

# PARADIGMAS - PARCIAL II

Lenguajes orientados a objetos

## Índice:

• Resumen - - - - -	2
- Introducción y propiedades generales - - - - -	2
- Simula - - - - -	5
- Smalltalk - - - - -	7
- Java - - - - -	10
- Otros conceptos importantes - - -	12
• Práctica - - - - -	13
- Ejercicios del libro - - - - -	13
- Parcial 14/05/24 - - - - -	27
- Recuperatorio 18/06/24 - - - - -	30
- Ejercicios sueltos de parciales (3/05/22; 23/06/23; 9/08/23) - - - - -	34

# Resumen

El objetivo de los lenguajes orientados a objetos es el desarrollo de programas modulares, para ello estos lenguajes brindan ciertas herramientas, todas giran en torno a la abstracción.

## modularidad

↳ componente: Unidad de programa con sentido

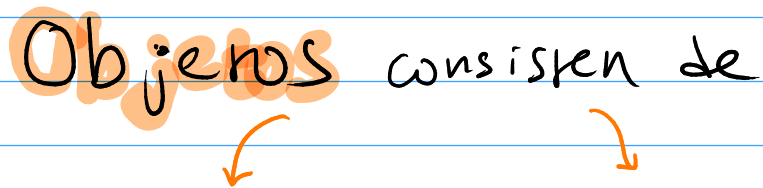
↳ interfaz: tipos y operaciones definidos dentro de un componente qd son visibles fuera de él.

↳ especificación: comportamiento esperado de un componente

↳ implementación: estructura de datos y funciones dentro del componente

## propiedades de la orientación a objetos

**Objetos** consisten de



```
graph TD; A[Objetos] --> B[datos ocultos]; A --> C[operaciones públicas];
```

**datos ocultos**  
variables de instancia  
funciones ocultas

**operaciones públicas**  
métodos  
variables públicas

Se envían mensajes a los objetos y ellos los manejan

## **Look up dinámico**

El código a ejecutar dependerá del objeto y del mensaje.  
Se resuelve en tiempo de ejecución.

## **Encapsulación**

Separa la vista del programador de la del usuario.

El código del cliente opera con un conjunto fijo de operaciones.

## Subtipado y Herencia

Relación entre interfaces  
Si la interfaz A (mensaje qe entiende el objeto) contiene todos los elementos de la interfaz B  $\Rightarrow$  obj de tipo A pueden usarse como obj de tipo B

Organiza datos semejantes en clases relacionadas

Relación entre implementaciones  
Nuevos objetos se pueden definir reusando implementaciones de otros obj.  
Evita reimplementar funciones ya definidas

## Estructura de un programa OO

Agrupar datos y funciones en Clases

Definen el comportamiento de todos los objetos qe son instancia de esa clase

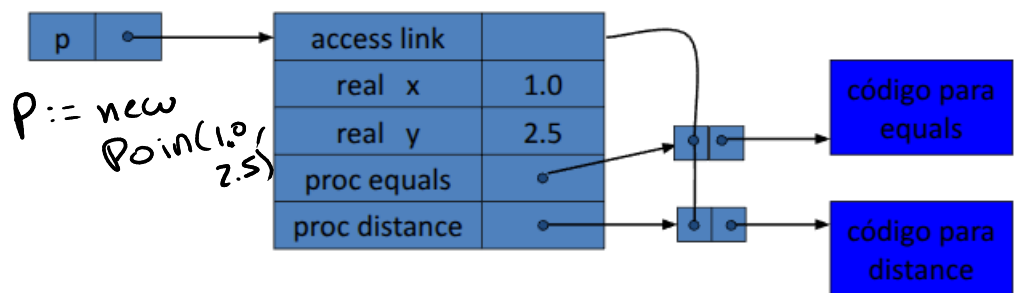
# Simula (lenguaje orientado a objetos)

**Objetos** → **clase**: proc que devuelve un puntero al activation record en el que se ejecuta

→ **objeto**: activation record que se genera al llamar a una clase.

→ **garbage collector**

Class Point (x, y); real x, y



Un objeto se representa con un activation record con un access link para encontrar las variables globales con alcance estático

## Subtizado

Mediante jerarquía de clases  
(el tipo de una subclase se trata como subtipo del tipo asociado con la superclase)

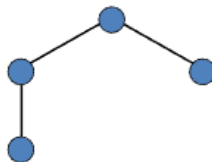
- class A(...); ...
- A class B(...); ...
- ref (A) a :- new A(...)
- ref (B) b :- new B(...)
- a := b /\* legal porque B es una subclase de A \*/
- b := a /\* también legal, pero hay que comprobarlo en tiempo de ejecución \*/

## Herencia

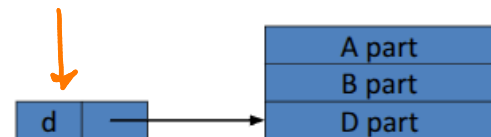
Por prefijado de clases.

Un objeto de una clase prefijada es la concatenación de objetos de C/ clase del prefijo

class A  
A class B  
A class C  
B class D



ref(D) d :- new D()



Simula NO tener encapsulación

# Smalltalk

Todo es un objeto (hasta una clase).

Todas las operaciones son mensajes a objetos.

## Terminología

- Objeto: instancia de clase
- Clase: define el comportamiento de sus objetos
- Subclase: clase con modificaciones a una superclase
- Selector: nombre de un mensaje
- Mensaje: selector con valores para sus parámetros
- Método: código que usa una clase para responder a un mensaje
- Variable de instancia: datos guardados en un objeto

## Encapsulación

Métodos públicos y variables de instancia o *slots* (métodos de subclases pueden manipularlas)

# Herencia

Subclases, self, super

## Subnigado

Implícito, el sistema de tipos estático

C++

## Objetos

Creados por clases Contienen datos del miembro y un puntero a la clase


tabla de funciones virtuales

- Se acceden a través de un puntero en el objeto.
- Se pueden redefinir en subclases derivadas
- Se declaran explícitamente o se heredan como virtuales
- Se pagan overhead solo si se usan.



# Herencia

Múltiple. Clases base públicas y privadas.  
Puede causar **name clashes**

usar un  calificador  
de alcance para explicar de qué clase  
usar el método. (A::f, usa f de A)

## Subtizado

A es subtipo de B si la clase A tiene  
como clase pública a B.

```
class parent { public:  
    void printclass() {printf("p ");};  
    virtual void printvirtual() {printf("p ");};  
};  
class child : public parent { public:  
    void printclass() {printf("c ");};  
    virtual void printvirtual() {printf("c ");};  
};
```

## Encapsulación

Un miembro se puede declarar

**público:** Visible en todos lados

**privado:** Visible solamente en

la clase donde se declara

**protegido:** en declaraciones de  
clase y sus subclases

Se puede forzar la inicialización de  
los obj.

## Otros

- No hay garbage collector
- Hay **clases abstractas**: clases sin implementación completa (=0)  
Sirven q/ construir jerarquías de tipos, ya qe pueden tener clases derivadas.

## JAVA

### Objetos

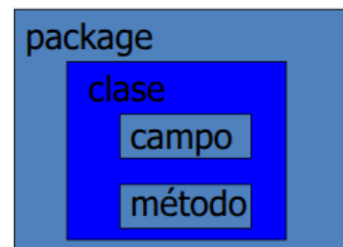
Tienen campos (datos) y métodos (func.)  
Miembros estáticos son miembros de la clase.

Se alojan en el heap (no stack).  
Garbage collector con finalize (para locks y archivos)  
Garantiza llamadas al constructor para cada objeto.

Son instancias de una clase.

### Encapsulación

Cada clase es parte de un package (conjunto de clases en un mismo namespace)



4 distinciones de visibilidad:  
public, private, protected, package  
Hay miembros estáticos en las  
clases (les el MISMO para todas las  
instancias de clase)

## Herencia

Simple, pero hay interfaces

Componentes que pueden incluirse  
en las clases, contienen firmas de  
funciones no implementadas.

Miembros con la palabra clave  
final (variables no pueden cambiarse;  
métodos no pueden modificarse;  
clases no pueden heredarse)

Preserva la llamada al constructor.  
Constructor de la subclase tiene  
que llamar al de la superclase  
con los parámetros adecuados.

## Tipos

Primitivos (no objetos) y de  
referencia (clases, arrays, interfaces).  
Chequeo estático de tipos.

# Subtipado

Por herencia e interfaces

↓  
pueden ser implementados por varias clases, teniendo así múltiples subtipos.

Otros conceptos importantes:

- + **name clash**: cuando dos implementaciones distintas tienen igual nombre y firma

- + **expresividad**: capacidad que tiene el lenguaje de hacer cosas de manera directa/clara.

- + **flexibilidad**: capacidad que tiene el lenguaje de hacer una misma cosa pero de distintas maneras.

- + **Static variables**: una sola copia de (class variables) la variable es compartida entre todos los objetos de la clase.

- + **Non-static variables**: cada vez que se (instance variables) instancia una clase se crea una copia nueva de la variable. Se destruye cdo el objeto se destruye.

7.6. Los **mixins** son una construcción de Ruby que permite incorporar algunas de las funcionalidades de la herencia múltiple, ya que Ruby es un lenguaje con herencia simple. Con un **mixin** se pueden incluir en una clase miembros de otra clase, con

la palabra clave **include**. Los *name clashes*, si los hay, se resuelven por el orden de los **include**, de forma que la última clase añadida prevalece, y sus definiciones son las que se imponen en el caso de conflicto. Teniendo esto en cuenta, describa el comportamiento del siguiente pedazo de código.

```
1 module EmailReporter
2   def send_report
3     # Send an email
4   end
5 end
6
7 module PDFReporter
8   def send_report
9     # Write a PDF file
10  end
11 end
12
13 class Person
14 end
15
16 class Employee < Person
17   include EmailReporter → send_report(email)
18   include PDFReporter → " " (PDF)
19 end
20
21 class Vehicle
22 end
23
24 class Car < Vehicle
25   include PDFReporter → " " (PDF)
26   include EmailReporter → " " (email)
27 end
```

Cuando haga `Employee.send_report()`  
se va a escribir un pdf  
y cuando haga `Car.send_report()`  
se va a enviar un email.

7.1. En las siguientes declaraciones de clase, indique qué partes son implementación y qué partes son interfaz, marcando la interfaz.

```
public class URLExpSimple {  
    public static void main(String[] args) {  
        try {  
            URL mySite = new  
            URL("http://www.cs.utexas.edu/~scottm");  
            URLConnection yc = mySite.openConnection();  
            Scanner in = new Scanner(new  
            InputStreamReader(yc.getInputStream()));  
            int count = 0;  
            while (in.hasNext()) {  
                System.out.println(in.next());  
                count++;  
            }  
            System.out.println("Number of tokens: " +  
            count);  
            in.close();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
public class Stopwatch  
{  
    private long startTime;  
    private long stopTime;  
  
    public static final double NANOS_PER_SEC =  
    1000000000.0;  
  
    public void start()  
    { startTime = System.nanoTime(); }  
  
    public void stop()  
    { stopTime = System.nanoTime(); }  
  
    public double time()  
    { return (stopTime - startTime) / NANOS_PER_SEC; }  
  
    public String toString(){  
        return "elapsed time: " + time() + " seconds.";  
    }  
  
    public long timeInNanoseconds()  
    { return (stopTime - startTime); }  
}
```

```

1 public class Minesweeper
2 { private int[][] myTruth;
3   private boolean[][] myShow;

5   public void cellPicked(int row, int col)
6   { if( inBounds(row, col) && !myShow[row][col] )
7     { myShow[row][col] = true;

9       if( myTruth[row][col] == 0)
10      { for(int r = -1; r <= 1; r++)
11        for(int c = -1; c <= 1; c++)
12          cellPicked(row + r, col + c);
13      }
14    }
15  }

17  public boolean inBounds(int row, int col)

18  { return 0 <= row && row < myTruth.length && 0 <= col
19    && col < myTruth[0].length;
20  }
21 }

```

```

1 public class PrimeEx {
2
3     public static void main(String[] args) {
4         printTest(10, 4);
5         printTest(2, 2);
6         printTest(54161329, 4);
7         printTest(1882341361, 2);
8         printTest(36, 9);
9
10        System.out.println(isPrime(54161329) + " expect false");
11        System.out.println(isPrime(1882341361) + " expect
12        true");
13        System.out.println(isPrime(2) + " expect true");
14        int numPrimes = 0;
15        Stopwatch s = new Stopwatch();
16        s.start();
17        for(int i = 2; i < 10000000; i++) {
18            if(isPrime(i)) {
19                numPrimes++;
20            }
21        }
22        s.stop();
23        System.out.println(numPrimes + " " + s);
24        s.start();
25        boolean[] primes = getPrimes(10000000);
26        int np = 0;
27        for(boolean b : primes)
28            if(b)
29                np++;
30        s.stop();
31        System.out.println(np + " " + s);
32
33        System.out.println(new BigInteger(1024, 10, new
34        Random()));
35    }
36
37    public static boolean[] getPrimes(int max) {
38        boolean[] result = new boolean[max + 1];
39
40        for(int i = 2; i < result.length; i++)
41            result[i] = true;
42        final double LIMIT = Math.sqrt(max);
43        for(int i = 2; i <= LIMIT; i++) {
44            if(result[i]) {
45                // cross out all multiples;
46                int index = 2 * i;
47                while(index < result.length){
48                    result[index] = false;
49                    index += i;
50                }
51            }
52        }
53        return result;
54    }
55 }

```



```

54 public static void printTest(int num, int
    expectedFactors) {
55     Stopwatch st = new Stopwatch();
56     st.start();
    int actualFactors = numFactors(num);
58     st.stop();
    System.out.println("Testing " + num + " expect " +
        expectedFactors + ", " +
60         "actual " + actualFactors);
    if(actualFactors == expectedFactors)
62         System.out.println("PASSED");
    else
64         System.out.println("FAILED");
    System.out.println(st.time());
66 }

68 // pre: num >= 2
public static boolean isPrime(int num) {
70     assert num >= 2 : "failed precondition. num must be >=
        2. num: " + num;
    final double LIMIT = Math.sqrt(num);
72     boolean isPrime = (num == 2) ? true : num % 2 != 0;
    int div = 3;
74     while(div <= LIMIT && isPrime) {
        isPrime = num % div != 0;
76         div += 2;
    }
78     return isPrime;
}

80 // pre: num >= 2
public static int numFactors(int num) {
82     assert num >= 2 : "failed precondition. num must be >=
        2. num: " + num;
    int result = 0;
84     final double SQRT = Math.sqrt(num);
    for(int i = 1; i < SQRT; i++) {
86         if(num % i == 0) {
88             result += 2;
        }
    }
90     if(num % SQRT == 0)
        result++;
92     return result;
94 }

```

7.2. Dibuje la jerarquía de clases en que se basa el siguiente código.

```
class Vehicle {
public:
    explicit
    Vehicle( int topSpeed )
        : m_topSpeed( topSpeed )
    {}
    int TopSpeed() const {
        return m_topSpeed;
    }

    virtual void Save( std::ostream& ) const = 0;

private:
    int m_topSpeed;
};

class WheeledLandVehicle : public Vehicle {
public:
    WheeledLandVehicle( int topSpeed, int numberOfWheels
    )
        : Vehicle( topSpeed ), m_numberOfWheels(

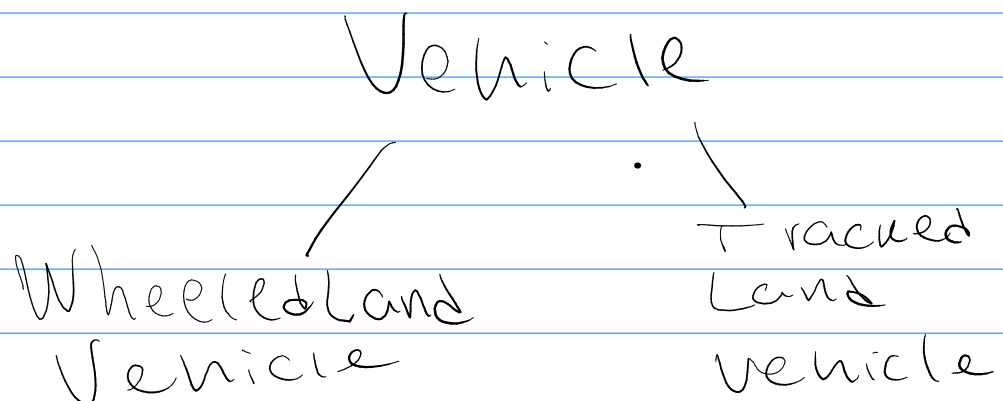
        numberOfWheels )
    {}
    int NumberOfWheels() const {
        return m_numberOfWheels;
    }

    void Save( std::ostream& ) const; // is implicitly
    virtual

private:
    int m_numberOfWheels;
};

class TrackedLandVehicle : public Vehicle {
public:
    TrackedLandVehicle ( int topSpeed, int numberOfTracks
    )
        : Vehicle( topSpeed ), m_numberOfTracks (
        numberOfTracks )
    {}
    int NumberOfTracks() const {
        return m_numberOfTracks;
    }
    void Save( std::ostream& ) const; // is implicitly
    virtual

private:
    int m_numberOfTracks;
};
```



```

class DrawableObject
{
public:
    virtual void Draw(GraphicalDrawingBoard&) const = 0;
    //draw to GraphicalDrawingBoard
};

class Triangle : public DrawableObject
{
public:
    void Draw(GraphicalDrawingBoard&) const; //draw a
    triangle
};

class Rectangle : public DrawableObject
{
public:
    void Draw(GraphicalDrawingBoard&) const; //draw a
    rectangle
};

class Circle : public DrawableObject
{
public:
    void Draw(GraphicalDrawingBoard&) const; //draw a
    circle
};

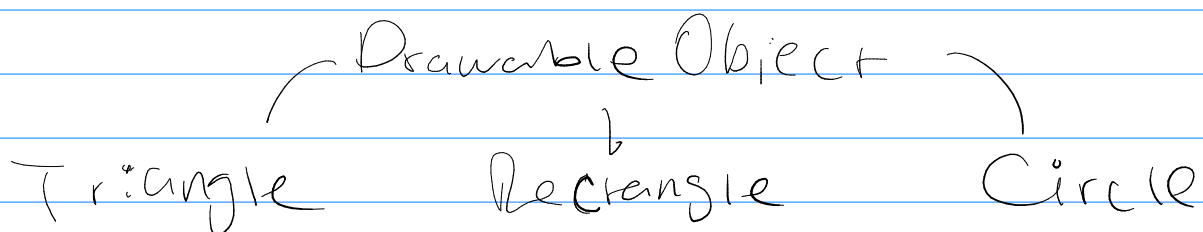
typedef std::list<DrawableObject*> DrawableList;

DrawableList drawableList;
GraphicalDrawingBoard drawingBoard;

drawableList.pushback(new Triangle());
drawableList.pushback(new Rectangle());
drawableList.pushback(new Circle());

for(DrawableList::const_iterator iter =
    drawableList.begin(),
    endIter = drawableList.end();
    iter != endIter;
    ++iter)
{
    DrawableObject *object = *iter;
    object->Draw(drawingBoard);
}

```



7.3. Identifique en el siguiente código en C++ un problema con la herencia del miembro meow.

```
1 class Felino {  
2 public:  
3 void meow() = 0; abstracta?  
4 };  
5  
6 class Gato : public Felino {  
7  
8 public:  
9 void meow() { std::cout << "miau\n"; }  
10 };  
11  
12 class Tigre : public Felino {  
13 public:  
14 void meow() { std::cout << "ROARRRRRR\n"; }  
15 };  
16  
17 class Ocelote : public Felino {  
18 public:  
19 void meow() { std::cout << "roarrrrr\n"; }  
20 };
```

tengo q  
agregarle  
el virtual  
para indicar  
q PUEDO  
redefinir  
en las  
subclases

ABSTRACTA (=0) => TENGO q  
redefinir  
(no hay implementación)

VIRTUAL(virtual) => PUEDO  
redefinir  
(si hay implementación)

7.4. A partir del siguiente código Ruby hemos tratado de escribir un código Java con la misma semántica, pero los resultados no son iguales. Cuál es la diferencia y por qué? Cómo deberíamos modificar el programa en Java para que haga lo mismo que el programa en Ruby?

```
1 #!/usr/bin/ruby
3 class Being
4     @@count = 0
5
6     def initialize
7         @@count += 1
8         puts "creamos un ser"
9     end
10
11     def show_count
12         "Hay #{@count} seres"
13     end
14 end
15
16 class Human < Being
17     def initialize
18         super
19         puts "creamos un humano"
20     end
21 end
22
23 class Animal < Being
24     def initialize
25         super
26         puts "creamos un animal"
27     end
28 end
29
30 class Dog < Animal
31     def initialize
32         super
33         puts "creamos un perro"
34     end
35 end
36
37 Human.new
38 d = Dog.new
39 puts d.show_count
```

Siempre que se use la variable count, nos referimos a LA MISMA variable, es decir es GLOBAL

3?

```

class Being {
    private int count = 0; => cada instancia
                           tiene su count
                           inicial en cero!!
    public Being() {
        count++;
        System.out.println("creamos un ser");
    }

    public void getCount() {
        System.out.format("hay %d seres%n", count);
    }
}

class Human extends Being {

    public Human() {
        System.out.println("creamos un humano");
    }
}

class Animal extends Being {

    public Animal() {

        System.out.println("creamos un animal");
    }
}

class Dog extends Animal {

    public Dog() {
        System.out.println("creamos un perro");
    }
}

public class Inheritance2 {

    @SuppressWarnings("ResultOfObjectAllocationIgnored")
    public static void main(String[] args) {
        new Human();
        Dog dog = new Dog();
        dog.getCount();
    }
}

```

Static indica qe la variable se comparte para TODAS las instancias de la clase

7.5. A partir de la siguiente **template** en C++, escriba una clase de C++ con la misma semántica pero específica para `int`.

```
1 template <class A_Type> class calc
2 {
3     public:
4         A_Type multiply(A_Type x, A_Type y);
5         A_Type add(A_Type x, A_Type y);
6 };
7 template <class A_Type> A_Type
8     calc<A_Type>::multiply(A_Type x, A_Type y)
9 {
10     return x*y;
11 }
12 template <class A_Type> A_Type calc<A_Type>::add(A_Type
13     x, A_Type y)
14 {
15     return x+y;
16 }
```

Class calc

public:

int multiply (int x, int y);  
int add (int x, int y);

int calc::multiply (int x, int y)  
return x \* y

int calc::add (int x, int y)  
return x + y

7.8. El siguiente ejemplo en C++ causa un error de compilación. Por qué? cómo puede solucionarse?

```
1 class trabajador
2 {
3     public:
4         void hora_de_levantarse( )
5         { .... }
6 };
7 class estudiante
8 {
9     void hora_de_levantarse( )
10    { .... }
11 };
12 class ayudante_alumno : public trabajador, public
13    estudiante
14 {
15 };
16
17 int main()
18 {
19     ayudante_alumno obj;
20
21     obj.hora_de_levantarse( )
22 }
```

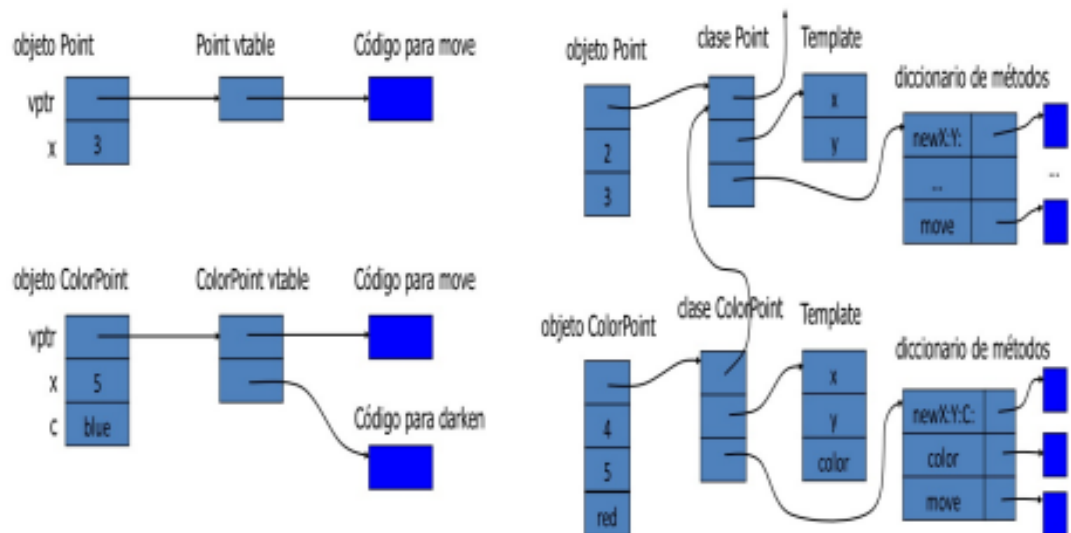
Hay un error de compilación  
ya que hay una sobrecarga  
en la función  
hora-de-levantarse.

ambos ← Al compilar, cuando se usa  
esa función del objeto  
ayudante\_alumno, como hereda  
de trabajador y estudiante,  
ya son objetos que tienen dicha  
función, el compilador no  
sabe bien cuál compilar.

Solución: hacer las funciones  
virtuales! y redefinirlas en una  
o usar :: para indicar de  
cuál clase usar el objeto



- 7.10. Explique por qué Smalltalk es menos eficiente que C++ utilizando el concepto de *overhead*. Puede ayudarse de los siguientes gráficos, en los que se muestra el proceso de lookup de información asociada a los objetos en uno y otro lenguaje.



Smalltalk es menos eficiente que C++ porque SIEMPRE se paga overhead al usar clases heredadas, ya que las subclases tienen siempre un puntero apuntando a su clase madre, en cambio en C++ se agrega un nivel de indirección extra únicamente si la clase madre tiene una función virtual, ya que las clases con funciones virtuales tendrán un puntero a la vtable, que contiene los códigos de dichas funciones virtuales.

7.11. En el siguiente código el compilador está creando una instancia de una subclase anónima de la clase abstracta, y luego se está invocando al método de la clase abstracta. Explique por qué es necesario que el compilador haga este proceso.

```
1 abstract class my {  
2     public void mymethod() {  
3         System.out.print("Abstract");  
4     }  
5 }  
6  
7 class poly {  
8     public static void main(String a[]) {  
9         my m = new my() {};  
10        m.mymethod();  
11    }  
12 }
```

Las clases abstractas no se pueden instanciar directamente, necesitamos si o si una subclase que herede de ella para invocar sus métodos. Con `my m = new my()` estamos creando una subclase `my` que hereda de la clase abstracta `my` y luego usamos la instancia de esa subclase para llamar al método de la clase abstracta. Como este ya está implementado, se usa ese código.

Si este no estuviera implementado, deberíamos implementarlo entre esas llaves de la subclase anónima.

# Paradigmas de la Programación – Segundo Parcial

14 de Mayo de 2024

1. [10 pt.] La recolección de basura es una rutina residente, que se ejecuta en background, sin necesidad de que intervenga explícitamente el programador. Su objetivo es disponibilizar memoria que había sido ocupada por un programa para almacenar información, pero que el programa ya no va a requerir en el resto de su ejecución.

Teniendo en cuenta que los objetos se almacenan en el *heap*, explique por qué la recolección de basura (o, en el caso de C++, la liberación manual de la memoria) tiene más impacto en eficiencia en los lenguajes con orientación a objetos que en otros lenguajes.

Eficiencia: mejor uso de los recursos computacionales (cómputo, memoria, tiempo de memoria)

En los lenguajes orientados a objetos el heap guarda muchas más cosas, ya que estos lenguajes utilizan, obviamente, casi en su totalidad, objetos, por lo que la recolección de basura aumenta mucho la eficiencia.

2. Indique en cuáles de las siguientes configuraciones se pueden dar *name clashes* (colisiones de nombre) en un lenguaje de programación:

- a) [2 pt.] si incorpora implementaciones de otras componentes (como los *traits* de Scala, como los *mixins* de Ruby) sin políticas de precedencia
- b) [2 pt.] si incorpora implementaciones de otras componentes (como los *traits* de Scala, como los *mixins* de Ruby) con políticas de precedencia
- c) [2 pt.] si tiene herencia múltiple con linearización
- d) [2 pt.] si tiene herencia múltiple sin linearización
- e) [2 pt.] si tiene interfaces como las de Java

Como no tienen implementación, no hay name clashes (si tienen la misma firma)

3. [10 pt.] Lea la siguiente explicación sobre **override**:

*The **override** keyword is redundant in the sense that it reiterates information that is already present without it. You cannot use the override specifier to make a function override another function that it would not otherwise override. Nor does omitting the override specifier ever cause overriding not to occur.*

*Redundancy is not necessarily a bad thing. It protects against errors. This is well known in the case of data storage. The override specifier ensures that you don't accidentally hide a base class member function where you intend to override it.*

Sabemos que el siguiente código es correcto:

```
1 class Base
2 {
3 public:
4     virtual void f()
5     {
6         std::cout << "Base";
7     }
8 };
9
10 class Derived : public Base
11 {
12 public:
13     void f() override
14     {
15         std::cout << "Derivada";
16     }
17 };
```

¿Cuál de las dos variantes que siguen no es equivalente al código anterior, y por qué? ¿Alguna de estas variantes produce un error de compilación? ¿Qué diferencias de comportamiento pueden llegar a ocasionar?

Código A:

```
1 class Base
2 {
3 public:
4     virtual void f()
5     {
6         std::cout << "Base";
7     }
8 };
9
10 class Derived : public Base
11 {
12 public:
13     void f()
14     {
15         std::cout << "Derivada";
16     }
17 };
```

Código B:

```
1 class Base
2 {
3 public:
4     virtual void f()
5     {
6         std::cout << "Base";
7     }
8 };
9
10 class Derived : public Base
11 {
12 public:
13     virtual void f()
14     {
15         std::cout << "Derivada";
16     }
17 };
```

*no es equivalente*

El código B no es equivalente porque la función virtual que hereda la clase derivada VUELVE a ser virtual, de manera que si llega a haber una clase derivada que herede de la primer clase derivada y de la clase base y NO redefiniera f() habría un name clash y por lo tanto un error de compilación, mientras que, si continuáramos el código A, una clase derivada de la primer clase derivada y de la clase base directamente NO podría redefinir f().

4. [10 pt.] Complete el siguiente texto con las palabras "Inheritance", "Subtyping", "is a subtype of" y "inherits from another type":

Subtyping refers to compatibility of interfaces. A type B is a subtype of A if every function that can be invoked on an object of type A can also be invoked on an object of type B.

inheritance refers to reuse of implementations. A type B inherits from A if some functions for B are written in terms of functions of A.

En C++ y Java, cuando una clase hereda de otra, la subclase es subtipo de la superclase (pero no a la inversa)

# Paradigmas de la Programación – Recuperatorio del Segundo Parcial

18 de Junio de 2024

1. [10 pt.] Estos dos códigos, en PHP y en Ruby respectivamente, parecen muy semejantes pero se comportan de forma distinta:

PHP:

```
1 class ClassA {
2     public $myvar;
3
4     public function __construct() {
5         $this->myvar = "hello";
6     }
7
8     public function getMyVar() {
9         echo $this->myvar;
10    }
11 }
12
13 class ClassB extends ClassA {
14     public function __construct() {
15         $this->myvar = "goodbye";
16     }
17 }
18
19 $demo1 = new ClassA();
20 $demo2 = new ClassB();
21
22 $demo1->getMyVar(); → hello
23 $demo2->getMyVar(); → goodbye
24 $demo1->getMyVar(); → hello
```

*Handwritten note: myvar = hello (with arrow pointing to line 19)*

Ruby:

```
1 class ClassA
2     @@my_var = nil
3     public en Ruby
4     def initialize
5         @@my_var = "hello"
6     end
7
8     def my_var
9         puts @@my_var
10    end
11 end
12
13 class ClassB < ClassA
14     def initialize
15         @@my_var = "goodbye"
16     end
17 end
18
19 demo1 = ClassA.new
20 demo1.my_var → hello
21 demo2 = ClassB.new
22
23 demo2.my_var → goodbye
24 demo1.my_var → goodbye
```

El código en PHP imprime "hello goodbye hello". El código en Ruby, en cambio, imprime: "hello goodbye goodbye". Según este comportamiento observable, de una descripción sobre el alcance y comportamiento de la variable *myvar* en cada uno de los dos programas. Si le resulta más cómodo, puede tener en cuenta que se trata de un constructor en PHP y de una variable de clase en Ruby.

En PHP la variable *myvar* al ser un constructor se instancia particularmente para cada clase, ya sea clase madre o clase hija, se comporta como una variable de INSTANCIA. En cambio, en Ruby, como es una variable de clase, será ESTÁTICA para todo el programa, es decir, la variable *myvar* es la MISMA para todas las instancias de clase que la utilicen, por eso cuando inicializo *demo2* también se modifica *myvar* en *demo1*.



2. [10 pt.] El siguiente texto describe el comportamiento de la instrucción *final* en Java:

When you declare a variable as **final**, it must be **initialized only once**. Once a final variable has been assigned a value, attempting to **modify it will result in a compile-time error**. This immutable property makes final variables akin to constants.

**Final methods** are methods that **cannot be overridden in any subclass**. This is particularly useful when you need to maintain a consistent implementation of a method across various subclasses in the class hierarchy, thus avoiding unintended behaviors or security breaches.

A **final class** is one that **cannot be subclassed**. This restriction is typically employed to maintain the immutability of the class or to provide a guarantee that certain behavior is preserved without alteration through inheritance. Final classes are often used in conjunction with final variables to create fully immutable objects which are thread-safe by design.

Según estas definiciones, argumente cuáles de los siguientes códigos no compilarían y por qué.

Código A:

```
1 public class Animal {
2     public final void makeSound() {
3         System.out.println("The animal makes a sound");
4     }
5 }
6
7 public class Dog extends Animal {
8     @Override
9     public void makeSound() {
10        System.out.println("The dog barks");
11    }
12 }
```

Código B:

```
1 public final class A {
2     private final int value;
3
4     public A(int value) {
5         this.value = value;
6     }
7
8     public int getValue() {
9         return value;
10    }
11 }
12
13 public class B extends A { }
```

Código C:

```
1 public class A {
2     public static void main() {
3         final int foo = 10;
4         foo = 20;
5     }
6 }
```

Código A no compila pues se está sobreescribiendo un final method, y esto no puede hacerse. El código B tampoco compila ya que una clase busca heredar una clase final, cuando estas no pueden tener subclases. El código C tampoco compila ya que hay una variable final que se está sobreescribiendo.

3. [10 pt.] Teniendo en cuenta el alcance de los diferentes niveles de visibilidad en la orientación a objetos, explique qué semántica es necesario que tenga la palabra clave *friend* de la línea 9 para que el siguiente código en C++ compile sin problemas.

```
1 class ClassB; // Forward declaration of ClassB
2
3 class ClassA {
4     private:
5         int numA;
6     public:
7         ClassA() : numA(0) {}
8
9         friend class ClassB; // Friend class declaration
10
11        void display() {
12            cout << "ClassA:_" << numA << endl;
13        }
14 };
15
16 class ClassB {
17     private:
18         int numB;
19     public:
20         ClassB() : numB(0) {}
21
22        void setValue(ClassA& objA, int value) {
23            objA.numA = value; // Accessing private member of ClassA
24        }
25
26        void display(ClassA& objA) {
27            cout << "ClassB:_" << numB << endl;
28            objA.display(); // Accessing ClassA's member function
29        }
30 };
31
32 int main() {
33     ClassA objA;
34     ClassB objB;
35
36     objB.setValue(objA, 100);
37     objB.display(objA);
38
39     return 0;
40 }
```

trata de setear  
una variable  
privada de  
A

La palabra clave *friend* debería permitir, a la clase que se declara con esa palabra clave, acceder a los campos y miembros privados de la clase en la que se está declarando. Si esto no fuera así, cuando la clase B trata de modificar el campo privado *numA* de la clase A, saltaría un error de compilación.



4. [10 pt.] Explique por qué el siguiente código en Java no genera un *name clash*.

```
1 public interface Interfacel {
2
3     // create a method
4     public void show();
5 }
6
7 public interface Interface2 {
8
9     // create a method
10    public void show();
11 }
12
13 public class Casel implements Interfacel , Interface2 {
14     // implement the methods of interface
15     public void show()
16     {
17         System.out.println("Geeks_For_Geeks");
18     }
19     public static void main(String[] args)
20     {
21         // create object
22         Casel obj = new Casel();
23         // using object call the implemented method
24         obj.show();
25     }
26 }
```

Este código no genera un name clash porque el método del cual se repite la firma NO tiene implementación en ninguna de las interfaces. Como el método tiene una única implementación, el compilador sabrá que es esa la que debe usar para ambas interfaces y no habrá conflicto.

# Paradigmas de la Programación – Primer Parcial

3 de Mayo de 2022

7. [20 pt.] Lea el siguiente texto y explique brevemente qué es una colisión de nombres. Atención: en clase hemos usado otro término para referirnos al mismo fenómeno. Si tenemos un lenguaje orientado a objetos, hay un contexto en el que podemos encontrar una mayor cantidad de colisiones de nombres, descríbalos.

No todos los lenguajes orientados a objetos permiten el tipo de contextos con que es más fácil que suceda una colisión de nombres. Explique algún caso de un lenguaje con orientación a objetos que limitó su expresividad para evitar esos contextos.

*An example of a naming collision*

```
1 a.cpp:
2
3 #include <iostream>
4
5 void myFcn(int x)
6 {
7     std::cout << x;
8 }
9 main.cpp:
10
11 #include <iostream>
12
13 void myFcn(int x)
14 {
15     std::cout << 2 * x;
16 }
17
18 int main()
19 {
20     return 0;
21 }
```

*When the compiler compiles this program, it will compile a.cpp and main.cpp independently, and each file will compile with no problems.*

*However, when the linker executes, it will link all the definitions in a.cpp and main.cpp together, and discover conflicting definitions for function myFcn. The linker will then abort with an error. Note that this error occurs even though myFcn is never called!*

Una colisión de nombres es un fenómeno que se da cuando en un programa tenemos dos o más componentes con la misma firma. El linker no sabe cuál de todas las implementaciones se querrá utilizar por lo que el programa dará error.

Un contexto en el que se pueden llegar a dar colisiones de nombres es cuando usamos herencia múltiple en C++. Supongamos que tenemos dos clases que implementan un método (cada una) con la misma firma y luego tenemos una clase que hereda ambas, cuando llamemos a dicho método se producirá un name clash (como lo llamamos en clase) porque no se podrá determinar a cuál de las dos implementaciones nos estamos refiriendo.

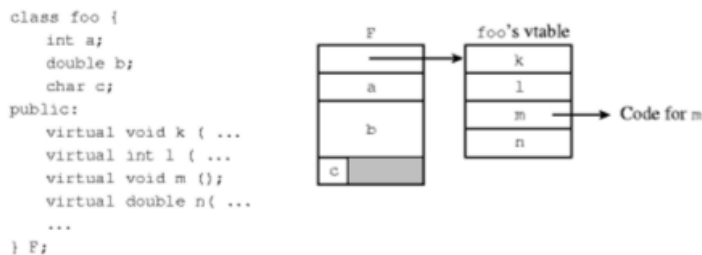
En Java no se permite esta herencia múltiple, limitando la expresividad del lenguaje pero evitando este tipo de contextos. En Java, para "sustituir" la herencia múltiple tenemos Interfaces.

# Paradigmas de la Programación

## Recuperatorio del Primer Parcial

23 de Junio de 2022

3. [10 pt.] En esta imagen se observa cómo C++ guarda los métodos virtuales en una vtable. Explique cómo es el proceso mediante el cual se usa la vtable en C++, explique la diferencia entre métodos virtuales y no virtuales en términos de *overhead* y *flexibilidad*, y qué decisión de diseño de C++ justifica que por defecto todos los métodos sean no virtuales.



En C++, usamos la vtable de una clase cuando queremos usar un método declarado con la keyword virtual. Enviamos el mensaje al objeto, este tiene, en el activation record que crea al instanciarse, todos los datos de la clase de la que es instancia + un puntero a su vtable, que contiene punteros a los códigos de los métodos virtuales de dicho objeto, por lo que si en el mensaje enviado solicitamos uno de estos métodos virtuales, se busca el puntero en la vtable que apunte a su código.

La diferencia entre un método virtual y uno no virtual es que el virtual requiere que paguemos cierto overhead, ya que para llegar a él tenemos un nivel más de indirección, mientras que los no virtuales se acceden directamente con el puntero que hace referencia al objeto en sí. Los métodos virtuales añaden flexibilidad al lenguaje ya que permiten implementar métodos de forma particular para cada clase heredada si es necesario, evitando tener que definir toda una clase nueva para modificar solo una parte de la clase ya existente.

La decisión de diseño que justifica que los métodos no sean virtuales por defecto es asegurar la EFICIENCIA, haciendo que el programador sólo pague dicho overhead (de tiempo y memoria) si es necesario.

# Paradigmas de la Programación – Examen Final

9 de Agosto de 2023

6. [10 pt.] Explique cómo en el siguiente código en Java se usan los mecanismos de herencia y de interfaz para lograr el comportamiento deseado en el programa sin repetir código.

```
1 interface Animal {  
2     void sound();  
3 }  
4  
5 class Mammal implements Animal {  
6     @Override  
7     public void sound() {  
8         System.out.println("Mammal sound");  
9     }  
10 }  
11  
12 class Dog extends Mammal {  
13     @Override  
14     public void sound() {  
15         System.out.println("Dog barks");  
16     }  
17 }  
18  
19 public class Example {  
20     public static void main(String[] args) {  
21         Animal dog = new Dog();  
22  
23         dog.sound(); // Output: Dog barks  
24     }  
25 }
```

En este código se usa herencia e interfaz combinadas para evitar la repetición de código.

La interfaz `Animal` declara la firma de un método `sound`, que implementa la clase

`Mammal` luego, la clase `Dog`,

hereda la clase `Mammal`, que tiene dentro la implementación de `sound`. Sin embargo `Dog` también sobrescribe este método, cosa que se puede en Java pues todos los métodos se tratan como virtuales si no se indica lo contrario.

Como en Java el subtipado viene de la herencia, podemos instanciar un objeto de la clase `Dog` usando la interfaz `Animal` como tipo.

Estos mecanismos, herencia simple combinado con uso de interfaces, evitan tener que usar herencia múltiple previniendo problemas como los name clashes.