Programación Orientada a Objetos

Integrantes:
Santiago Hernández Bolívar
Edwin Alexander Bohórquez

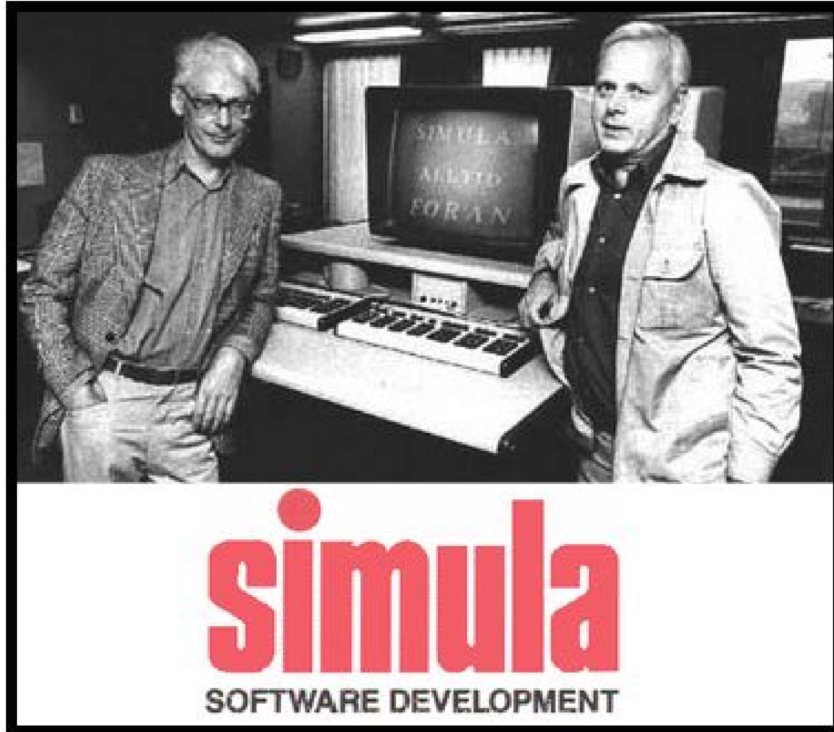
UNIVERSIDAD NACIONAL DE COLOMBIA

UNIVERSIDAD NACIONAL DE COLOMBIA

Tabla de Contenido

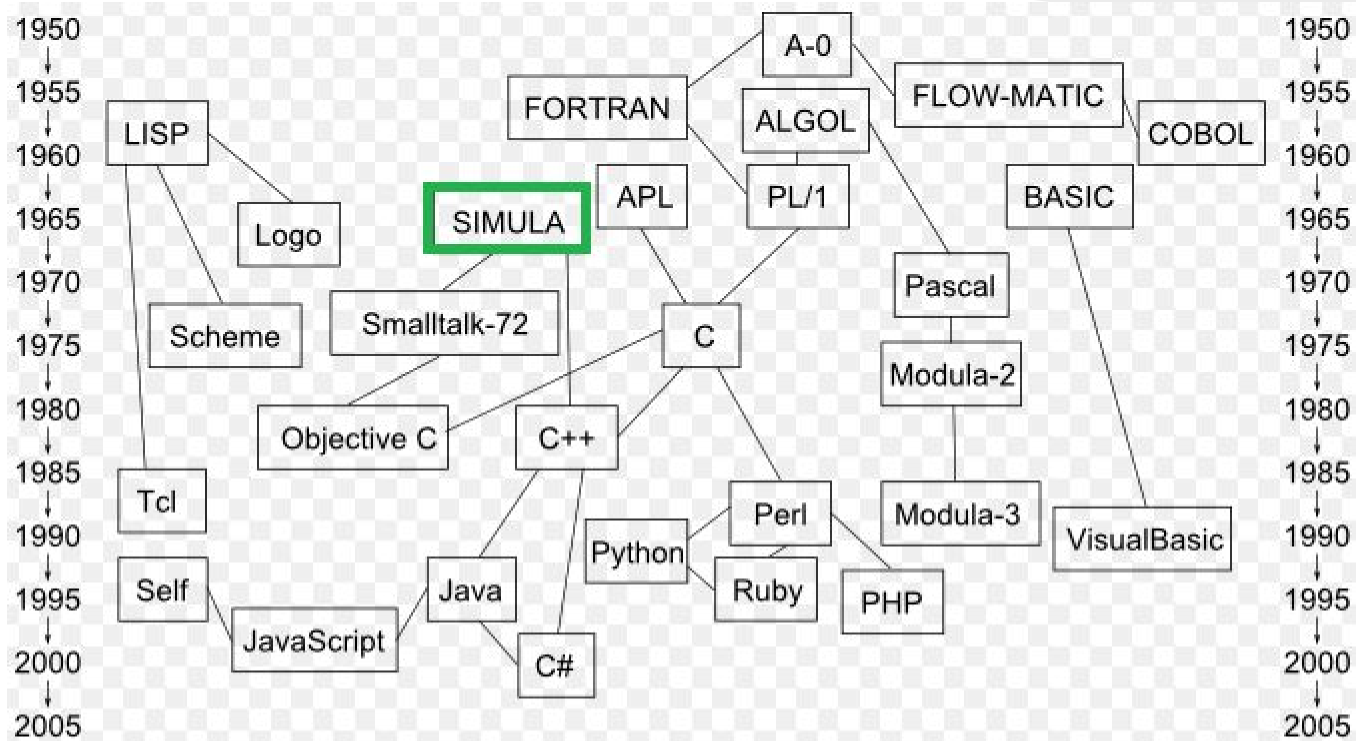
Historia.
Filosofía del paradigma.
Conceptos claves.
Ventajas y desventajas.
Lenguajes de programación.
Aplicaciones.
Referencias.

Historia



La Programación Orientada a Objetos surge en Noruega en 1967 con un lenguaje llamado Simula 67, desarrollado por Krinsten Nygaard y Ole-Johan Dahl, en el centro de cálculo noruego.
Simula 67 introdujo por primera vez los conceptos de clases, corrutinas y subclases.

Evolución



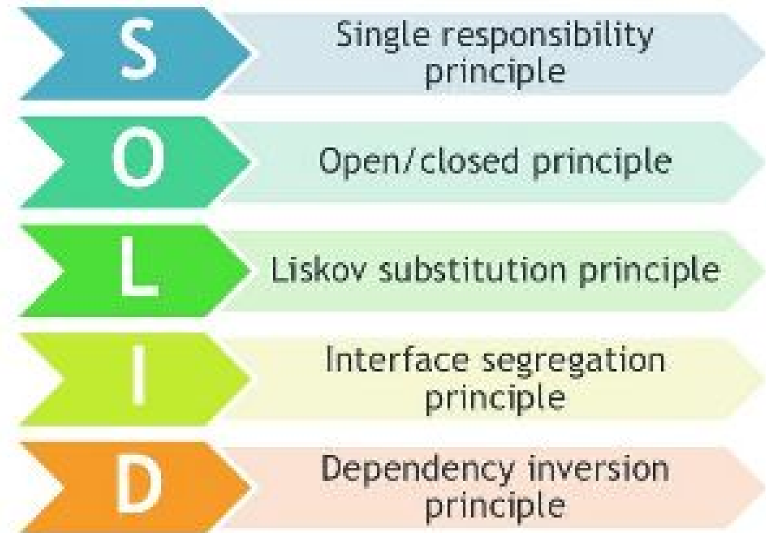
Filosofía del paradigma

- Pensar todo en términos de **objetos**.
- Representar los objetos de la forma más cercana a cómo expresamos las cosas en la vida real.
- Los programas se definen en términos de objetos, propiedades, métodos, y la interacción (comunicación) entre objetos.

Principios de la POO

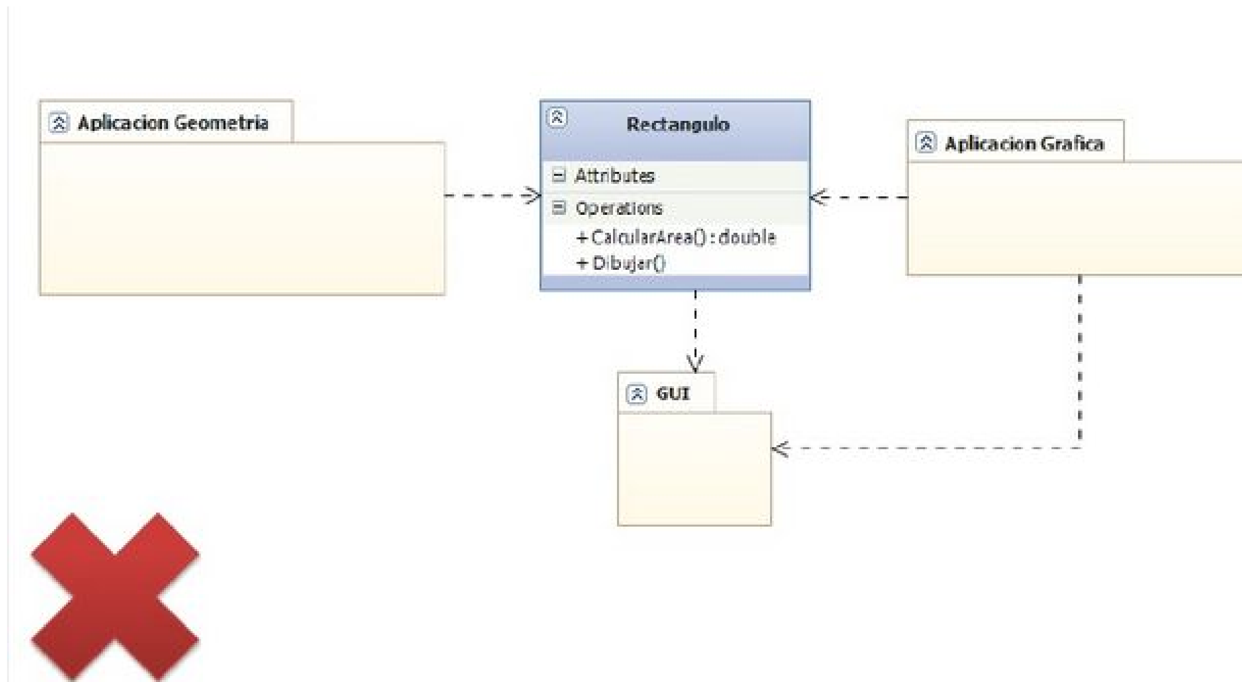
SOLID es un acrónimo mnemónico introducido por Robert C. Martín a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño.

- Principio de responsabilidad única.
- Principio de abierto/cerrado.
- Principio de sustitución de Liskov.
- Principio de segregación de la interfaz.
- Principio de inversión de la dependencia.



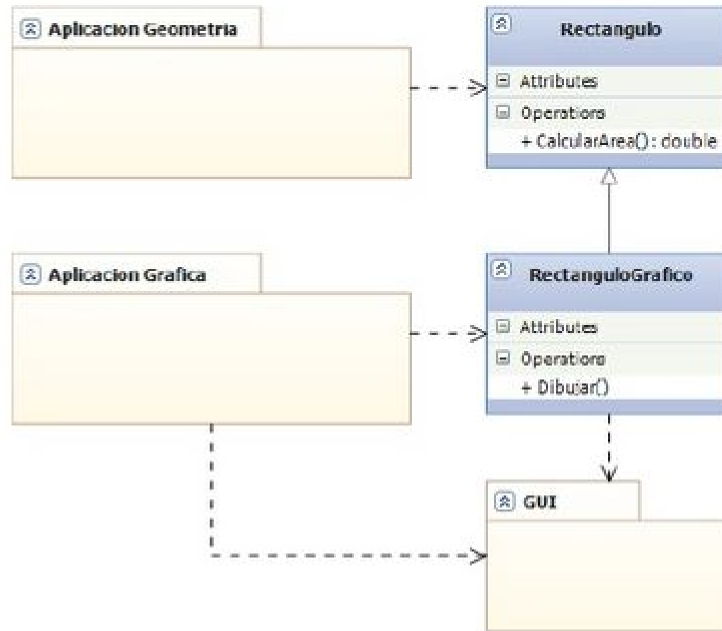
Principios de la POO

Principio de responsabilidad única.



Principios de la POO

Principio de responsabilidad única.



Principios de la POO

Principio de abierto/cerrado.

```
public enum TipoEmpleado
{
    Programador,
    Gerente
}

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

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

    public void CalcularBonos()
    {
        foreach (var empleado in Empleados)
        {
            double bono = 0;

            switch (empleado.Tipo)
            {
                case TipoEmpleado.Programador:
                    bono = empleado.Sueldo * 2;
                    break;
                case TipoEmpleado.Gerente:
                    bono = empleado.Sueldo * 10;
                    break;

                // Estamos contratando
                // analistas funcionales!!
            }
            empleado.Bono = bono;
        }
    }
}
```

Principios de la POO

Principio de abierto/cerrado.

```
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();
        }
    }
}
```

Principios de la POO

Principio de Sustitución de Liskov.

```
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();
    }
```

Principios de la POO

Principio de Segregación de la interfaz.

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

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

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

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

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

Principios de la POO

Principio de Segregación de la interfaz.

```
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
    }
}
```

Principios de la POO

Principio de Inversión de la dependencia.

```
public class ShoppingBasket {

    public void buy(Shopping shopping) {

        SqlDatabase db = new SqlDatabase();
        db.save(shopping);

        CreditCard creditCard = new CreditCard();
        creditCard.pay(shopping);
    }
}

public class SqlDatabase {
    public void save(Shopping shopping){
        // Saves data in SQL database
    }
}

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

Principios de la POO

Principio de Inversión de la dependencia.

```java
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
    }
}
```

```java
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);
    }
}
```

Conceptos clave

- Abstraccion.
- Clase
- Objeto.
- Modularidad.
- Modificadores de acceso.
- Encapsulamiento.
- Herencia.
- Polimorfismo.
- Interfaz.

Abstraccion

- Dejar a un lado los detalles de un objeto y definir las características específicas de éste, aquellas que lo distingan de los demás tipos de objetos.
- Hay que centrarse en lo que es y lo que hace un objeto, antes de decidir cómo debería ser implementado.
- Se hace énfasis en el ¿**Que hace**? más que en el ¿Cómo se hace?.

Ejemplo: Aplicar la abstracción a las aves:

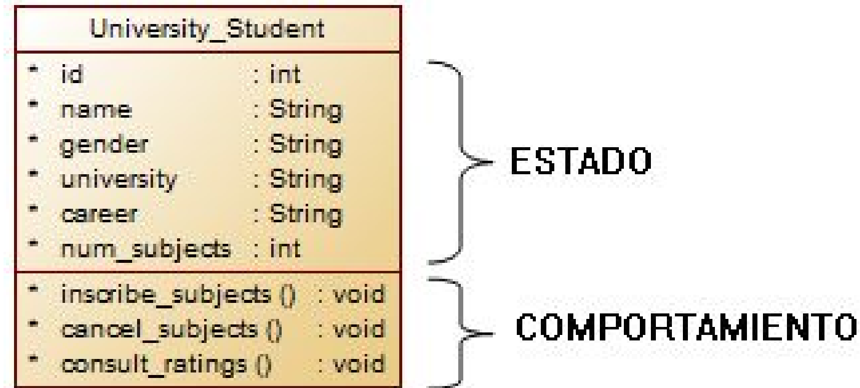
Objeto: Pajaro

Características:	**Funcionalidades:**
Pico	Volar
Alas	Picar
Patas	Parar
Plumas	

Clase

Es una construcción que se utiliza como un modelo (o plantilla) para crear objetos de ese tipo. El modelo describe el **estado** y contiene el **comportamiento** que todos los objetos creados a partir de esa clase tendrán.

Ejemplo:



Clase

Ejemplo en Java

University_Student	
* id	: int
* name	: String
* gender	: String
* university	: String
* career	: String
* num_subjects	: int
* inscribe_subjects ()	: void
* cancel_subjects ()	: void
* consult_ratings ()	: void

```
public class UniversityStudent {
    int id;
    String name;
    String gender;
    String university;
    String career;
    int numSubjects;
    public UniversityStudent(int id, String name, String gender,
        String university, String  career, int numSubjects) {
        this.id = id;
        this.name = name;
        this.gender = gender;
        this.university = university;
        this.career = career;
        this.numSubjects = numSubjects;
    }
    void inscribeSubjects() {
      // TODO: implement
    }
    void cancelSubjects() {
       // TODO: implement
    }
    void consultRatings() {
       // TODO: implement
    }
}
```

Clase

Ejemplo en C++

University_Student	
* id	: int
* name	: String
* gender	: String
* university	: String
* career	: String
* num_subjects	: int
* inscribe_subjects ()	: void
* cancel_subjects ()	: void
* consult_ratings ()	: void

```
class UniversityStudent {
    int id;
    string name;
    string gender;
    string university;
    string career;
    int numSubjects;

    public:
    UniversityStudent(int _id,string _name,string _gender,
        string _university,string _career,int _numSubjects){
        id = _id;
        name = _name;
        gender = _gender;
        university = _university;
        career = _career;
        numSubjects = _numSubjects;
    };
    void inscribeSubjects() {
        // TODO: implement
    }
    void cancelSubjects() {
        // TODO: implement
    }
    void consultRatings() {
        // TODO: implement
    }
};
```

Clase

Ejemplo en Python

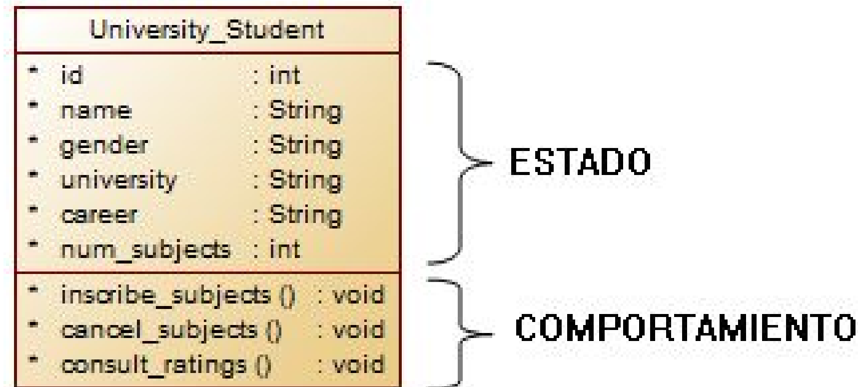
University_Student	
* id	: int
* name	: String
* gender	: String
* university	: String
* career	: String
* num_subjects	: int
* inscribe_subjects ()	: void
* cancel_subjects ()	: void
* consult_ratings ()	: void

```
class UniversityStudent:

    def __init__(self,id,name,gender,university,career,numsubjects):
        self.id = id
        self.name = name
        self.gender = gender
        self.university = university
        self.career = career
        self.numsubjects = numsubjects

    def inscribeSubjects(self):
        pass

    def cancelSubjects(self):
        pass

    def consultRatings(self):
        pass
```

Objeto

- Es una entidad real o abstracta, con un papel definido en el dominio del problema.
- Es una instancia de una clase, que tiene:
  - Identidad.
  - Estado (atributos).
  - Comportamiento (métodos).

Ejemplo:



Objeto

Ejemplo en Java

```
UniversityStudent student = new UniversityStudent(123, "Pepe", "masculino", "UN", "Medicina", 8);
```

Ejemplo en C++

Se crea un objeto de forma convencional:

```
UniversityStudent student(123, "Pepe", "masculino", "UN", "Medicina", 8);
```

Se crea un objeto usando memoria dinámica:

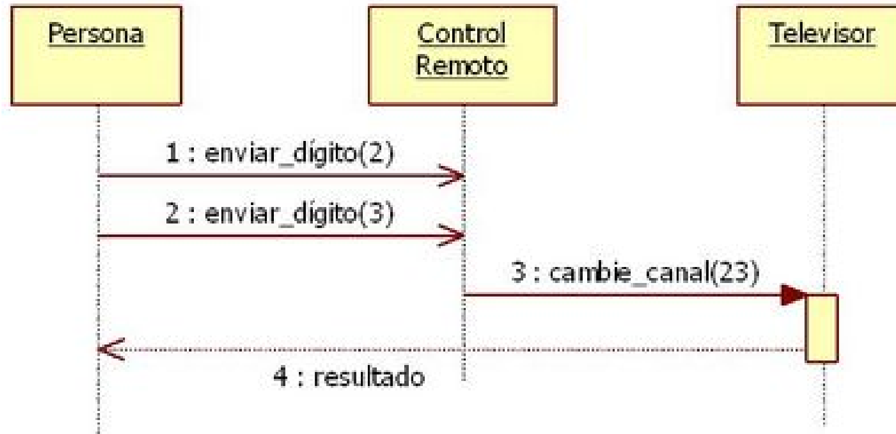
```
UniversityStudent *student = new UniversityStudent(123, "Pepe", "masculino", "UN", "Medicina", 8);
```

Ejemplo en Python

```
student = UniversityStudent(123, "Pepe", "masculino", "UN", "Medicina", 8)
```

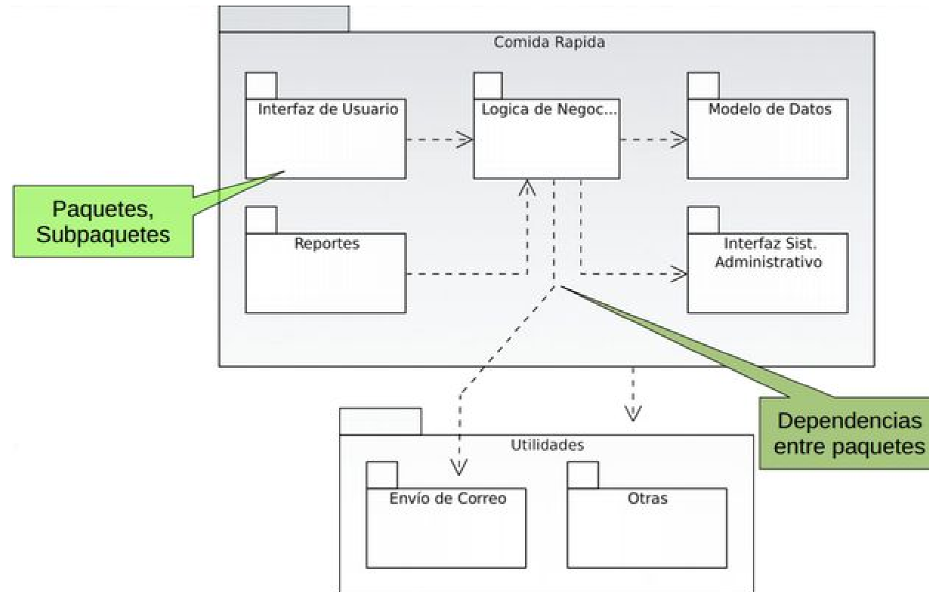
Objeto

Comunicación entre Objetos:
Un **Mensaje** es una comunicación dirigida desde un objeto A ordenando a otro objeto B que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.



Modularidad

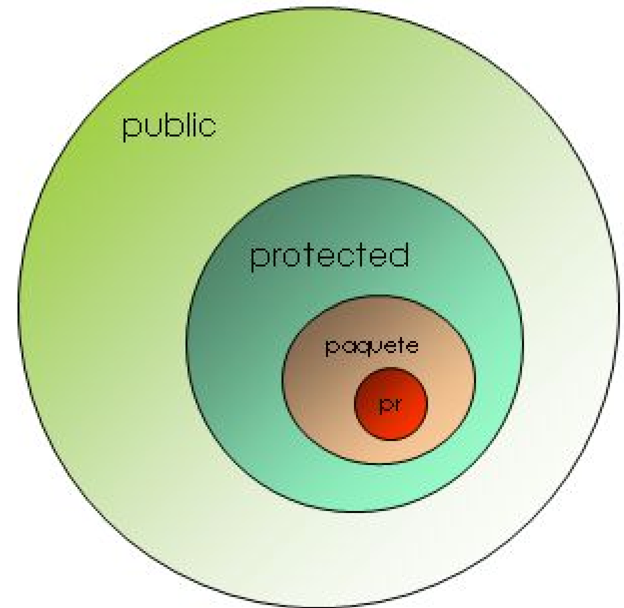
Es la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes.



Modificadores de Acceso

Los modificadores de acceso se utilizan para definir la visibilidad de los miembros de una clase.

	Clase	Paquete	Subclase	Otros
public	✓	✓	✓	✓
private	✓	✗	✗	✗
protected	✓	✓	✓	✗
Default (sin modificador)	✓	✓	✗	✗



Encapsulamiento

- Es la propiedad que permite asegurar que la información de un objeto está oculta del mundo exterior.
- El encapsulamiento consiste en agrupar en una Clase las características(atributos) con un acceso privado y los comportamientos (métodos) con un acceso público.
- Acceder o modificar los miembros de una clase a través de sus métodos.

Encapsulamiento

Cuando no hay encapsulamiento se pueden presentar problemas:

```
class Main{
    public static void main(String []args){
        UniversityStudent student = new UniversityStudent(123, "Pepe", "masculino", "UN", "Medicina", 8);
        student.numSubjects = -15;
        System.out.println("Numero de materias "+student.numSubjects);
    }
}
```

```
run:
Numero de materias -15
BUILD SUCCESSFUL (total time: 0 seconds)
```

Encapsulamiento

```java
public class UniversityStudent {
    private int id;
    private String name;
    private String gender;
    private String university;
    private String career;
    private int numSubjects;
    public UniversityStudent(int id, String name, String gender,
        String university, String  career, int numSubjects) {
        this.id = id;
        this.name = name;
        this.gender = gender;
        this.university = university;
        this.career = career;
        this.numSubjects = numSubjects;
    }

    public void inscribeSubjects() {
      // TODO: implement
    }
    public void cancelSubjects() {
       // TODO: implement
    }
    public void consultRatings() {
       // TODO: implement
    }
```

```java
    public void setNumSubjects(int numSubjects){
        if( numSubjects < 0 || numSubjects > 10 )
            System.out.println("Numero invalido de materias");
        else
            this.numSubjects = numSubjects;
    }
    public int getNumSubjects(){
        return numSubjects;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public String getGender() {
        return gender;
    }
    public String getUniversity() {
        return university;
    }
    public String getCareer() {
        return career;
    }
}
```

Encapsulamiento

Con encapsulamiento:

```
class Main{
   public static void main(String []args){
        UniversityStudent student = new UniversityStudent(123, "Pepe", "masculino", "UN", "Medicina", 8);
        student.setNumSubjects(-15);
        System.out.println("Numero de materias "+student.getNumSubjects());
        student.setNumSubjects(6);
        System.out.println("Numero de materias "+student.getNumSubjects());
   }
}
```

```
run:
Numero invalido de materias
Numero de materias 8
Numero de materias 6
BUILD SUCCESSFUL (total time: 0 seconds)
```

Encapsulamiento

```
class UniversityStudent {
   private:
   int id;
   string name;
   string gender;
   string university;
   string career;
   int numSubjects;

   public:
   UniversityStudent(int _id,string _name,string _gender,
      string _university,string _career,int _numSubjects){
      id = _id;
      name = _name;
      gender = _gender;
      university = _university;
      career = _career;
      numSubjects = _numSubjects;
   };
   void inscribeSubjects() {
      // TODO: implement
   }
   void cancelSubjects() {
      // TODO: implement
   }
   void consultRatings() {
      // TODO: implement
   }
```

```
   void setNumSubjects(int _numSubjects){
        if(_numSubjects < 0 || _numSubjects > 10)
            cout<<"Numero invalido de materias";
        else
            numSubjects = _numSubjects;
   }
   int getNumSubjects(){
        return numSubjects;
   }
   int getId() {
        return id;
   }
   string getName() {
        return name;
   }
   string getGender() {
        return gender;
   }
   string getUniversity() {
        return university;
   }
   string getCareer() {
        return career;
   }

};
```

Encapsulamiento

```
class UniversityStudent:

    def __init__(self,id,name,gender,university,career,numsubjects):
        self.__id = id
        self.__name = name
        self.__gender = gender
        self.__university = university
        self.__career = career
        self.__numsubjects = numsubjects

    def inscribeSubjects(self):
        pass

    def cancelSubjects(self):
        pass

    def consultRatings(self):
        pass
```

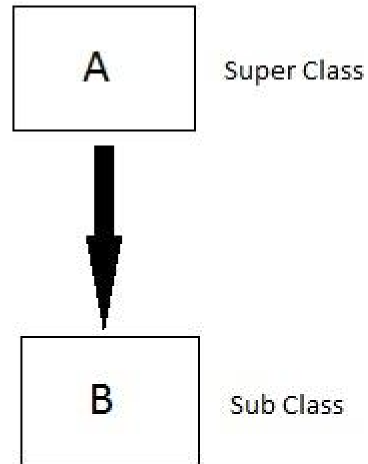
```
def setNumSubjects(self,numSubjects):
    if numSubjects < 0 or numSubjects > 10:
        print "Numero invalido de materias"
    else:
        self.__numsubjects = numSubjects

def getNumSubjects(self):
    return self.__numsubjects

def getId(self):
    return self.__id

def getName(self):
    return self.__name

def getGender(self):
    return self.__gender

def getUniversity(self):
    return self.__university;

def getCareer(self):
    return self.__career
```

Herencia

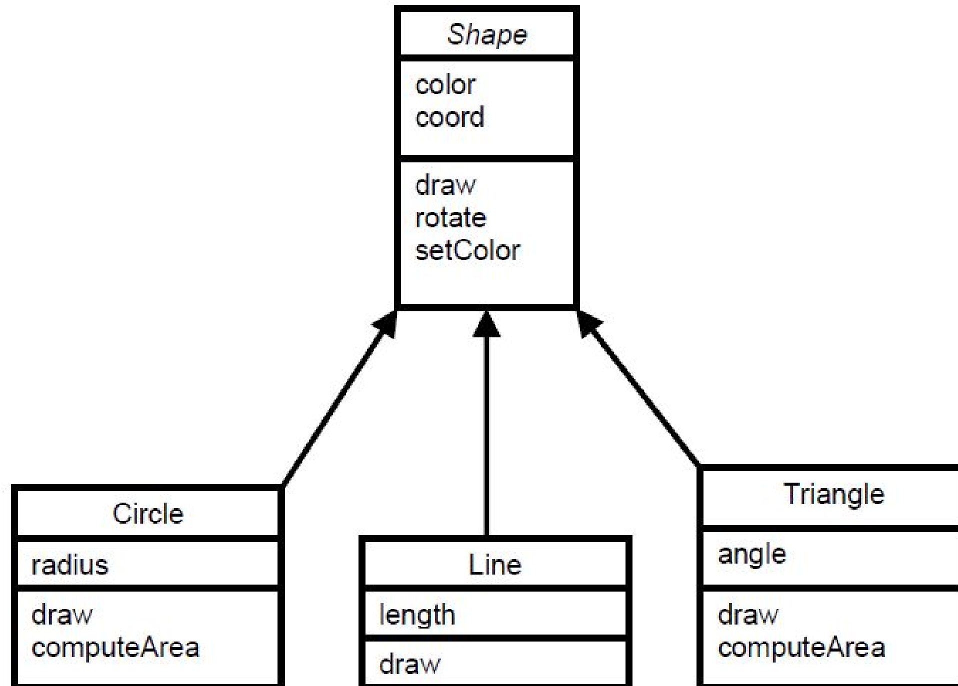
En programación orientada a objetos, la herencia es cuando un objeto o clase es creado basado en otra clase (herencia basada en clases) u objeto (herencia basada en prototipos), utilizando la misma implementación o especificando una nueva implementación.

Herencia

Simple: Es cuando una clase hereda de sólo una clase base.

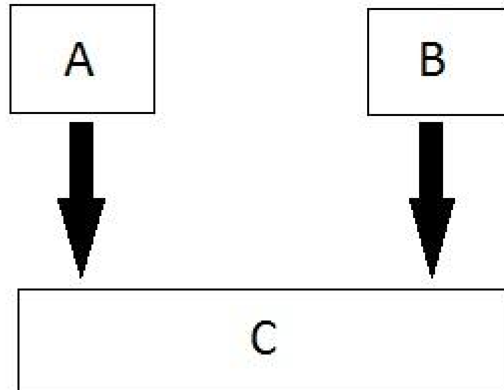


Herencia

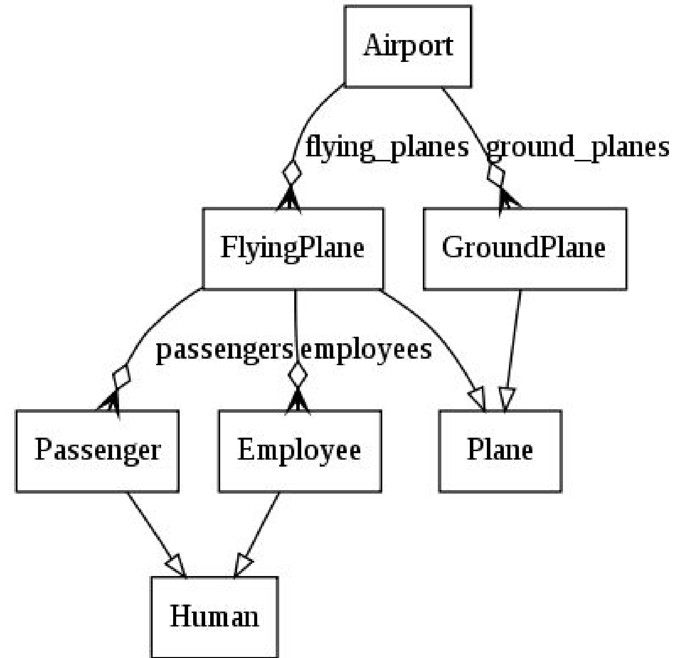


Herencia

Múltiple: Es cuando una clase hereda de dos o más clases base.



Herencia

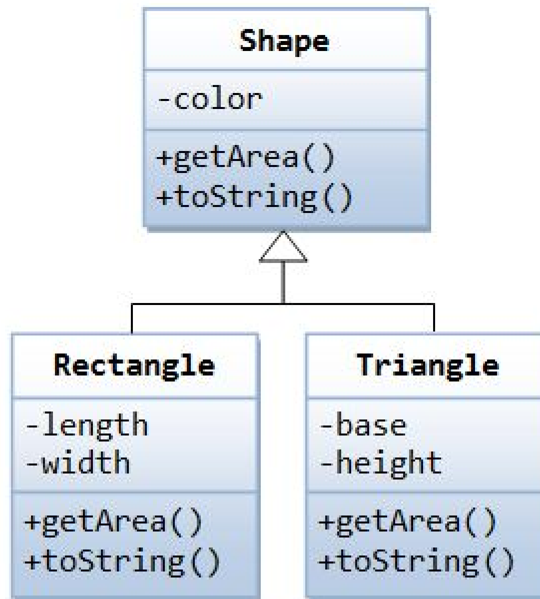


Polimorfismo

Polimorfismo es una propiedad que permite procesar objetos del mismo tipo de manera diferente.

Tipos de polimorfismo:
- Polimorfismo Ad Hoc
- Polimorfismo paramétrico
- Polimorfismo de subtipos

Polimorfismo



Polimorfismo

Polimorfismo Ad Hoc: Se refiere a funciones que cambian su comportamiento dependiendo del tipo de argumentos que reciben (sobrecarga de funciones).

Polimorfismo

Polimorfismo paramétrico: El polimorfismo paramétrico permite que las funciones y las clases puedan escribirse de forma genérica, de tal manera que pueda manipular los datos de la misma manera sin importar el tipo.

Ejemplos:

C++ - Templates

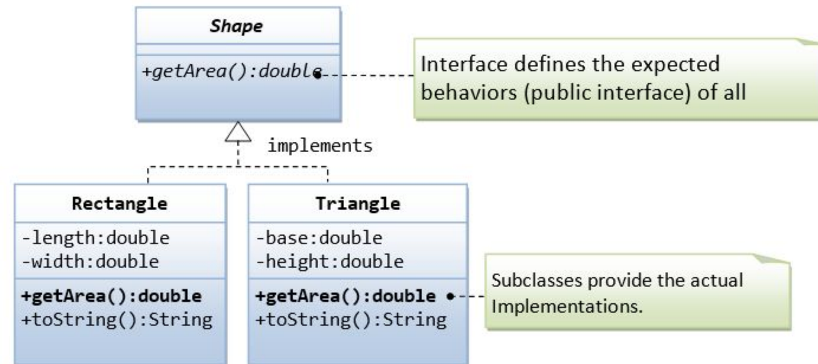
Java - Generics

Polimorfismo

Polimorfismo de subtipos: Es una forma de polimorfismo en la cual los subtipos de un tipo pueden sustituir el comportamiento de las funciones del supertipo con su propia implementación.

Interfaz

Una interfaz es una descripción de las acciones que un objeto puede hacer. En programación orientada a objetos una interfaz "X" describe todas las funciones que un objeto debe tener para poder ser un "X".



Ventajas

Mejora la productividad del software
Mejora la mantenibilidad del software
Desarrollo más rápido
Reduce el costo de desarrollo
Mejora la calidad del software

Desventajas

Curva de aprendizaje
El tamaño del programa es más grande
Lentitud

Lenguajes de Programación

Java
C++
C#
Python
Ruby

Aplicaciones

- Bases de datos orientadas a objetos.
- Sistemas de tiempo real.
- Modelamiento y simulación de agentes.
- Inteligencia artificial.

Referencias

- Conceptos basicos de POO
  https://docs.oracle.com/javase/tutorial/java/concepts/
- Principios SOLID
  https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- Fundamentos de la POO
  https://msdn.microsoft.com/es-co/library/bb972232.aspx
- Libro de POO
  https://unefazuliasistemas.files.wordpress.com/2011/04/programacion-orientada-a-objetos-luis-joyanes-aguilar.pdf
- Principios SOLID
  http://es.slideshare.net/JuanjoFuchs/solid-cmo-lo-aplico-a-mi-cdigo