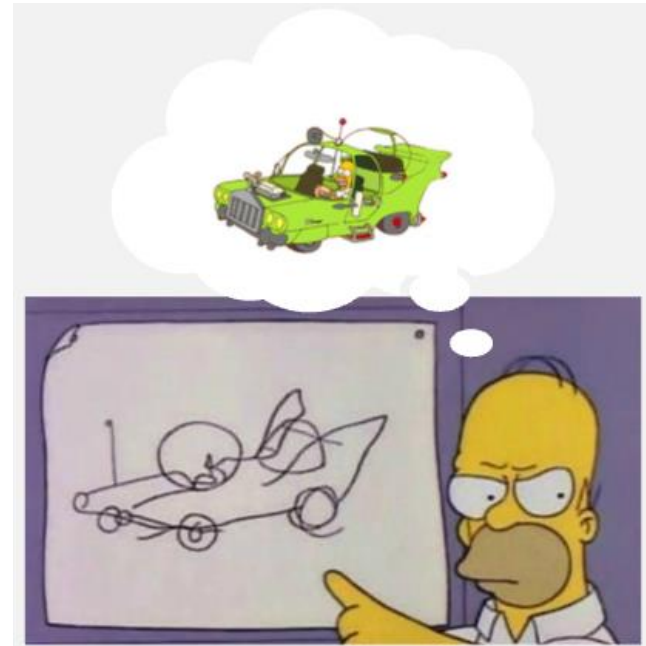
5. PRINCIPIOS DE LA POO



- 
- Abstracción
  - Modularidad
  - Encapsulamiento
  - Herencia
  - Polimorfismo

# Abstracción

Proceso que implica reconocer y enfocarse en las características importantes de una situación u objeto, y filtrar o ignorar todos los detalles no esenciales.



## Clase

Son tipos complejos que tienen múltiples piezas de información con propiedades (o atributos) y comportamientos (o métodos).

```
class MiClase:
    """Simple clase de ejemplo"""
    i = 12345      → Atributo
    def f(self): → Método
        return 'hola mundo'
```

## Objeto

Instancia

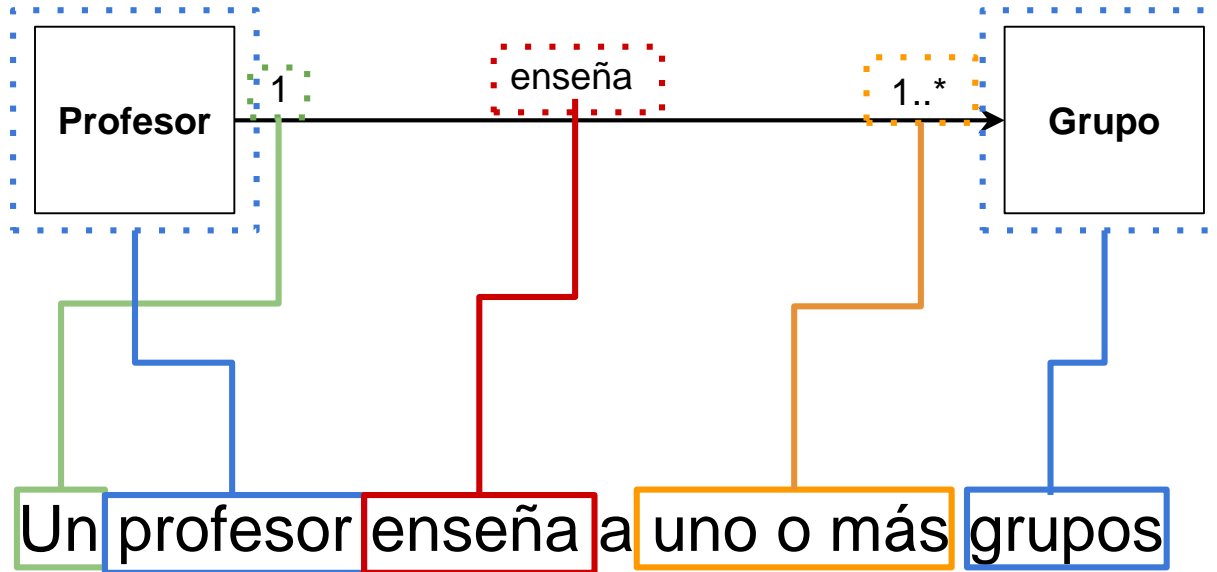


```
x = MiClase( )
```

Constructor

```
def __init__( self, i ):
    self.i = i
```

# Diagrama de clases

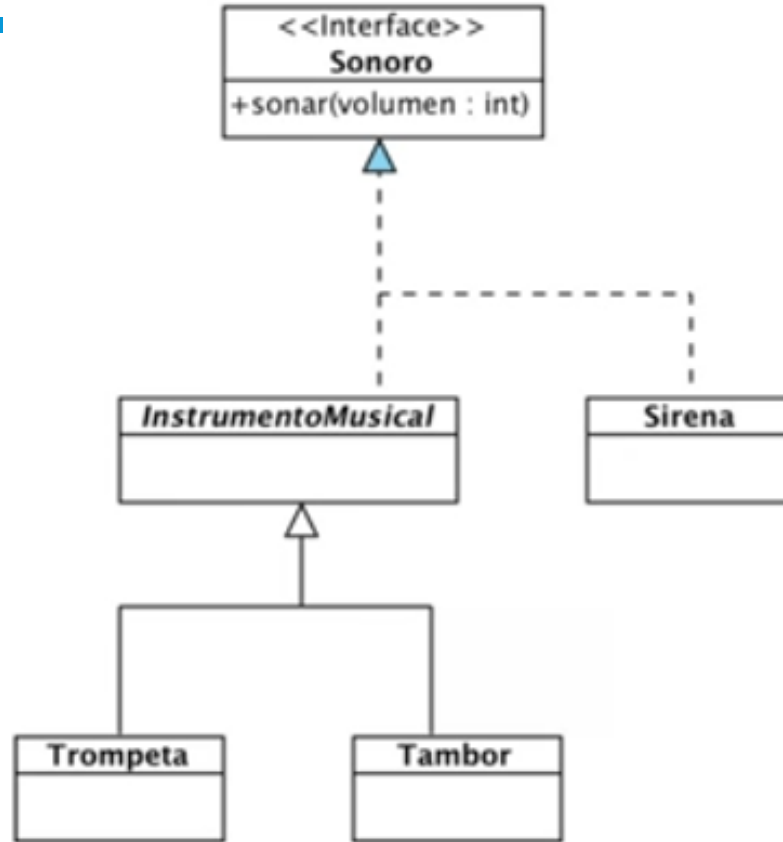


## Clase Abstracta

Es un tipo de clase con sentido nominativo que recoge las características comunes de otra serie de clases.

## Interfaz

Una clase que implementa una interfaz necesita implementar la funcionalidad de negocio real (los métodos).

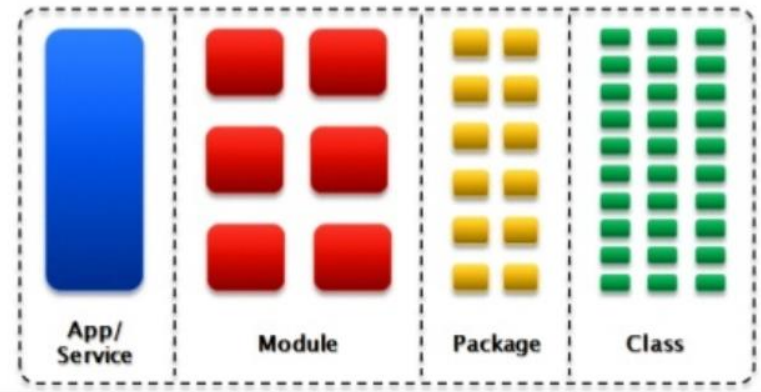




# Modularidad

- ❑ Propiedad que permite subdividir una aplicación en partes más pequeñas.
- ❑ La modularidad debe seguir los conceptos de bajo acoplamiento y alta cohesión.

## Modularity in Java



10. Modularidad en Java

Malas prácticas



# Encapsulamiento

- Es el proceso de ocultar todos los detalles internos de un objeto del mundo exterior.
- Es una barrera protectora que impide que el código y los datos sean accesibles al azar por otro código o por fuera de la clase.

```
public class Student {
```

```
    private int id;
```

```
    private String name;
```

```
    private String surName;
```

```
    private Date birthDate;
```

```
    private double papa;
```

```
    // advisor ???
```

```
    // courses ???
```

```
    public String getName() {  
        return "My name is: " + this.name;  
    }
```

Accesor

```
    public void setName(String name) {  
        this.name = name;  
    }
```

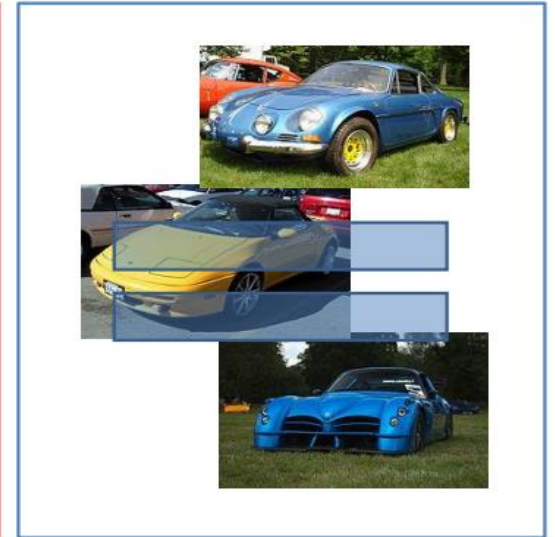
Mutador

```
}
```

# Herencia

Una clase heredada de otra quiere decir que esa clase obtiene los mismos métodos y propiedades de la otra clase.

Jerarquía



```
class Persona {
public:
    Persona(char *n, int e);
    const char *LeerNombre(char *n) const;
    int LeerEdad() const;
    void CambiarNombre(const char *n);
    void CambiarEdad(int e);

protected:
    char nombre[40];
    int edad;
};
```



```
// ...
class Empleado : public Persona {
public:
    Empleado(char *n, int e, float s);
    float LeerSalario() const;
    void CambiarSalario(const float s);

protected:
    float salarioAnual;
};
```

# Polimorfismo

Es la habilidad de dos o más objetos pertenecientes a diferentes clases para responder exactamente al mismo mensaje (llamada de método) de diferentes formas específicas de la clase.



Sobrecarga de  
métodos

# Sobrecarga en ABAP4

```
CLASS empleado DEFINITION.
```

```
    DATA nombre TYPE string.
```

```
    DATA apellido TYPE string.
```

```
    METHODS: modificarNombre IMPORTING i_nombre TYPE string,  
             modificarNombre IMPORTING i_nombre TYPE string  
             i_apellido TYPE string.
```

```
ENDCLASS.
```



# Sobrecarga en ABAP4

CLASS empleado IMPLEMENTATION.

METHOD modificarNombre IMPORTING i\_nombre TYPE string.

nombre = i\_nombre.

ENDMETHOD.

METHOD modificarNombre IMPORTING i\_nombre TYPE string

i\_apellido TYPE string.

nombre = i\_nombre.

apellido = i\_apellido.

ENDMETHOD.

ENDCLASS.



4. CONCEPTOS CLAVE

## **5. PRINCIPIOS DE LA POO**

6. VENTAJAS Y DESVENTAJAS



# Principios de la POO

- Según Robert C. Martin existen 5 principios básicos que constituyen la programación orientada a objetos.

SRP - Single Responsibility Principle

OCP - Open Closed Principle

LSP - Liskov Substitution Principle

ISP - Interface Segregation Principle

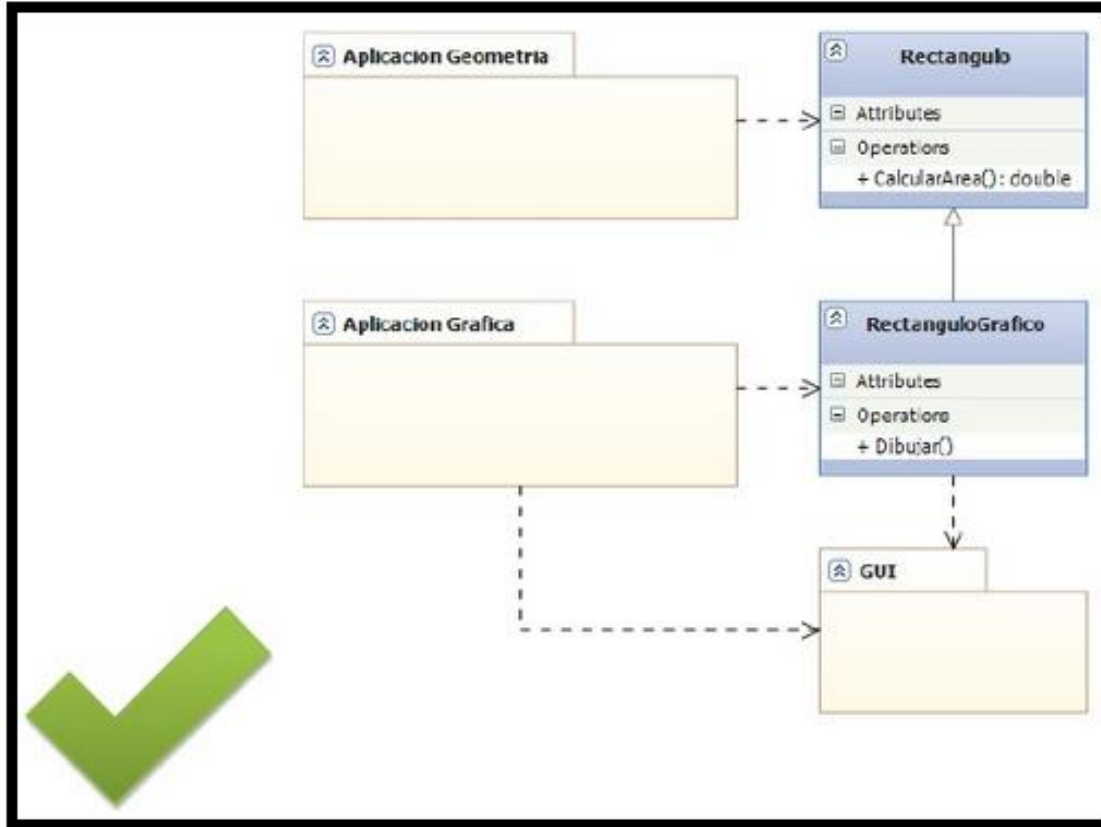
DIP - Dependency Inversion Principle



18. Robert Cecil Martin

## Principio de responsabilidad única

- Cada clase debe tener una única responsabilidad, y esta responsabilidad debe estar contenida únicamente en esa clase.
- Cada responsabilidad es el eje y la razón de cambio.
- Para contener la propagación del cambio, se deben separar las responsabilidades.



19. Principio de responsabilidad única

## Principio de abierto - cerrado

- Una entidad (clase, módulo, función, etc.) debe quedarse abierta para su extensión, pero cerrada para su modificación.
- Si un cambio impacta a varios módulos, entonces la aplicación no está bien diseñada.
- Se deben diseñar módulos que procuren no cambiar y así, reutilizar el código más adelante (extensión).

```
public abstract class Empleado
{
    public string Nombre { get; set; }
    public double Sueldo { get; set; }
    public double Bono { get; set; }

    public abstract void CalcularBono();
}

public class Programador : Empleado
{
    public override void CalcularBono()
    {
        Bono = Sueldo * 2;
    }
}

public class Gerente : Empleado
{
    public override void CalcularBono()
    {
        Bono = Sueldo * 15;
    }
}

public class EmpleadosServicio
{
    public List<Empleado> Empleados { get; set; }

    public void CalcularBonos()
    {
        foreach (var empleado in Empleados)
        {
            empleado.CalcularBono();
        }
    }
}
```



## Principio de sustitución de Liskov

- Si en alguna parte de un programa se utiliza una clase, y esta clase es extendida, se puede utilizar cualquiera de las clases hijas sin que existan modificaciones en el código.
- Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.



```

foreach (var empleado in Empleados)
{
    if (empleado is Programador)
    {
        empleado.Bono = empleado.Sueldo * 2;
    }
    else if (empleado is Gerente)
    {
        empleado.Bono = empleado.Sueldo * 10;
    }
}

public void CalcularBonos()
{
    foreach (var empleado in Empleados)
    {
        empleado.CalcularBono();
    }
}

```



21. Principio de sustitución de Liskov

## Principio de segregación de interfaz

- Este principio hace referencia a que muchas interfaces cliente específicas son mejores que una interfaz de propósito general.
- Se aplica a una interfaz amplia y compleja para dividirla en otras más pequeñas y específicas, de tal forma que cada cliente use solo aquella que necesite pudiendo así ignorar al resto.

```
public class Perro : Animal
{
    public override void Alimentar()
    {
        // Alimentar al perro
    }

    public override void Acariciar()
    {
        // Acariciar al perro
    }
}
```



```
public abstract class Animal
{
    public abstract void Alimentar();
    public abstract void Acariciar();
}
```

```
public class Escorpion : Animal
{
    public override void Alimentar()
    {
        // Alimentar al escorpion
    }

    public override void Acariciar()
    {
        // Estas loco ?????!
    }
}
```

- Interfaces detalladas.
- Implementar métodos necesarios.

```
public abstract class Animal
{
    public abstract void Alimentar();
}

public interface IMascota
{
    void Acariciar();
}
```

```
public class Perro : Animal, IMascota
{
    public override void Alimentar()
    {
        // Alimentar al perro
    }

    public void Acariciar()
    {
        // Acariciar al perro
    }
}
```

```
public class Escorpion : Animal
{
    public override void Alimentar()
    {
        // Alimentar al escorpion
    }
}
```



23. Principio de segregación de interfaz

# Principio de inversión de dependencias

```
public interface Persistence {
    void save(Shopping shopping);
}

public class SqlDatabase implements Persistence {

    @Override
    public void save(Shopping shopping){
        // Saves data in SQL database
    }
}


public interface PaymentMethod {
    void pay(Shopping shopping);
}

public class CreditCard implements PaymentMethod {

    @Override
    public void pay(Shopping shopping){
        // Performs payment using a credit card
    }
}
```

- Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

```
public class ShoppingBasket {  
  
    private final Persistence persistence;  
    private final PaymentMethod paymentMethod;  
  
    public ShoppingBasket(Persistence persistence, PaymentMethod payment  
Method) {  
        this.persistence = persistence;  
        this.paymentMethod = paymentMethod;  
    }  
  
    public void buy(Shopping shopping) {  
        persistence.save(shopping);  
        paymentMethod.pay(shopping);  
    }  
}
```



25. Principio de inversión de dependencias



5. PRINCIPIOS DE LA POO

## **6. VENTAJAS Y DESVENTAJAS**

7. LENGUAJES DE PROGRAMACIÓN



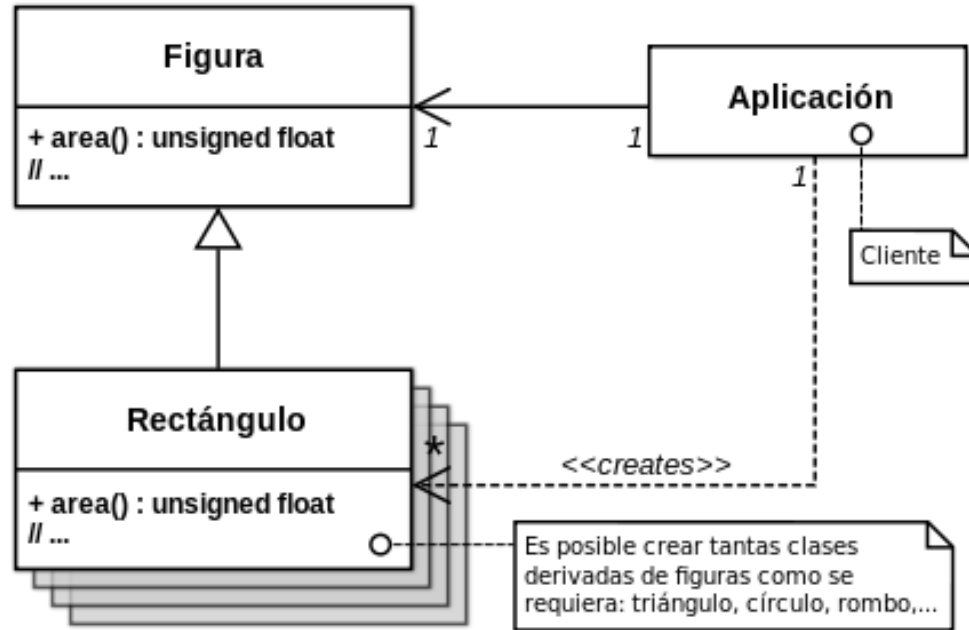
# Ventajas

- Reducción de código redundante, lo que permite un código conciso y sin repeticiones. (Herencia)
- Posibilita reusar código y extenderlo a través de la clases sin necesidad de probarlo. (Testing)
- La jerarquía y abstracción de los objetos brinda una implementación más detallada, puntual y coherente.



# Ventajas

- ✓ La implementación de clases y objetos proporciona una relación más directa con la realidad al implementar funciones y métodos como comportamientos de las entidades.
- ✓ Bajo acoplamiento y alta cohesión: Gracias a la modularidad, cada componente o módulo de un desarrollo tiene independencia de los demás componentes.
- ✓ Facilidad en el desarrollo y el mantenimiento debido a la filosofía del paradigma.



26. Diagrama de clases

# Desventajas

- × Velocidad de ejecución.
- × Se hereda código no usable en la nueva clase.
- × El uso para tareas simples termina siendo improductivo.



6. VENTAJAS Y DESVENTAJAS

# 7. LENGUAJES DE PROGRAMACIÓN

8. APLICACIONES



# Lenguajes de programación

Un lenguaje es orientado a objetos si cumple con lo siguiente:

- ❑ Soporta objetos que son abstracciones de datos con una interfaz de operaciones con nombre y un estado local oculto.
- ❑ Los objetos tienen un tipo asociado (la clase).
- ❑ Los tipos (clases) pueden heredar atributos de los supertipos (superclases).

# Lenguajes de programación

- SmallTalk (Entorno puro o “mundo virtual” de objetos)
- Scala (Influido por Java)
- Perl (Soporta herencia múltiple)
- ABAP (SAP)
- D (Rediseño de C++, influido por Eiffel, C#)
- Eiffel
- Ruby
- Delphi
- Muchos más
- [https://en.wikipedia.org/wiki/List\\_of\\_object-oriented\\_programming\\_languages](https://en.wikipedia.org/wiki/List_of_object-oriented_programming_languages)

## Ejemplo en PHP

```
class Circle {
    public $radius;

    public function __construct($radius) {
        $this->radius = $radius;
    }
}

class Square {
    public $length;

    public function __construct($length) {
        $this->length = $length;
    }
}
```

## Ejemplo en C++

```
class Circle {
private:
    double radius;    // Data member (Variable)
    string color;    // Data member (Variable)

public:
    // Constructor with default values for data members
    Circle(double r = 1.0, string c = "red") {
        radius = r;
        color = c;
    }

    double getRadius() { // Member function (Getter)
        return radius;
    }
}
```

27, 28. Ejemplos de clases

## Ejemplo en Python

```
class Employee:
    'Common base class for all employees'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary
```

29, 30. Ejemplos de clases

## Ejemplo en C#

```
public class Person
{
    // Field
    public string name;

    // Constructor that takes no arguments.
    public Person()
    {
        name = "unknown";
    }

    // Constructor that takes one argument.
    public Person(string nm)
    {
        name = nm;
    }

    // Method
    public void SetName(string newName)
    {
        name = newName;
    }
}
```



## Ejemplo en C++

```
#include <iostream>
using namespace std;

class pareja {
private:
    // Datos miembro de la clase "pareja"
    int a, b;
public:
    // Funciones miembro de la clase "pareja"
    void Lee(int &a2, int &b2);
    void Guarda(int a2, int b2) {
        a = a2;
        b = b2;
    }
};
```

```
void pareja::Lee(int &a2, int &b2) {
    a2 = a;
    b2 = b;
}

int main() {
    pareja par1;
    int x, y;

    par1.Guarda(12, 32);
    par1.Lee(x, y);
    cout << "Valor de par1.a: " << x << endl;
    cout << "Valor de par1.b: " << y << endl;

    return 0;
}
```



7. LENGUAJES DE PROGRAMACIÓN

## **8. APLICACIONES**

9. REFERENCIA Y BIBLIOGRAFÍA

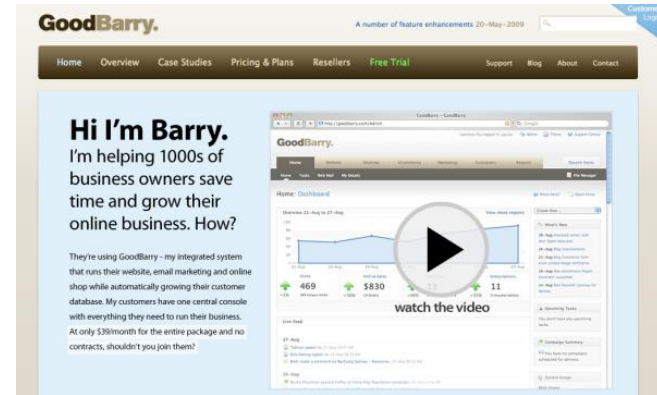


# Aplicaciones

“Cada elemento en el mundo real puede ser modelado e implementado como un objeto”.

- Twitter es hecho en SCALA (Objetos)
- Bases de datos orientadas a objetos.
- Interfaces de usuario.
- Modelamiento y simulación de agentes.

db4objects  
BY VERSANT



31- 34. Imágenes aplicaciones

# Serialización

La serialización consiste en un proceso de codificación de un objeto en un medio de almacenamiento con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato más legible (XML, JSON):

- ❑ El objeto serializado pueda guardarse en un fichero o puede enviarse por red para reconstruirlo en otro lugar.
- ❑ El objeto copiado es un copia idéntica al original.

# Referencias y Bibliografía

Conceptos básicos de POO:

- <https://msdn.microsoft.com/es-co/library/bb972232.aspx>
- <https://styde.net/que-es-la-programacion-orientada-a-objetos/>
- <http://c.conclase.net/curso/?cap=036>
- Durán F. Gutierrez F. Pimentel E. Programación orientada a objetos con Java . 2007. Editorial Thomson. Madrid.

Clases en Python

- <http://docs.python.org.ar/tutorial/2/classes.html>

Ventajas y aplicaciones de POO:

- <https://www.emaze.com/@ACZOZZLZ/poo>

# Referencias y Bibliografía

Serialización:

- <https://es.wikipedia.org/wiki/Serializaci%C3%B3n>

Presentaciones de programación orientada objetos de cursos anteriores.

- <https://drive.google.com/drive/u/1/folders/0B7IRdmOoUVf5fjVBd1lvSmFvN0xmZmoteS1rQzN3cIlTQVZRWnBhcGhBakxmazBQbEFKYmc>
- [https://prezi.com/iechoqrsv1ur/programacion-orientada-a-objetos/?utm\\_campaign=share&utm\\_medium=copy](https://prezi.com/iechoqrsv1ur/programacion-orientada-a-objetos/?utm_campaign=share&utm_medium=copy)
- <https://www.emaze.com/@ACZOZZLZ/poo>

Presentaciones del curso de programación orientada a objetos, Juan Mendivelso, UNAL. <https://sites.google.com/a/unal.edu.co/poo2014-2/>

# Referencias imágenes

1. GitHub Pages – Paradigmas de programación:

[http://ferestrepoqa.github.io/paradigmas-de-programacion/poo/poo\\_teoría/index.html](http://ferestrepoqa.github.io/paradigmas-de-programacion/poo/poo_teoría/index.html)

2. Fechas importantes

<https://www.timetoast.com/linea-del-tiempo-evolucion-de-los-lenguajes-de-programacion>

3. Clase Carro

[http://www.aprenderaprogramar.com/index.php?option=com\\_content&view=article](http://www.aprenderaprogramar.com/index.php?option=com_content&view=article)

4 - 9. Abstracción, Clase, Objeto, Interfaz

<https://www.mindomo.com/es/mindmap/programacion-orientada-a-objetos-699df6ea39c24846b53082db41e6f3b1>

10. Modularidad.

<https://www.slideshare.net/CristianoCostantini/modular-java-with-osgi-and-karaf>



11 – 16. Conceptos clave

<https://sites.google.com/a/unal.edu.co/poo2014-2/>

17, 19 - 24. SOLID Principles

<https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>

25. Robert Cecil Martin

<https://alchetron.com/Robert-Cecil-Martin-228711-W>

27 - 30. Ejemplos de lenguajes

[http://ferestrepora.github.io/paradigmas-de-programacion/poo/poo\\_teoría/examples.html](http://ferestrepora.github.io/paradigmas-de-programacion/poo/poo_teoría/examples.html)

31 – 34. Aplicaciones. [http://ferestrepora.github.io/paradigmas-de-programacion/poo/poo\\_teoría/index.html#applications](http://ferestrepora.github.io/paradigmas-de-programacion/poo/poo_teoría/index.html#applications)