



**Universidad Nacional Autónoma de México**  
**Facultad de Ingeniería**  
**Programación orientada a objetos**  
Tema 3:  
**HERENCIA Y POLIMORFISMO**



## 3 Herencia y polimorfismo

**Objetivo:** Aplicar las diferentes propiedades de la programación orientada a objetos para la resolución de problemas.



# 3 Herencia y polimorfismo

3.1 Herencia.

3.2 Método constructor.

3.3 Polimorfismo (moldeado o casting entre tipos referencia o instancias).

3.4 Referencias a this y a la clase base.

3.5 Modificadores de acceso (encapsulamiento).

3.6 Tipos de clases: abstractas, comunes y finales.

3.7 Interfaces.

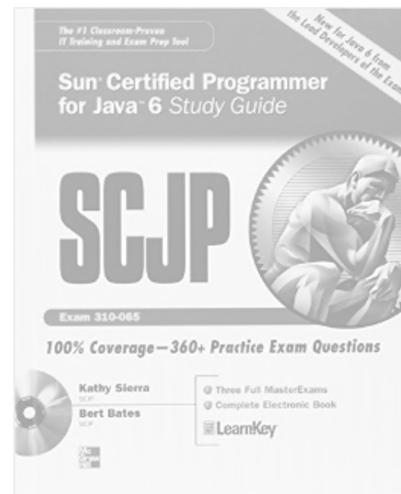
3.8 Paquetes y documentación.

## Bibliografía



**JAVA 2 Curso de programación. Francisco Javier Ceballos, 2da edición Alfaomega, 2003.**

## Bibliografía



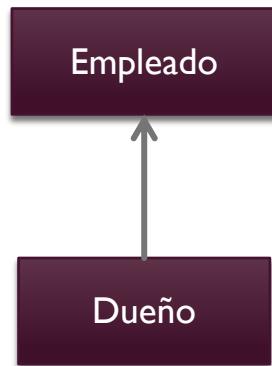
**Sierra Katy, Bates Bert**  
**SCJP Sun Certified Programmer for Java 6 Study Guide**  
**Mc Graw Hill**



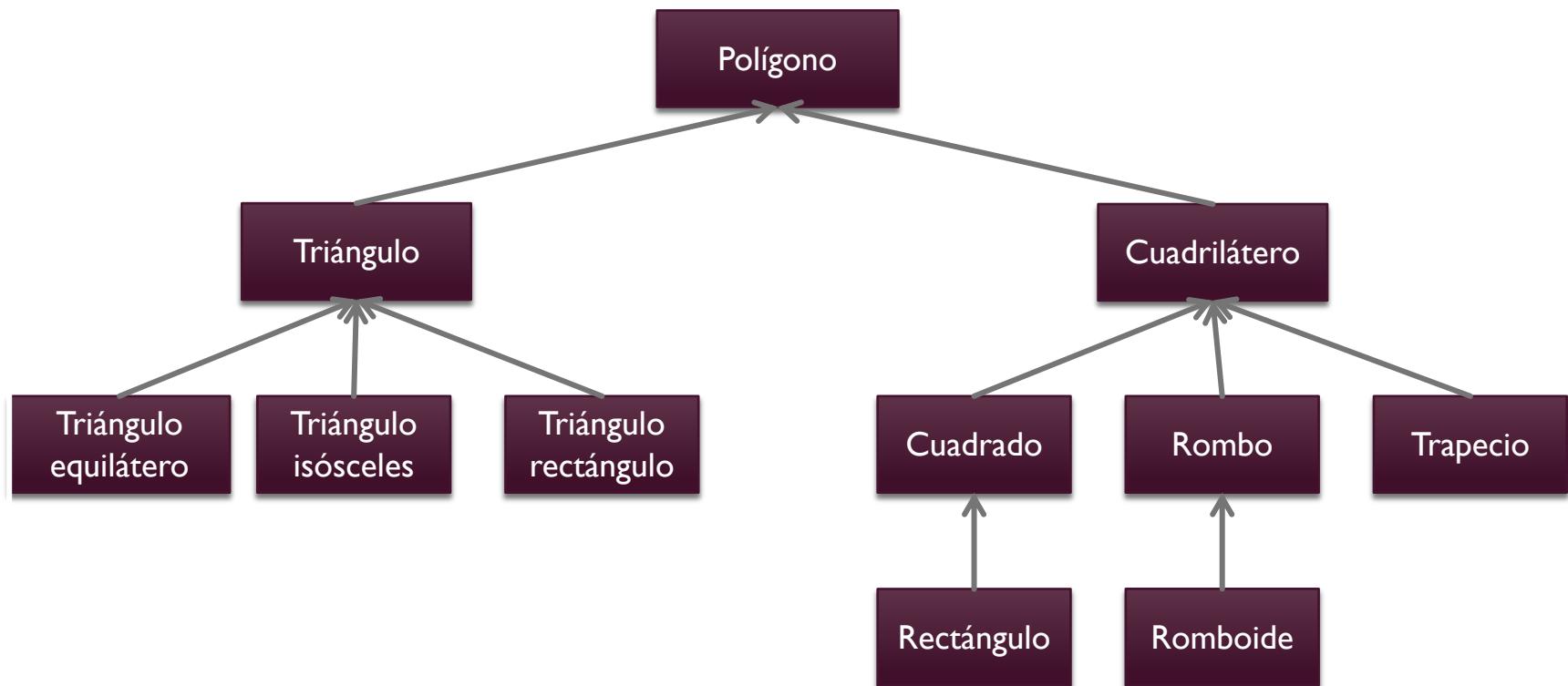
**“The only way to learn a new programming language is by writing programs in it.”**

**Dennis Ritchie**  
An American computer scientist.  
He created the C programming language.

**Abstraer la siguiente jerarquía de clases:**



**Abstraer la siguiente jerarquía de clases:**



**Crear y abstraer la siguiente jerarquía de clases:**

Cuenta

ConexionBD

Email



# HERENCIA Y POLIMORFISMO

TEMA 3

# HERENCIA

## TEMA 3.1

3 HERENCIA Y POLIMORFISMO





**La herencia es una propiedad que permite crear nuevos objetos que asumen las propiedades de objetos existentes.**

**La herencia permite la reutilización de código sin necesidad de volver a escribirlo. Es una manera más limpia de programar, ya que permite programar por módulos funcionales uniendo todos al final.**



**La herencia permite crear un objeto a partir de la definición de otro ya existente, otorgando la posibilidad de compartir automáticamente atributos y métodos entre clases, subclases y, por ende, objetos.**

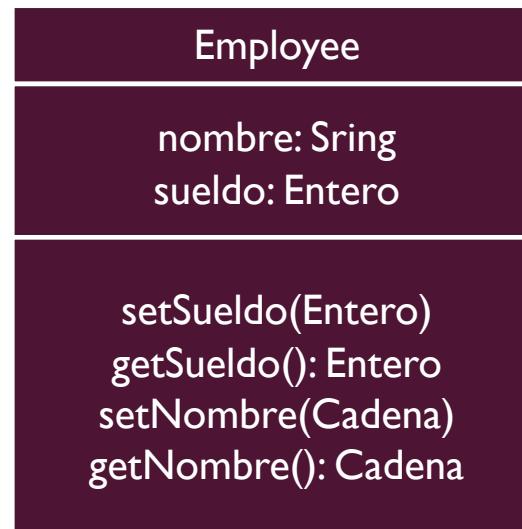
**Las dos razones más comunes para utilizar herencia son:**

- **Para promover la reutilización de código.**
- **Para usar polimorfismo.**

**En un negocio pequeño se tienen dos roles, por un lado está el empleado que atiende a los clientes y por otro el dueño de la tienda.**

**De la descripción anterior se puede concluir que existen dos roles, el rol *empleado* y el rol *dueño*, además de que ambos pueden fungir como empleados de la tienda en la ausencia uno de otro.**

La abstracción del rol empleado sería la siguiente:



Por otro lado, la abstracción del rol Dueño sería la siguiente:

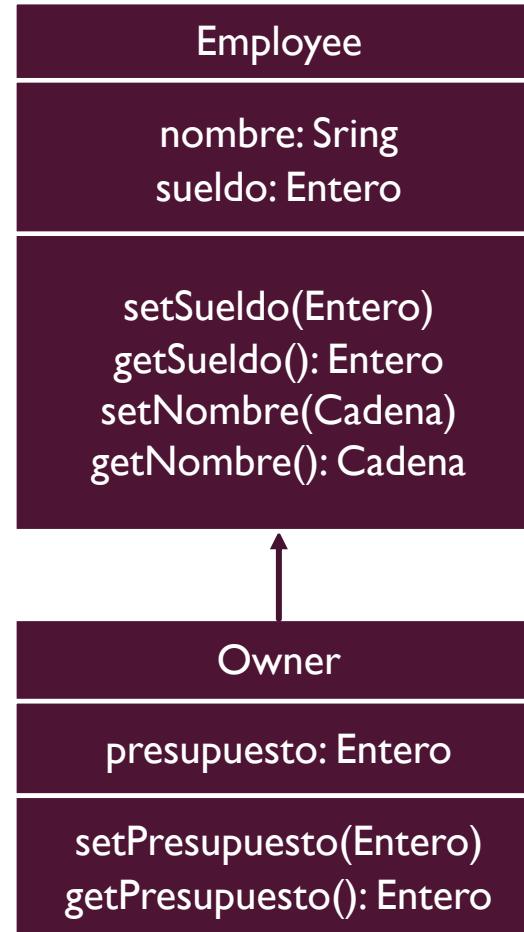
Owner
nombre: String sueldo: Entero presupuesto: Entero
setSueldo(Entero) getSueldo(): Entero setNombre(Cadena) getNombre(): Cadena setPresupuesto(Entero) getPresupuesto(): Entero

**De las abstracciones obtenidas se pueden obtener las siguientes conclusiones:**

- **El rol Dueño tiene atributos y funcionalidades comunes al rol Empleado.**
- **El rol Dueño se puede desempeñar como un Empleado.**

**Como se puede observar, se tendrían que escribir de manera repetida atributos y métodos comunes para ambos roles. Es en estos casos cuando se ocupa la herencia.**

La jerarquía de clases de la tienda sería la siguiente:



```
class Employee {  
    private String nombre;  
    private float sueldo;  
  
    public void setNombre(String n){  
        nombre = n;  
    }  
    public String getNombre(){  
        return nombre;  
    }  
    public void setSueldo(float s){  
        if (s > 0)  
            sueldo = s;  
    }  
    public float getSueldo(){  
        return sueldo;  
    }  
}
```

**Para heredar en Java se utiliza la palabra reservada `extends` al momento de definir la clase.**

**Los objetos de las clases derivadas (subclases) se crean (instancian) igual que los de la clase base y pueden acceder tanto a datos miembro propios como a datos miembro de la clase base.**

## Ejemplo I

```
class Owner extends Employee {  
    private float presupuesto;  
  
    public void setPresupuesto(float p){  
        if (p > 0)  
            presupuesto = p;  
    }  
    public float getPresupuesto(){  
        return presupuesto;  
    }  
}
```

## Ejemplo I

```
class TestOwner {  
    public static void main (String [] args){  
        Employee e = new Employee();  
        e.setNombre("Juan");  
        e.setSueldo(6000);  
        System.out.println("Nombre:" + e.getNombre() +  
                           ", sueldo:" + e.getSueldo());  
  
        Owner o = new Owner();  
        o.setNombre("Jose");  
        o.setSueldo(20000);  
        o.setPresupuesto(50000);  
        System.out.println("Nombre:" + o.getNombre() +  
                           ", sueldo:" + o.getSueldo() +  
                           ", presupuesto:" + o.getPresupuesto());  
    }  
}
```

## **toString()**

**El método `toString` permite mostrar el estado de un objeto, es decir, mostrar los atributos que se consideren importantes de un objeto en algún instante de tiempo.**

**Este método está descrito en la clase `Object` pero se puede modificar desde cualquier clase, su firma es la siguiente:**

**public String `toString()`**

## Ejemplo 2

```
class Employee {  
    String nombre;  
    int sueldo;  
  
    public void setNombre(String nombre){  
        this.nombre = nombre;  
    }  
    public String getNombre(){  
        return nombre;  
    }  
    public void setSueldo(int sueldo){  
        if (s > 0)  
            this.sueldo = sueldo;  
    }  
    public int getSueldo(){  
        return sueldo;  
    }  
    @override  
    public String toString(){  
        return "Nombre: " + nombre + "\nSueldo: " + sueldo;  
    }  
}
```

### Sobrescritura

**Es una propiedad que permite a partir de un método definido en la clase base modificar su comportamiento en la clase derivada.**

**La sobrescritura solo tiene sentido en la herencia, es decir, en una clase no se podría tener dos métodos con la misma firma, ya que el compilador (ni el programador) sabrían a qué método llamar.**

Para el ejemplo anterior, la clase Owner hereda el método `toString()` de la clase Employee, sin embargo, este método no imprime el atributo presupuesto, ya que es un miembro propio de Gerente. Por ello, se debe volver a definir este método para que imprima el atributo faltante en la clase Owner.

## Ejemplo 2

```
class Owner extends Empleado {  
    int presupuesto;  
    public void setPresupuesto(int presupuesto){  
        if (p > 0)  
            this.presupuesto = presupuesto;  
    }  
    public int getPresupuesto(){  
        return presupuesto;  
    }  
    @override  
    public String toString() {  
        return "Nombre: " + nombre +  
            "\nSueldo: " + sueldo +  
            " Presupuesto: " + presupuesto;  
    }  
}
```

**En la clase Owner se sobrescribe el método `toString` de la clase Employee para agregar el atributo presupuesto.**

**De esta manera, cuando se manda imprimir un objeto, de manera implícita se manda llamar al método `toString` de la clase base. Es importante aclarar que si no se sobrescribiese el método `toString` y se enviase a imprimir un objeto, por defecto se ejecutaría como está implementado en la clase Object.**

## Ejemplo 2

```
class TestOwner {  
    public static void main (String [] args){  
        Employee e = new Employee();  
        e.setNombre("Juan");  
        e.setSueldo(6000);  
        System.out.println(e);  
        Owner o = new Owner();  
        o.setNombre("Jose");  
        o.setSueldo(20000);  
        o.setPresupuesto(50000);  
        System.out.println(o);  
    }  
}
```

**Crear una fábrica de 5 empleados y un propietario, utilizando todo lo visto hasta el momento y cuidando la eficiencia del código.**



**“Before software can be reusable it first has to be usable.”**

**Ralph Johnson**  
**(He is a Research Associate Professor in the**  
**Department of Computer Science at the**  
**University of Illinois at Urbana-Champaign.)**



## 3.2 MÉTODO CONSTRUCTOR.



**El constructor de una clase es un método estándar para inicializar los objetos de esa clase. Esta función se ejecuta siempre al crear un objeto.**



**Si no se declara un constructor explícitamente, el sistema crea un constructor por defecto sin argumentos. Cuando se define un constructor de manera explícita, el constructor del sistema se elimina y es sustituido por éste o los constructores definidos.**

## Constructor

**Las características principales de los métodos constructores son:**

- **El nombre del constructor tiene que ser igual al de la clase.**
- **Puede recibir cualquier número y tipo de argumentos.**
- **No devuelve ningún valor (en su firma no se declara ni siquiera void).**

**El constructor no es un miembro más de una clase, por lo tanto, solo es invocado cuando se crea el objeto (con el operador new) y no puede ser invocado explícitamente en ningún otro momento.**

**Los constructores proveen un mecanismo eficiente para inicializar objetos. Por ejemplo, para agregar valores a los atributos de la clase Employee se realizaba lo siguiente:**

```
Employee e = new Employee();
e.setNombre("Juan");
e.setSueldo(6000);
```

## Sobrecarga

**La sobrecarga de métodos permite usar el mismo nombre de un método en una clase, pero con diferentes argumentos (y, opcionalmente, con diferentes valores de retorno). Los métodos sobrecargados hacen más cómoda la implementación de un objeto y, por ende, la utilización de los métodos del mismo.**



**Los métodos sobrecargados tienen las siguientes características:**

- **Deben tener el mismo nombre.**
- **Deben cambiar la lista de argumentos.**
- **Pueden cambiar el valor de retorno.**
- **Pueden cambiar el nivel de acceso.**

**Cuando se llama un método sobrecargado, Java sigue algunas reglas para determinar el método concreto que debe llamar:**

- Si existe el método cuyos argumentos se ajustan exactamente al tipo de los argumentos de la llamada (argumentos actuales), se llama ese método.
- Si no existe un método que se ajuste exactamente, se intenta promover los argumentos actuales al tipo inmediatamente superior (up-casting o cast implícito) y se llama el método correspondiente.

- Si sólo existen métodos con argumentos de un tipo más restringido, el programador debe hacer un cast explícito (down-casting) en la llamada, responsabilizándose de esta manera de lo que pueda ocurrir.
- El valor de retorno no influye en la elección del método sobrecargado. En realidad es imposible saber desde el propio método lo que se va a hacer con él. No es posible crear dos métodos sobrecargados con el mismo nombre que sólo difieran en el valor de retorno.

**Una manera más eficiente de inicializar los valores se puede realizar utilizando un constructor**

```
Employee(String n, float s){  
    setNombre(n);  
    setSueldo(s);  
}
```

**De tal manera que para inicializar un empleado solo basta una línea:**

```
Employee e = new Employee("Juan", 6000);
```

**Si no se declara explícitamente un constructor, java genera un constructor por defecto (llamado constructor no-args).**

```
// Ejemplo constructor no-args  
Employee() { }
```

**Este constructor tiene el mismo nombre y tipo de acceso de la clase (este tema se verá más adelante) y no recibe argumentos.**

**Es importante aclarar que lo primero que se realiza dentro del método constructor es la reservación de memoria dentro del heap para todos los atributos del objeto.**



**Es posible declarar más de un constructor con la condición de que todos reciban parámetros distintos. A esta propiedad se le conoce como sobrecarga y permite que un objeto pueda inicializarse de varias formas.**

## Ejemplo 3

---

```
class Employee {  
    String nombre;  
    float sueldo;  
  
    Employee(String n){  
        setNombre(n);  
    }  
  
    Employee(String n, float s){  
        setNombre(n);  
        setSueldo(s);  
    }  
}
```

## Ejemplo 3

```
public void setNombre(String n){  
    nombre = n;  
}  
public String getNombre(){  
    return nombre;  
}  
public void setSueldo(float s){  
    if (s > 0)  
        sueldo = s;  
}  
public float getSueldo(){  
    return sueldo;  
}  
public String toString(){  
    return "Nombre: " + nombre + "\nSueldo: " + sueldo;  
}  
}
```

## Ejemplo 3

```
class TestEmployee {  
    public static void main (String [] args){  
        Employee e1 = new Employee("Juan", 6000);  
        System.out.println(e1);  
  
        Employee e2 = new Employee("Miguel");  
        System.out.println(e2);  
    }  
}
```

## Inicialización de clases derivadas

Cuando se crea un objeto de una clase derivada se crea, de manera implícita, un objeto de la clase base. La creación de objetos de la super clase se realiza hasta que se llega a la clase Object.

**Cuando se crea un objeto de una subclase, implícitamente, se hace una llamada al constructor no-args de la clase base. Por lo tanto, es importante que cuando se modifica un constructor se cree de manera explícita un constructor sin argumentos para crear objetos simples (sin inicializar valores).**

**En caso de que no exista un constructor no-args en una clase base, el compilador marca un error indicando que el constructor sin argumentos no se encuentra en la clase base.**

## Ejemplo 4

```
class Employee {  
    String nombre;  
    float sueldo;  
  
    Employee(){  
    }  
  
    Employee(String n){  
        setNombre(n);  
    }  
  
    Employee(String n, float s){  
        setNombre(n);  
        setSueldo(s);  
    }  
}
```

## Ejemplo 4

```
public void setNombre(String n){  
    nombre = n;  
}  
public String getNombre(){  
    return nombre;  
}  
public void setSueldo(float s){  
    if (s > 0)  
        sueldo = s;  
}  
public float getSueldo(){  
    return sueldo;  
}  
public String toString(){  
    return "Nombre: " + nombre + "\nSueldo: " + sueldo;  
}  
}
```

## Ejemplo 4

```
class Owner extends Employee {  
    float presupuesto;  
  
    Owner() {}  
  
    Owner (String n, float s, float p){  
        nombre = n;  
        sueldo = s;  
        presupuesto = p;  
    }  
  
    Owner (String n, float s){  
        nombre = n;  
        sueldo = s;  
    }  
  
    Owner (String n){  
        nombre = n;  
    }  
}
```

## Ejemplo 4

```
public void setPresupuesto(float p){  
    if (p > 0)  
        presupuesto = p;  
}  
  
public float getPresupuesto(){  
    return presupuesto;  
}  
  
public String toString(){  
    String s = super.toString();  
    return "Nombre: " + nombre +  
        "\nSueldo: " + sueldo +  
        ", presupuesto:" + presupuesto;  
}  
}
```

## Ejemplo 4

```
class TestOwner {  
    public static void main (String [] args){  
        Owner o1 = new Owner("Roberto", 15000, 50000);  
        System.out.println(o1);  
        Owner o2 = new Owner("Nadia", 15000);  
        System.out.println(o2);  
        Owner o3 = new Owner("Benito");  
        System.out.println(o3);  
    }  
}
```

## 3.3 POLIMORFISMO.



**Polimorfismo se refiere a la habilidad de tener diferentes formas. Así mismo, el término IS-A se refiere a la pertenencia de un objeto con un tipo, es decir, si se crea un objeto de tipo A, se dice que el objeto creado es un A.**



**En Java, cualquier objeto que pueda comportarse como más de un IS-A (es un) puede ser considerado polimórfico.**

**Debido a que en Java todos los objetos se pueden comportar como objetos de su propio tipo y como objetos de la clase Object, todos los objetos pueden ser considerados polimórficos.**

**En la jerarquía de clases el polimorfismo aumenta, ya que el objeto puede adquirir la forma de su clase base y la forma de la clase base de su súper clase hasta llegar a Object.**

**Por otro lado, debido a que la única manera de acceder a un objeto es a través de su referencia, existen algunos puntos clave que se deben recordar sobre las mismas:**

- **Una referencia puede ser solo de un tipo y, una vez declarado, el tipo no puede ser cambiado.**
- **Una referencia es una variable, por lo tanto, ésta puede ser reasignada a otros objetos (a menos que la referencia sea declarada como final).**

- **El tipo de una referencia determina los métodos que pueden ser invocados del objeto al que referencia, es decir, solo se pueden ejecutar los métodos definidos en el tipo de la referencia.**
- **A una referencia se le puede asignar cualquier objeto que sea del mismo tipo con el que fue declarada la referencia o de algún subtipo (Polimorfismo).**

Dada la siguiente jerarquía de clases:

0

Polígono

1

Triángulo

Cuadrilátero

2

Triángulo  
equilátero

Triángulo  
isósceles

Triángulo  
rectángulo

Cuadrado

Rombo

Trapecio

3

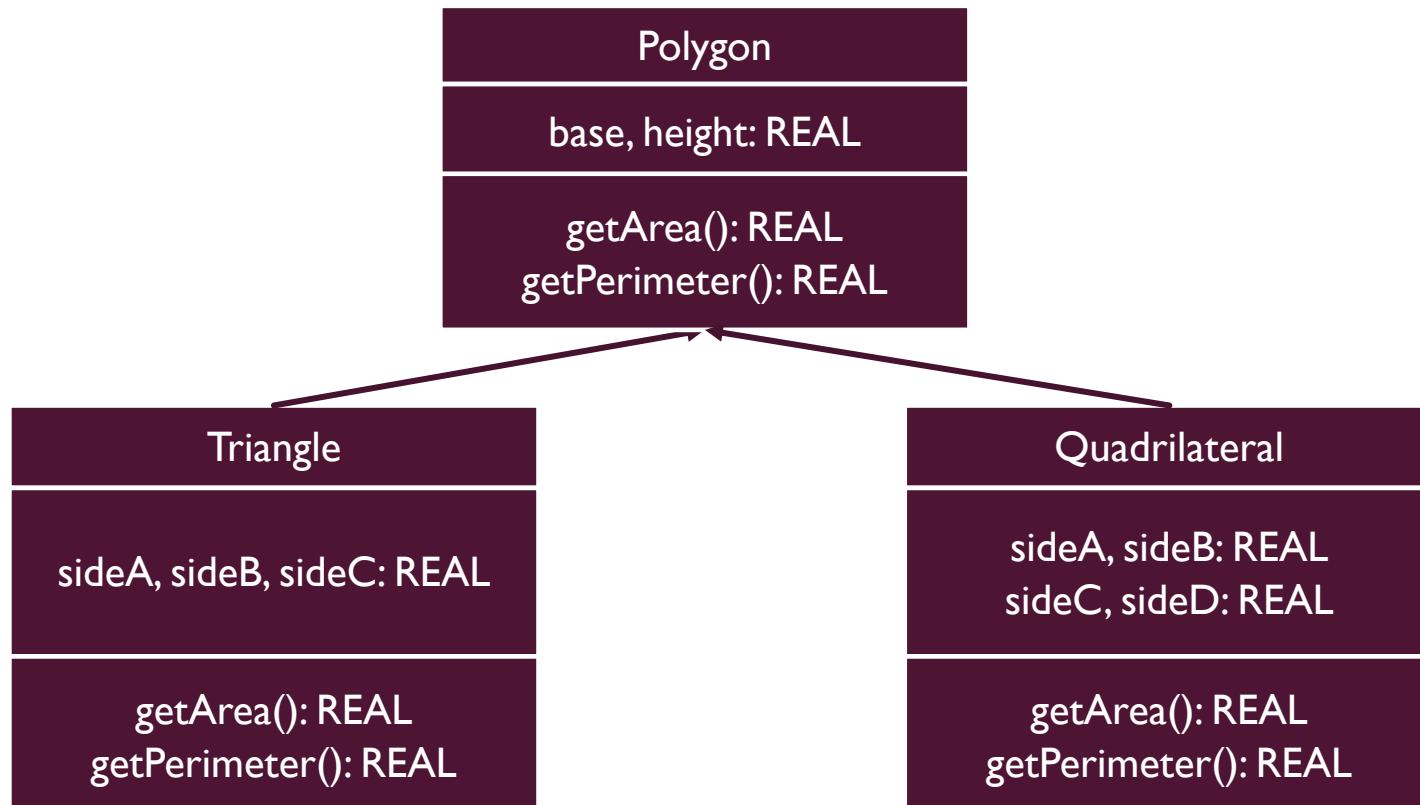
Rectángulo

Romboide



**Abstraer las características necesarias para genera los objetos con las propiedades necesarias para calcular las áreas y perímetros de cada uno de los polígonos.**

Los diagramas de clases de los objetos Polígono, Triángulo y Cuadrilátero son:



## Ejemplo 5

```
class Polygon {  
    float base, height;  
  
    public float getBase(){  
        return base;  
    }  
  
    public float getHeight(){  
        return height;  
    }  
  
    public double getArea(){  
        return 0.0d;  
    }  
  
    public double getPerimeter(){  
        return 0.0d;  
    }  
}
```

## Ejemplo 5

```
class Triangle extends Polygon {  
    float sideA, sideB, sideC;  
  
    Triangle (){}  
  
    Triangle (float ba, float h,  
              float a, float b, float c){  
        base = ba;  
        height = h;  
        sideA = a;  
        sideB = b;  
        sideC = c;  
    }  
  
    public float getSideA(){  
        return sideA;  
    }  
  
    public float getSideB(){  
        return sideB;  
    }  
  
    public float getSideC(){  
        return sideC;  
    }  
  
    public double getArea(){  
        return (base * height) / 2;  
    }  
  
    public double getPerimeter(){  
        return sideA + sideB + sideC;  
    }  
  
    public boolean hasParallelSides(){  
        return false;  
    }  
}
```

## Ejemplo 5

```
class Quadrilateral extends Polygon {  
    float sideA, sideB, sideC, sideD;  
  
    Quadrilateral (){}  
  
    Quadrilateral (float ba, float h,  
                  float a, float b, float c, float d){  
        base = ba;  
        height = h;  
        sideA = a;  
        sideB = b;  
        sideC = c;  
        sideD = d;  
    }  
  
    public float getSideA(){  
        return sideA;  
    }
```

```
    }  
  
    public float getSideB(){  
        return sideB;  
    }  
  
    public float getSideC(){  
        return sideC;  
    }  
  
    public float getSideD(){  
        return sideD;  
    }  
  
    public double getArea(){  
        return base * height;  
    }  
  
    public double getPerimeter(){  
        return sideA + sideB + sideC + sideD;  
    }  
  
    public boolean hasParallelSides(){  
        return false;  
    }  
}
```



**El polimorfismo permite que una referencia de un tipo se comporte como una referencia de otro tipo mientras exista una relación de herencia.**

**Para el ejemplo anterior, se puede tener un polígono que se comporte como triángulo o como cuadrilátero, debido a la jerarquía de clases entre ellos. La desventaja es que la referencia Polygon sólo puede ejecutar los métodos declarados en dicha clase.**

## Ejemplo 5

```
class GeometricFigures {  
    public static void main (String [] args){  
        Polygon p = new Triangle(5, 6, 2, 2, 2);  
        System.out.println(p.getArea());  
        System.out.println(p.getPerimeter());  
        // System.out.println(p.hasParallelSides());  
  
        System.out.println("\n#####\n");  
  
        p = new Quadrilateral(5, 6, 2, 2, 2, 2);  
        System.out.println(p.getArea());  
        System.out.println(p.getPerimeter());  
        // System.out.println(p.hasParallelSides());  
    }  
}
```

## Cast o moldeo de objetos

Es posible convertir un objeto de un tipo más general a uno más específico mientras exista una relación de herencia a través de un cast o moldeo. La única restricción es que el objeto sea de un tipo más específico.

**Para validar si el objeto al que apunta la referencia es de un tipo en específico se utiliza la palabra reservada instanceof de la siguiente manera:**

```
ref instanceof Triangulo
```

**En este caso se verifica si la referencia ref está apuntando en el heap a un objeto del tipo Triangulo. La validación anterior devuelve verdadero o falso.**

## Ejemplo 5

```
class GeometricFigures {  
    public static void main (String [] args){  
        Polygon p = new Triangle(5, 6, 2, 2, 2);  
        System.out.println(p.getArea());  
        System.out.println(p.getPerimeter());  
        if (p instanceof Triangle){  
            Triangle t = (Triangle)p;  
            System.out.println(t.hasParallelSides());  
        }  
  
        System.out.println("\n#####\n");  
  
        p = new Quadrilateral(5, 6, 2, 2, 2, 2);  
        System.out.println(p.getArea());  
        System.out.println(p.getPerimeter());  
        if (p instanceof Quadrilateral){  
            Quadrilateral q = (Quadrilateral)p;  
            System.out.println(q.hasParallelSides());  
        }  
    }  
}
```

## Método final

El modificador *final* en un método impide que éste pueda redefinirse en una clase derivada, es decir, protege su comportamiento durante todo el tiempo, no importa cuantas veces se herede o cuántas clases hereden de la clase que lo contiene, un método final no se puede modificar.

## Ejemplo 6

```
class Triangle extends Polygon {  
    float sideA, sideB, sideC;  
  
    Triangle (){}  
  
    Triangle (float ba, float h,  
              float a, float b, float c){  
        base = ba;  
        height = h;  
        sideA = a;  
        sideB = b;  
        sideC = c;  
    }  
  
    public float getSideA(){  
        return sideA;  
    }  
  
    public float getSideB(){  
        return sideB;  
    }  
  
    public float getSideC(){  
        return sideC;  
    }  
  
    public final double getArea(){  
        return (base * height) / 2;  
    }  
  
    public double getPerimeter(){  
        return sideA + sideB + sideC;  
    }  
  
    public boolean hasParallelSides(){  
        return false;  
    }  
}
```

## Ejemplo 6

```
class IsoscelesTriangle extends Triangle {  
    public double getArea(){  
        return base * height * 2;  
    }  
  
    public double getPerimeter(){  
        return 3*sideA;  
    }  
}
```

**Cuando se compila la clase IsoscelesTriangle.java el compilador detecta que se quiere volver a definir un método final y marca un error:**

**IsoscelesTriangle.java:2: getArea() in IsoscelesTriangle cannot override getArea() in Triangle; overridden method is final**

```
public double getArea(){  
    ^
```

**1 error**



## 3.4 REFERENCIAS A THIS Y A LA CLASE BASE.

## Referencia a this

**La palabra reservada this es una referencia que tienen todos los objetos y que apunta a sí mismo.**

**this evita lo que se conoce como shadowing, esto es, que el nombre de una variable le haga sombra a otra variable porque poseen el mismo nombre.**

```
class Circle {  
    float radius;  
  
    public void setRadius(float radius){  
        this.radius = radius;  
    }  
  
    public float getRadius (){  
        return radius;  
    }  
  
    public Circle getLargerRadius(Circle c) {  
        if (this.radius > c.radius)  
            return this;  
        else  
            return c;  
    }  
}
```

## Ejemplo 7

```
class TestCircle {  
    public static void main (String [] args){  
        Circle c1 = new Circle();  
        Circle c2 = new Circle();  
  
        c1.setRadius(8.5f);  
        c2.setRadius(10.5f);  
  
        Circle circleLargerRadius = c1.getLargerRadius(c2);  
        System.out.println("El radio más grande es:"  
                           + circleLargerRadius.getRadius());  
  
        circleLargerRadius = c2.getLargerRadius(c1);  
        System.out.println("El radio más grande es:"  
                           + circleLargerRadius.getRadius());  
    }  
}
```



En el ejemplo anterior, el método `elMayor()` devuelve una referencia al círculo que tiene el radio mayor, comparando el radio del círculo `c` que se recibe como argumento y el radio de la propia clase (`this`). En caso de que el radio de la propia clase resulte mayor, el método debe devolver una referencia a sí mismo, a través de la referencia a `this`.

## this en constructores

Por otro lado, desde un constructor puede invocarse explícitamente a otro constructor utilizando la palabra reservada `this`.

La única restricción es que la llamada a `this` debe ser la primera instrucción dentro del constructor.

## Ejemplo 8

```
class Polygon {  
    float base, height;  
  
    public float getBase(){  
        return base;  
    }  
  
    public float getHeight(){  
        return height;  
    }  
  
    public double getArea(){  
        return 0.0d;  
    }  
  
    public double getPerimeter(){  
        return 0.0d;  
    }  
}
```

## Ejemplo 8

```
class Triangle extends Polygon {  
    float sideA, sideB, sideC;  
  
    Triangle (){  
        this(0, 0, 0, 0, 0);  
    }  
  
    Triangle (float base, float height){  
        this(base, height, 0, 0, 0);  
    }  
  
    Triangle (float sideA, float sideB,  
              float sideC){  
        this (0, 0, sideA, sideB, sideC);  
    }  
  
    Triangle (float base, float height,  
              float sideA, float sideB,  
              float sideC){  
        this.base = base;  
        this.height = height;  
        this.sideA = sideA;  
        this.sideB = sideB;  
        this.sideC = sideC;  
    }  
  
    public float getSideA(){  
        return sideA;  
    }  
  
    public float getSideB(){  
        return sideB;  
    }  
  
    public float getSideC(){  
        return sideC;  
    }  
  
    public final double getArea(){  
        return (base * height) / 2;  
    }  
  
    public double getPerimeter(){  
        return sideA + sideB + sideC;  
    }  
}
```

## Ejemplo 8

```
class GeometricFigures {  
    public static void main (String [] args){  
        Polygon p = new Triangle(5, 6, 2, 2, 2);  
        System.out.println("new Triangle (5, 6, 2, 2, 2)");  
        System.out.println(p.getArea());  
        System.out.println(p.getPerimeter());  
  
        p = new Triangle(8, 4);  
        System.out.println("new Triangle (8, 4)");  
        System.out.println(p.getArea());  
        System.out.println(p.getPerimeter());  
  
        p = new Triangle(1, 2, 3);  
        System.out.println("new Triangle (1, 2, 3)");  
        System.out.println(p.getArea());  
        System.out.println(p.getPerimeter());  
  
        p = new Triangle();  
        System.out.println("new Triangle ()");  
        System.out.println(p.getArea());  
        System.out.println(p.getPerimeter());  
    }  
}
```

## **Referencia a la clase base**

**Cuando se crea un objeto de una clase derivada se crea, implícitamente, un referencia a la clase base, permitiendo con esto hacer uso de los miembros de ésta (atributos y métodos).**

**En Java la referencia a la clase base se llama *super* y, a través de ella, se pueden acceder a todos los elementos de la súper clase.**

## Ejemplo 9

```
class Polygon {  
    float base, height;  
  
    Polygon(){  
        this(0, 0);  
    }  
  
    Polygon(float base, float height){  
        if (base > 0)  
            this.base = base;  
        if (height > 0)  
            this.height = height;  
    }  
  
    public double getArea(){  
        return 0.0d;  
    }  
  
    public double getPerimeter(){  
        return 0.0d;  
    }  
  
    public String toString(){  
        return "Base: " + base +  
               "\nHeight: " + height;  
    }  
}
```

## Ejemplo 9

```
class Triangle extends Polygon {  
    float sideA, sideB, sideC;
```

```
    Triangle (){  
        this(5, 5, 5, 5, 5);  
    }
```

```
    Triangle (float base, float height){  
        this(base, height, 5, 5, 5);  
    }
```

```
    Triangle (float sideA, float sideB,  
              float sideC){  
        this (5, 5, sideA, sideB, sideC);  
    }
```

```
    Triangle (float base, float height,  
              float sideA, float sideB,  
              float sideC){  
        this.base = base;  
        this.height = height;  
        this.sideA = sideA;  
        this.sideB = sideB;  
        this.sideC = sideC;  
    }
```

```
    public final double getArea(){  
        return (base * height) / 2;  
    }
```

```
    public double getPerimeter(){  
        return sideA + sideB + sideC;  
    }
```

```
    public String toString(){  
        return super.toString() +  
            "\nSide A: " + sideA +  
            "\nSide B: " + sideB +  
            "\nSide C: " + sideC;  
    }
```

## Ejemplo 9

```
class GeometricFigures {  
    public static void main (String [] args){  
        Polygon p = new Triangle(5, 6, 2, 2, 2);  
        System.out.println("new Triangle (5, 6, 2, 2, 2)");  
        System.out.println(p);  
  
        p = new Triangle(8, 4);  
        System.out.println("new Triangle (8, 4)");  
        System.out.println(p);  
  
        p = new Triangle(1, 2, 3);  
        System.out.println("new Triangle (1, 2, 3)");  
        System.out.println(p);  
  
        p = new Triangle();  
        System.out.println("new Triangle ()");  
        System.out.println(p);  
    }  
}
```

## Referencia a la clase base en constructores

Cuando se crea un objeto se realiza de manera implícita una llamada al constructor sin argumentos de la clase base (por eso es siempre importante definir este constructor cuando se sobre cargan los mismos).

Sin embargo, si se requiere utilizar constructores sobrecargados en la clase base es necesario invocarlos explícitamente a través de su referencia.

## Ejemplo 10

```
class Polygon {  
    float base, height;  
  
    Polygon(){  
        this(0, 0);  
    }  
  
    Polygon(float base, float height){  
        if (base > 0)  
            this.base = base;  
        if (height > 0)  
            this.height = height;  
    }  
  
    public double getArea(){  
        return 0.0d;  
    }  
  
    public double getPerimeter(){  
        return 0.0d;  
    }  
  
    public String toString(){  
        return "Base: " + base +  
               "\nHeight: " + height;  
    }  
}
```

## Ejemplo 10

```
class Triangle extends Polygon {  
    float sideA, sideB, sideC;
```

```
    Triangle (){  
        this(0, 0, 0, 0, 0);  
    }
```

```
    Triangle (float base, float height){  
        this(base, height, 0, 0, 0);  
    }
```

```
    Triangle (float sideA, float sideB,  
              float sideC){  
        this (0, 0, sideA, sideB, sideC);  
    }
```

```
    Triangle (float base, float height,  
              float sideA, float sideB,  
              float sideC){  
        super(base, height);  
        this.sideA = sideA;  
        this.sideB = sideB;  
        this.sideC = sideC;  
    }
```

```
    public final double getArea(){  
        return (base * height) / 2;  
    }
```

```
    public double getPerimeter(){  
        return sideA + sideB + sideC;  
    }
```

```
    public String toString(){  
        return super.toString() +  
            "\nSide A: " + sideA +  
            "\nSide B: " + sideB +  
            "\nSide C: " + sideC;  
    }
```

**Es importante resaltar que el constructor de la clase Triangle que recibe como parámetros cinco valores flotantes es el que invoca explícitamente al constructor de la clase Polygon mediante la palabra reservada super.**

**La invocación al constructor de la superclase debe ser la primera sentencia del constructor.**

```
class GeometricFigures {  
    public static void main (String [] args){  
        Polygon p = new Triangle(5, 6, 2, 2, 2);  
        System.out.println("new Triangle (5, 6, 2, 2, 2)");  
        System.out.println(p);  
  
        p = new Triangle(8, 4);  
        System.out.println("new Triangle (8, 4)");  
        System.out.println(p);  
  
        p = new Triangle(1, 2, 3);  
        System.out.println("new Triangle (1, 2, 3)");  
        System.out.println(p);  
  
        p = new Triangle();  
        System.out.println("new Triangle ()");  
        System.out.println(p);  
    }  
}
```

## Ejemplo 11

```
class Super{  
    public int i = 0;  
  
    public Super(String text){  
        i = 1;  
    }  
}  
  
class Sub extends Super{  
    public Sub(String text){  
        i = 2;  
    }  
  
    public static void main(String args[]){  
        Sub sub = new Sub("Hello");  
        System.out.println(sub.i);  
    }  
}
```



## 3.5 MODIFICADORES DE ACCESO (ENCAPSULAMIENTO).



**Los modificadores son palabras reservadas del lenguaje que se colocan antes de definir un elemento (atributos, métodos o, incluso, clases) y que alteran o condicionan el acceso a dicho elemento.**

**Por lo tanto, los modificadores de acceso permiten determinar el acceso a los datos y métodos miembros de una clase. Los modificadores de clases se van a tratar más adelante.**



**Los modificadores de acceso preceden a la declaración de un elemento de la clase.**

**[modificadores] tipo\_variable nombre;**

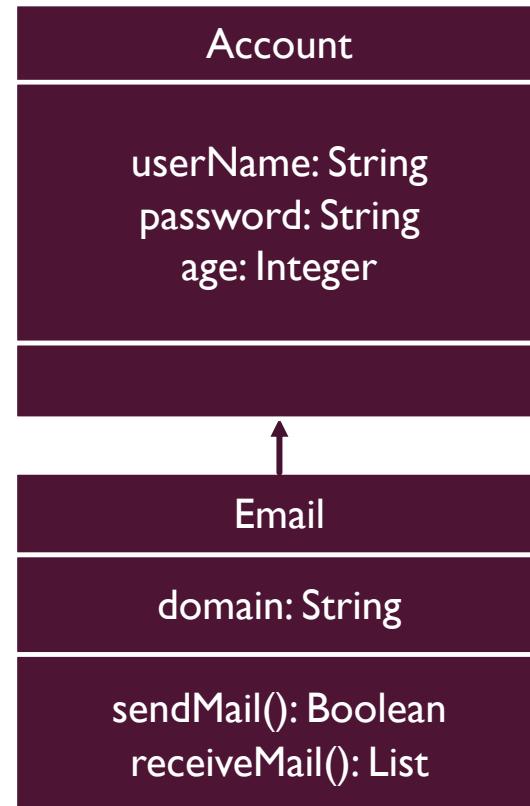
**[modificadores] tipo\_devuelto metodo([parámetros]);**

## Java posee 4 modificadores de acceso, los cuales son:

- **private:** Sólo se puede acceder al atributo desde métodos de la clase. Sólo puede invocarse un método privado desde otro método de la clase.
- **Default (sin modificador):** Se puede acceder al elemento desde cualquier clase del mismo paquete donde se define la clase.
- **protected:** Proporciona acceso público para las clases derivadas y acceso privado (prohibido) para el resto de clases.
- **public:** Se puede acceder al elemento en cualquier momento. Si es un dato miembro, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo. Si es un método todo el mundo puede invocarlo.

	public	protected	sin modificador	private
Clase	Sí	Sí	Sí	Sí
Subclase en el mismo paquete	Sí	Sí	Sí	No
No-Subclase en el mismo paquete	Sí	Sí	Sí	No
Subclase en diferente paquete	Sí	Sí	No	No
No-Subclase en diferente paquete (Universo)	Sí	No	No	No

### Diagrama de clases de los objetos Cuenta y Correo electrónico:



```
public class Account {  
    private String userName;  
    private String password;  
    private short age;  
}
```



```
public class Email extends Account{  
    private String domain;  
}
```

## Ejemplo 12

```
public class TestEmail {  
    public static void main (String [] args){  
        Email mail = new Email();  
        mail.userName = "user1";  
        mail.password = "qwerty";  
        mail.age = (short)20;  
        mail.domain = "gmail.com";  
    }  
}
```



**Todos los atributos de una clase se deben proteger restringiendo el acceso mediante el modificador `private`.**

**Así mismo, el acceso a los atributos se debe realizar mediante los métodos respectivos para proteger la integridad de los datos. Los modificadores de acceso de los métodos deben ser públicos.**

## Ejemplo 13

```
public class Account {  
    private String userName;  
    private String password;  
    private short age;  
  
    public Account() {}  
  
    public Account(String userName, String password, short age) {  
        this.setUserName(userName);  
        this.setPassword(password);  
        this.setAge(age);  
    }  
    public String getUserName(){  
        return userName;  
    }  
    public String getPassword(){  
        return password;  
    }  
}
```

## Ejemplo 13

```
public short getAge(){
    return age;
}
public void setUserName(String userName){
    this.userName = userName.trim();
}
public void setPassword(String password){
    this.password = password.trim();
}
public void setAge(short age){
    if (age > 17 && age < 130) {
        this.age = age;
    }
}
public String toString() {
    return "User name=" + getUserName() +
        "\nPassword=*****" + "\nAge=" + getAge();
}
```

## Ejemplo 13

```
public class Email extends Account{
    private String domain;

    public Email(){}
    public Email(String userName, String password, short age, String domain){
        super(userName, password, age);
        this.setDomain(domain);
    }
    public void setDomain(String domain){
        this.domain = domain.trim();
    }
    public String getDomain(){
        return domain;
    }
    public String toString(){
        return super.toString() + "\nDomain=" + getDomain();
    }
}
```

## Ejemplo 13

```
public class TestEmail {  
    public static void main (String [] args){  
        Email mail = new Email("user1", "qwerty", (short)22, "gmail.com");  
        System.out.println(mail);  
    }  
}
```



## 3.6 TIPOS DE CLASES: ABSTRACTAS, COMUNES Y FINALES.



**Como se mencionó en el tema anterior, Java permite definir distintos tipos de clase dependiendo del comportamiento esperado u objetivo final.**

**En este tema se implementarán todos los posibles tipos de clases que se pueden implementar en Java.**

**Por su nivel de acceso, las clases pueden ser públicas o no poseer ningún modificador (default).**

**Por su tipo, las clases se pueden clasificar en concretas y abstractas.**

**Por jerarquía de clases, pueden ser abstractas (más general) o finales (más específico).**

**Una clase no puede ser declarada como privada o protegida o estática al menos que sea miembro de otra clase. Estas clases se conocen como clases internas.**

## Clase pública

**Una clase pública proporciona acceso a todas las clases desde cualquier paquete, es decir, todas las clases en el Universo de clases tienen acceso a ella.**

**Para poder utilizar una clase de tipo publica sólo es necesario importar el paquete al que pertenece. Una vez hecho lo anterior, se puede crear una instancia de dicha clase en cualquier momento.**

**La sintaxis de una clase pública es la siguiente:**

```
public class NombreClase {  
    // Comentarios NombreClase  
    // Atributos de NombreClase  
    // Métodos de NombreClase  
}
```



## **Clase sin modificador**

**Una clase sin modificador proporciona acceso a todas las clases que pertenecen a su mismo paquete, hereden o no de ella.**

**Por tanto, para poder utilizar una clase sin modificador de acceso sólo es necesario pertenecer al paquete de dicha clase.**



**La sintaxis de una clase sin modificador es la siguiente:**

```
class NombreClase {
    // Comentarios de NombreClase
    // Atributos de NombreClase
    // Métodos de NombreClase
}
```

## **Clase abstracta**

**Una clase abstracta define la existencia de métodos, pero no su implementación, es decir, permite identificar métodos comunes para la toda jerarquía de clase sin especificar cómo se deben realizar las acciones de dichos métodos.**

**Las clases abstractas sirven como modelo para la creación de otras clases (clases derivadas).**



**Las características principales de las clases abstractas son:**

- **Pueden contener métodos abstractos y métodos concretos (con o sin implementación).**
- **Pueden contener atributos.**
- **Pueden heredar de otras clases.**



**La sintaxis de una clase abstracta es la siguiente:**

```
[modificadores] abstract class NombreClase {  
    [modificadores] abstract float metodo();  
}
```



**Una clase abstracta se declara simplemente con el modificador *abstract*, adicionalmente, se puede usar un modificador de acceso.**

**Un método abstracto se declara de utilizando el modificador *abstract*, pero no se define una implementación, es decir, sólo se define la firma de la función que deben respetar todas las clases que sobrescriban dicho método.**

**La clase que herede de una clase abstracta debe declarar e implementar el comportamiento de los métodos abstractos (a menos que ésta, la que hereda, sea abstracta).**

**De no declararse (implícitamente), el compilador generará un error indicando que no se han implementado todos los métodos abstractos y que, o bien se implementen, o bien se declare la clase como abstracta.**

## Ejemplo 14

```
public abstract class Polygon {  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

```
public class Triangle extends Polygon {  
    private float base, height, sideA, sideB, sideC;  
    public Triangle(){  
    }  
    public Triangle(float base, float height){  
        this.base = base;  
        this.height = height;  
    }  
    public final double getArea(){  
        return (base * height)/2;  
    }  
    public double getPerimeter(){  
        return sideA + sideB + sideC;  
    }  
}
```

**Es posible crear referencias a una clase abstracta:**

**Polygon figure;**

**Sin embargo, no es posible crear una instancia (objeto) de una clase abstracta.**

**Polygon figure = new Polygon();**

**El que una clase abstracta no se pueda instanciar es coherente con su descripción dado que este tipo de clases no tiene completa su implementación y encaja bien con la idea de que un ente abstracto no puede materializarse.**

**Sin embargo, se puede realizar el llamado up-casting, un objeto se comporta como un elemento base:**

```
Polygon figure = new Triangle();
figure.getPerimeter();
```

**El up-casting es automático, es decir, no se requiere hacer un casting explícito para indicar el moldeo.**

## Ejemplo 14

```
public class GeometricFigures {  
    public static void main (String [] args){  
        Polygon figure;  
        //figure = new Polygon();  
        figure = new Triangle();  
        System.out.println(figure.getPerimeter());  
    }  
}
```

## Clase final

Una clase que representa el final de la jerarquía de clases, es decir, es un elemento suficientemente específico que no necesita modificarse. Cuando esto ocurre el funcionamiento de la clase se puede proteger evitando ser modificada por otras clases.

En java para evitar que una clase pueda ser modificada se agrega la palabra reservada final a la declaración de la misma, consiguiendo que ésta no pueda ser heredada y, por ende, modificada.

```
public final class IsoscelesTriangle extends Triangle {  
    public double getPerimter(){  
        return 3 * sideA;  
    }  
}
```



// Esta clase no compilaría porque no se puede  
// heredar de una clase final

```
public class InheritIsoscelesTriangle extends IsoscelesTriangle { }
```

## **Clase interna (inner class)**

**Una clase puede ser etiquetada como privada, protegida o estática únicamente si es un miembro de otra clase. A estas clases se les conoce como clases internas.**

**Una clase externa solo se puede crear con un modificador de acceso público (public) o sin modificador (default).**

**Por lo tanto, una clase interna (inner o nested class) es un clase definida dentro de otra clase es decir, una clase que es, a su vez, miembro de otra clase.**

```
class Externa {  
    // Comentarios  
    // Atributos  
    // Métodos  
    class Interna {  
        // Comentarios  
        // Atributos  
        // Métodos  
    }  
}
```



**Normalmente, en este tipo de clases el objeto de la clase interna depende directamente de la clase externa, es decir, los objetos de la clase interna deben ser creados a partir de una instancia de la clase externa.**

**Una instancia de la clase interna tiene acceso a todos los datos miembro de la clase que lo contiene (clase externa) sin utilizar un calificador de acceso especial, es decir, como si le pertenecieran.**



**Las clases internas se clasifican en 4 tipos:**

- **Clases internas estáticas (clases anidadas de alto nivel).**
- **Clases internas miembro.**
- **Clases internas locales.**
- **Clases internas anónimas.**

## **Clase interna estática**

**Las clases internas estáticas, obviamente, debe declararse como clases estáticas.**

**Para acceder a las clases estáticas internas no es necesario crear una referencia (instancia) de la clase externa, sólo hay que especificar donde se encuentra la clase interna estática, es decir, especificar la clase que la contiene.**



**Desde la clase estática interna se puede acceder a los miembros estáticos de la clase externa. Para acceder a los miembros de instancia de la clase externa es necesario crear una referencia dentro de la clase interna hacia la clase externa.**

## Ejemplo 16

```
public class OuterClass {  
  
    static public int i = 5;  
  
    public void outerMethod() {  
        System.out.println("Inside outer class i = " + i);  
    }  
  
    public static class InnerStatic {  
        static int a = 25;  
        public static void main(String args[]) {  
            System.out.println("Inside inner class i = " + i);  
            System.out.println("Inside inner class a = " + a);  
            OuterClass oc = new OuterClass();  
            oc.outerMethod();  
        }  
    }  
}
```

## **Clase interna miembro**

**Estas clases se definen como miembro (no estático) de la clase contenedora. Se pueden declarar, incluso, como privadas o protegidas.**

**A cada instancia de la clase externa (contenedora) se asocia una instancia de la clase interna miembro. La clase interna miembro tiene acceso a todos los atributos y métodos de la clase externa, incluyendo los privados.**



**Una clase interna miembro no puede tener miembros estáticos.  
Tampoco debe tener nombres comunes con la clase externa.**

**La única forma para acceder a la clase interna es creando una  
referencia a la clase externa.**

**Para las clases internas, la palabra reservada this hace referencia a la clase contenedora: ClaseExterna.this.**

**Para crear una instancia de una clase interna se utiliza la clase externa: InstanciaClaseExterna.newInstance(ClaseInterna).**

**Si se hereda de una clase interna miembro y se quiere acceder a la clase base, es necesario hacer referencia a la instancia de la clase externa: InstanciaClaseExterna.super().**

## Ejemplo 17

```
public class ExternaMiembro {  
    public final int TAM = 5;  
    Object arreglo[] = new Object[TAM];  
    int cima = 0;  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
    public boolean isEmpty(){  
        return cima == 0;  
    }  
    class InternaMiembro{  
        public boolean hasMoreElements(){  
            return !isEmpty();  
        }  
  
        public Object nextElement(){  
            return getObject();  
        }  
    }  
}
```

## Ejemplo 17

```
public class PruebaExternaMiembro {  
    public static void main (String [] args){  
        ExternaMiembro pila = new ExternaMiembro();  
  
        for (int i = 0 ;i < pila.arreglo.length ;i++){  
            String txt = "Objeto " + i;  
            pila.setObject(txt);  
        }  
  
        ExternaMiembro.InternaMiembro enumerador;  
        enumerador = pila.new InternaMiembro();  
  
        while(enumerador.hasMoreElements()){  
            String txt = (String) enumerador.nextElement();  
            System.out.println(txt);  
        }  
    }  
}
```

## Clase interna local

Estas clases también llamadas clases internas de métodos locales, se definen dentro del bloque de código de un método, por lo tanto, solo se pueden utilizar dentro del código donde están declaradas.



**Pueden hacer uso de las variables locales y los parámetros del método declarados como final. No utilizan ningún modificador de acceso. No pueden ser estáticas.**

**Estas clases no están disponibles de manera pública, por lo tanto son inaccesibles.**

## Ejemplo 18

```
public class ExternaLocal {  
    public final int TAM = 5;  
    Object arreglo[] = new Object[TAM];  
    int cima = 0;  
  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
  
    public boolean isEmpty(){  
        return cima == 0;  
    }  
}
```

## Ejemplo 18

```
public void getElements(final Object objetos[]){
    class InternaLocal{
        public boolean hasMoreElements(){
            return !isEmpty();
        }

        public Object nextElement(){
            return getObject();
        }
    }
    InternaLocal enumerador = new InternaLocal();
    while(enumerador.hasMoreElements()){
        String txt = (String) enumerador.nextElement();
        System.out.println(txt);
    }
}
```

## Ejemplo 18

```
public class PruebaExternaLocal {  
    public static void main (String [] args){  
        ExternaLocal pila = new ExternaLocal();  
  
        for (int i = 0 ;i < pila.arreglo.length ;i++){  
            String txt = "Objeto " + i;  
            pila.setObject(txt);  
        }  
  
        pila.getElements(pila.arreglo);  
    }  
}
```

## **Clase interna anónima**

**En esencia, las clases anónimas son clases internas locales sin nombre, de ahí que sea anónima.**

**Se define al mismo tiempo al que se crea la instancia y, por tanto, sólo puede existir una instancia de una clase anónima.**

**Los constructores de estas clases deben ser sencillos (sin argumentos), para evitar anidar demasiado código en una sola línea.**

## Ejemplo 19

```
public class ExternaAnonima {  
    public final int TAM = 5;  
    Object arreglo[] = new Object[TAM];  
    int cima = 0;  
  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
  
    public boolean isEmpty(){  
        return cima == 0;  
    }  
}
```

## Ejemplo 19

```
public void getElements(final Object objetos[]) {  
    abstract class InternaLocal {  
        public abstract boolean hasMoreElements();  
        public abstract Object nextElement();  
    }  
  
    InternaLocal enumerador = new InternaLocal(){  
        @Override  
        public boolean hasMoreElements(){  
            return !isEmpty();  
        }  
        @Override  
        public Object nextElement(){  
            return getObject();  
        }  
    };  
    while(enumerador.hasMoreElements()){  
        String txt = (String) enumerador.nextElement();  
        System.out.println(txt);  
    }  
}
```

## Ejemplo 19

```
public class PruebaExternaAnonima {  
    public static void main (String [] args){  
        ExternaAnonima pila = new ExternaAnonima();  
  
        for (int i = 0 ;i < pila.arreglo.length ;i++){  
            String txt = "Objeto " + i;  
            pila.setObject(txt);  
        }  
  
        pila.getElements(pila.arreglo);  
    }  
}
```

**En resumen, los tipos de clases que existen en Java son:**

**Publicas o sin modificador:** Son las más comunes, accesibles desde cualquier otra clase en el mismo paquete (de otro modo hay que importarlas).

**Abstractas:** Son clases bases que dan las bases para que rigen el comportamiento de las clases que las heredan.

**Finales:** Son las que terminan la cadena de herencia. Útiles por motivos de seguridad y eficiencia de un programa, ya que no permiten crear más sub-divisiones por debajo de ellas.

**Internas:** Estas clases sólo se pueden crear dentro de otra clase, es decir, una clase interna es una clase miembro de otra clase.

## 3.7 INTERFACES.



**El concepto de Interfaz lleva un paso más adelante la idea de las clases abstractas. Una interfaz es una clase abstracta pura, es decir, una clase donde todos los métodos son abstractos (no se implementa ninguno).**

**Este tipo de diseños permiten establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno), pero no bloques de código.**

**Una interfaz puede contener atributos, pero estos son siempre públicos, estáticos y finales.**

**Para crear una interfaz, se utiliza la palabra reservada interface en lugar de la palabra reservada class. La interfaz puede definirse publica o sin modificador de acceso, y tiene el mismo significado que para las clases.**

**Todos los métodos que declara una interfaz son siempre públicos y abstractos.**

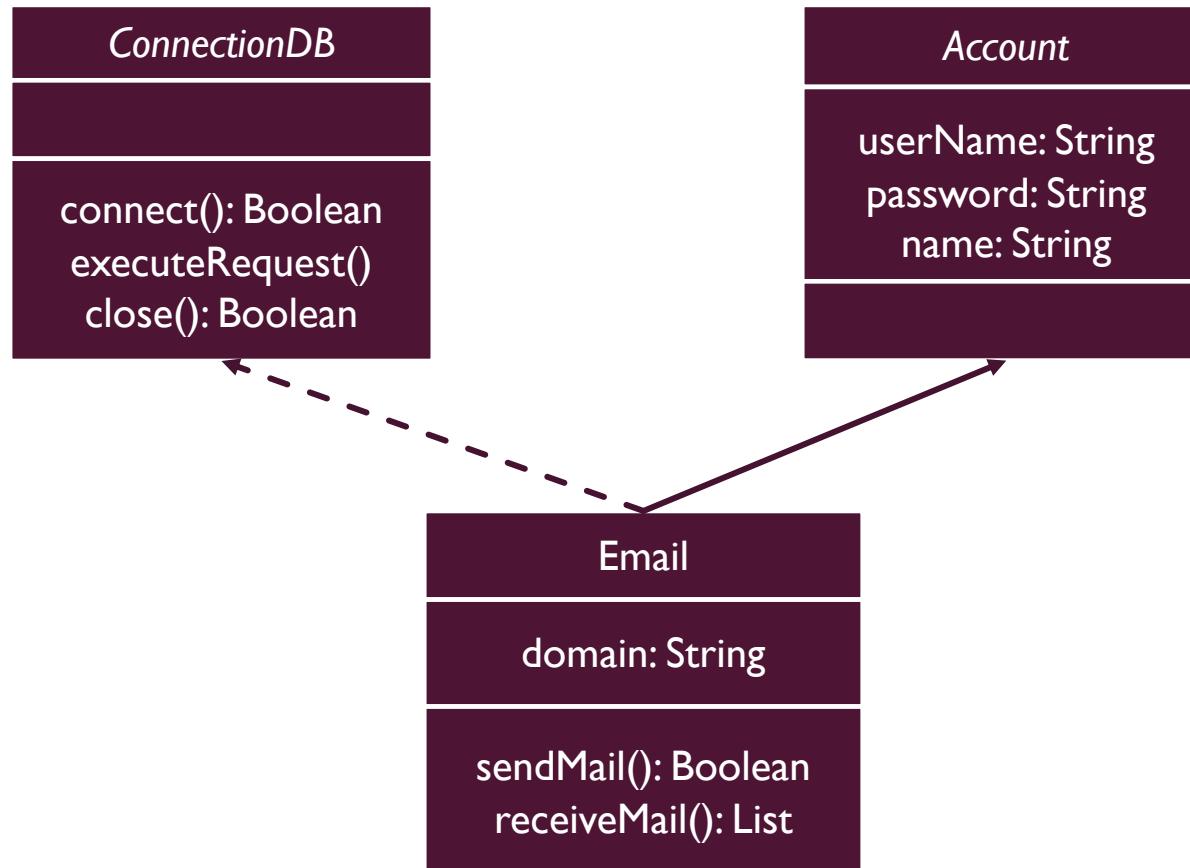
Para indicar que una clase desea utilizar los métodos de una interfaz se utiliza la palabra reservada `implements`. El compilador se encargará de verificar que la clase que implementa una interfaz efectivamente declare e implemente todos los métodos de la interfaz. Una clase puede implementar más de una interfaz.

La sintaxis general para declarar una interfaz es la siguiente:

```
interface NombreInterfaz {  
    tipoRetorno nombreMetodo([Parametros]);  
}
```

**Una cuenta obtiene la información de un repositorio de datos. Por lo tanto ocupa operaciones básicas para extraer e insertar información. Estas operaciones se deben proveer desde un punto de vista general, sin importar el tipo de repositorio donde se encuentra la información (una base de datos, un archivo de texto, el disco duro, un arreglo de discos en red, etc.).**

### Diagrama de clases de los objetos Cuenta y Correo electrónico:



## Ejemplo 20

---

```
public interface ConnectionDB {  
    Boolean connect();  
    void executeRequest();  
    Boolean close();  
}
```

## Ejemplo 20

---

```
public class Account {  
    private String userName;  
    private String password;  
    private short age;  
  
    public Account() {}  
  
    public Account(String userName, String password, short age) {  
        this.setUserName(userName);  
        this.setPassword(password);  
        this.setAge(age);  
    }  
    public String getUserName(){  
        return userName;  
    }  
    public String getPassword(){  
        return password;  
    }  
}
```

## Ejemplo 20

```
public short getAge(){
    return age;
}
public void setUserName(String userName){
    this.userName = userName;
}
public void setPassword(String password){
    this.password = password;
}
public void setAge(short age){
    if (age > 17 && age < 130) {
        this.age = age;
    }
}
public String toString() {
    return "User name=" + getUserName() +
        "\nPassword=*****" + "\nAge=" + getAge();
}
```

## Ejemplo 20

```
public class Email extends Account implements ConnectionDB {  
    private String domain;  
  
    public Email(){  
  
        public Email(String userName, String password, short age, String domain){  
            super (userName, password, age);  
            this.setDomain(domain);  
        }  
        public void setDomain(String domain){  
            this.domain = domain;  
        }  
        public String getDomain(){  
            return domain;  
        }  
        public String toString(){  
            return super.toString() + "\nDomain=" + getDomain();  
        }  
}
```

## Ejemplo 20

```
public Boolean connect(){
    System.out.println("Conectando con la BD...");
    return true;
}

public void executeRequest(){
    System.out.println("Se ejecuta la consulta.");
}

public Boolean close(){
    System.out.println("Cerrando la conexión a la BD...");
    return true;
}
```

## Ejemplo 20

```
public class TestEmail {  
    public static void main (String [] args){  
        Email mail = new Email("user1", "qwerty", (short)22, "gmail.com");  
        System.out.println(mail);  
        mail.connect();  
        mail.executeRequest();  
        mail.close();  
    }  
}
```

Al igual que con las clases abstractas, se pueden crear referencias de una interfaz, pero no es posible instanciar una interfaz.

```
// La instancia directa provoca un error  
ConnectionDB mail = new ConnectionDB();
```

Sin embargo, una referencia de una interfaz puede ser asignada a cualquier objeto que la implemente.

```
ConnectionDB mail = new Email();
```

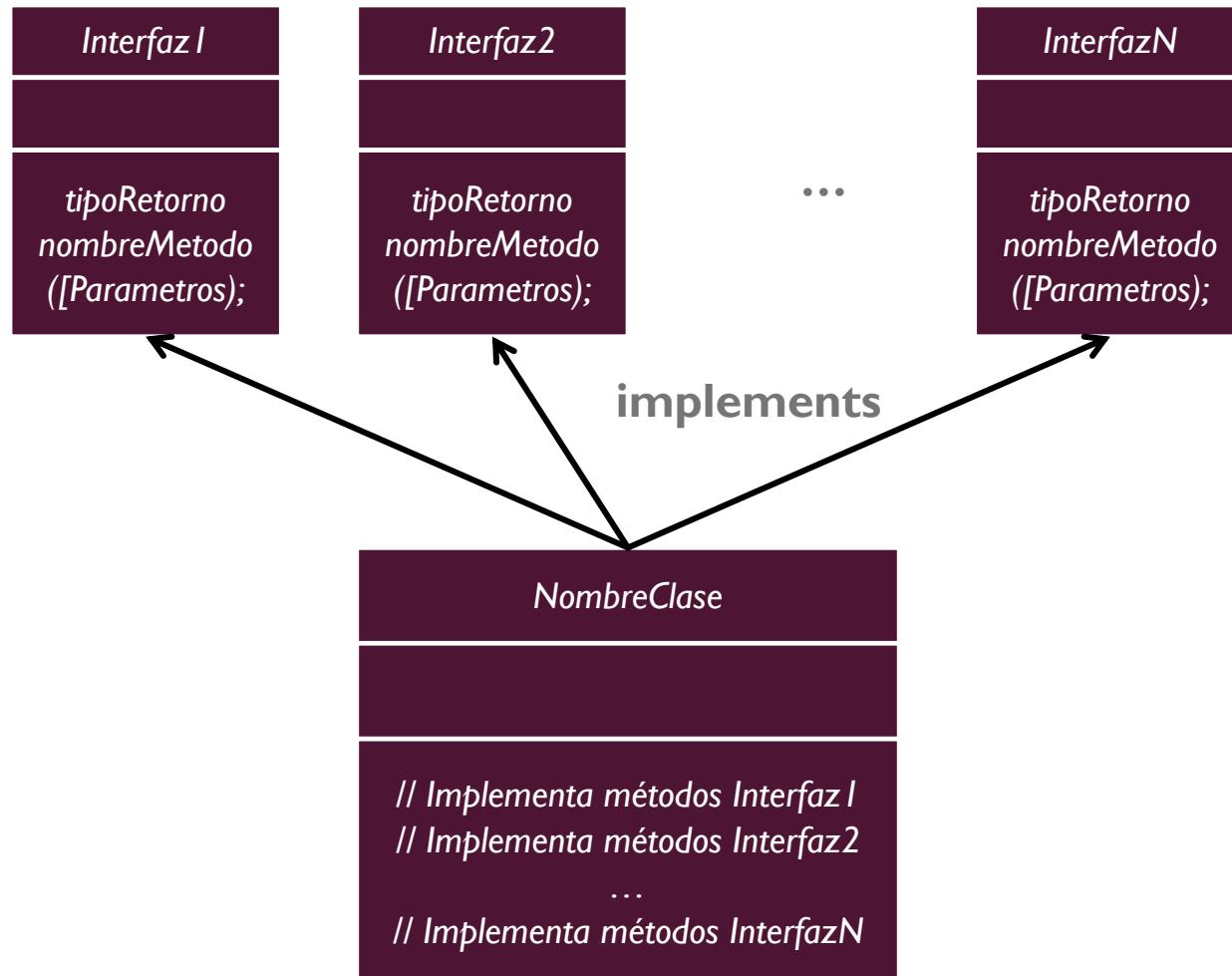
## Ejemplo 21

```
public class TestEmail {  
    public static void main (String [] args){  
        ConnectionDB mail;  
        mail = new Email("user1", "qwerty", (short)22, "gmail.com");  
        System.out.println(mail);  
        mail.connect();  
        mail.executeRequest();  
        mail.close();  
    }  
}
```

**Una clase puede implementar ene interfaces separadas por coma, la única restricción es que, dentro del cuerpo de la clase, se deben implementar todos los métodos de cada una de las interfaces.**

```
class NombreClase implements Interfaz1, Interfaz2, ..., InterfazN {  
    // Métodos de la Interfaz1  
    // Métodos de la interfaz 2  
    // ...  
    // Métodos de la interfaz N  
}
```

**El orden de la implementación de las interfaces y el orden en el que se implementan los métodos de las interfaces dentro de la clase no es relevante.**



Las interfaces pueden heredar de otras interfaces y, a diferencia de las clases, una interfaz puede heredar de más de una interfaz (herencia múltiple).

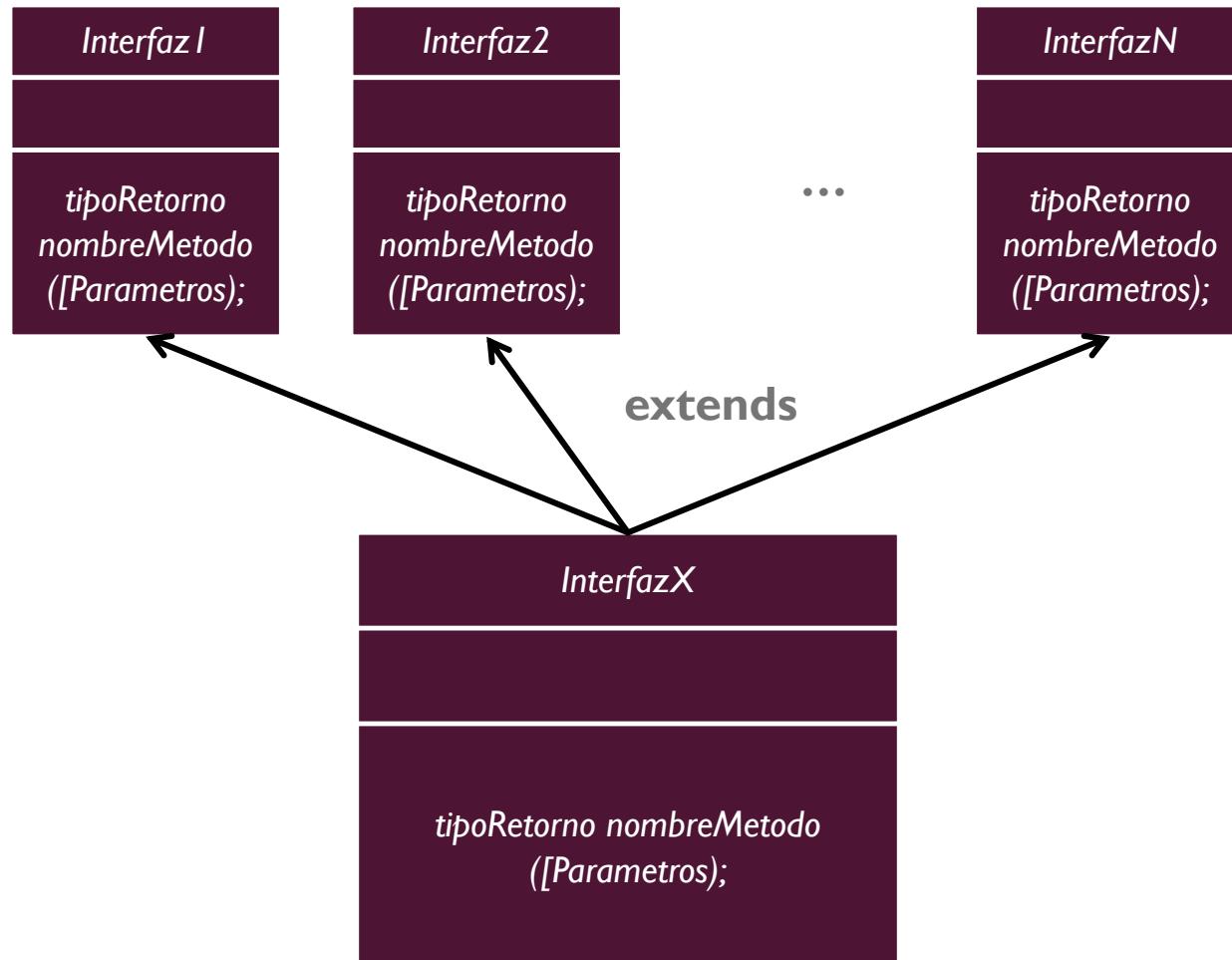
La sintaxis para heredar entre interfaces es la siguiente:

```
interface Interfaz extends Interfaz1, Interfaz2, ..., InterfazN{  
    tipoRetorno nombreMetodo([Parametros]);  
}
```

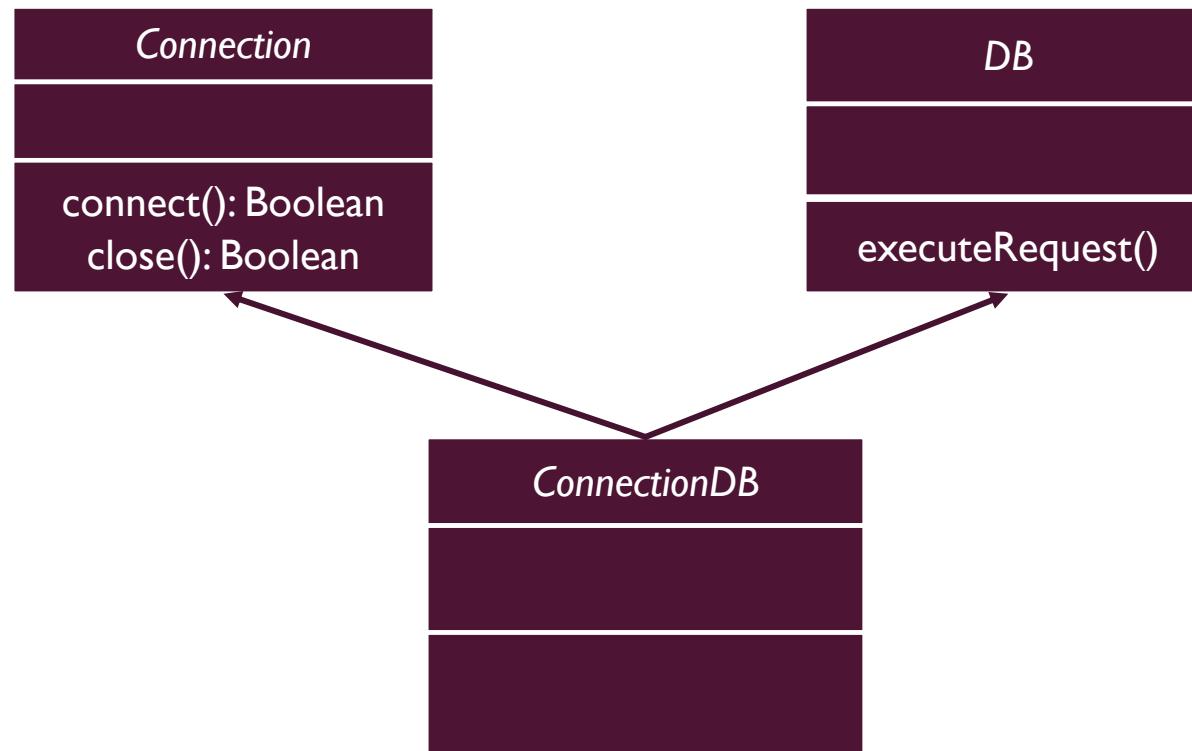


**Igual que en las clases, la interfaz que hereda de otras interfaces posee todos los métodos definidos en ellas.**

**Por lo tanto, la clase que implemente esta interfaz (la interfaz que hereda de otras interfaces) debe definir los métodos de todas las interfaces.**



### Diagrama de clases de las interfaces Connection, DB y ConnectionDB:



## Ejemplo 22

```
public interface Connection{  
    Boolean connect();  
    Boolean close();  
}
```

```
public interface DB {  
    void executeRequest();  
}
```

```
public interface ConnectionDB extends Connection, DB { }
```

**Dado que, por definición, todos los datos miembros (atributos) que se definen en una interfaz son estáticos y finales, resulta un buen lugar para implantar grupos de constantes.**

**Debido a que las interfaces no se pueden instanciar y a que los atributos son estáticos, el acceso a éstos se debe realizar a través del nombre de la interfaz.**

```
public interface Months {  
    int ONE = 1, TWO = 2, THREE = 3;  
    int FOUR = 4, FIVE = 5, SIX = 6;  
    int SEVEN = 7, EIGHT = 8, NINE = 9;  
    int TEN = 10, ELEVEN = 11, TWELVE = 12;  
  
    String [] MONTHS_NAME = {"",  
        "Enero", "Febrero", "Marzo",  
        "Abril", "Mayo", "Junio",  
        "Julio", "Agosto", "Septiembre",  
        "Octubre", "Noviembre", "Diciembre"};  
}
```

## Ejemplo 23

```
public class TestMonths {  
    public static void main (String [] args){  
        System.out.println("Los nombres de los meses son:");  
        for (String month : Months.MONTHS_NAME){  
            System.out.println(month);  
        }  
  
        System.out.println("\nMes elegido:");  
        System.out.println(Months.MONTHS_NAME[Months.SEVEN]);  
    }  
}
```

# PROGRAMACIÓN ORIENTADA A OBJETOS





**‘OOP does model the real world! Things at the top of the hierarchy seemingly do nothing but tell those at the bottom what to do.’**

**@fogus**  
**Author of The Joy of Clojure.**

# PRINCIPIOS SOLID

DISEÑO DE CLASES



**Las clases son los bloques de construcción de una aplicación orientada a objetos. Por lo tanto, si estos bloques no son suficientemente robustos, la construcción (aplicación) pasará problemas en tiempos futuros.**

**Una clase mal diseñada generará situaciones complejas de resolver cuando el alcance de la aplicación cambie.**



**Por ende, un conjunto de clases bien diseñadas y escritas puede agilizar el proceso codificación y reducir el número de errores (bugs) en la aplicación.**

**Para poder crear clases bien diseñadas se deben tener en mente los principios **SOLID**, los cuales son parte de las mejores prácticas para el diseño de aplicaciones.**

## **Single Responsibility Principle**

**El principio de responsabilidad única dicta que una clase debe tener una y sólo una responsabilidad.**

**Una clase debe tener un solo propósito. Si es una clase modelo debe representar solo un actor / entidad. Esto permite flexibilidad, debido a que se puede modificar la clase sin preocuparse del impacto de los cambios en otra entidad.**



**De manera similar, para una clase de servicio / administración, ésta solo debe contener los métodos a llamar y nada más. Tampoco deben contener funciones globales de acceso público relacionadas a un módulo. Es mejor separar estas funciones en una clase particular.**

**Lo anterior permite que la clase conserve su propósito particular, además de proteger el acceso a módulos específicos de la clase.**

## **Open Closed Principle**

**El principio de abierto-cerrado dicta que los componentes de software deben estar abiertos para extender su comportamiento, pero se deben cerrarse a modificaciones.**

**Las clases deben estar diseñadas de tal manera que si algún desarrollador necesita cambiar el comportamiento de un método, lo único que tenga que hacer sea heredar la clase y sobrescribir cierta función.**

## **Liskov's Substitution Principle**

**El principio de sustitución de (Barbara) Liskov es una variación del principio anterior, dicta que los tipos derivados deben ser sustituibles por sus tipos base.**

**Continuando el principio anterior, si un desarrollador crea una clase a partir de una clase base, la clase derivada debería ser capaz de ajustarse a la aplicación sin provocar fallas o errores fatales.**



**Este principio se puede asegurar siguiendo la primera regla. Así, si la clase base hace estrictamente una cosa, lo peor que puede pasar es que lo que sobrescriba el desarrollador en la clase derivada haga que falle una característica, pero no toda la aplicación.**

## **Interface Segregation Principle**

**El principio de segregación de interfaces dicta que los clientes no deben ser forzados a implementar métodos innecesarios, los cuales no van a utilizar.**

**Este principio está muy unido al de responsabilidad única y se enfoca en el diseño adecuado y general de las interfaces para evitar métodos ociosos.**



**Suponiendo que una interfaz define dos tipos de impresiones, una en Word y otra en PDF, ¿qué pasa si un desarrollador quiere imprimir en PDF pero no en Word? ¿Debe sobrescribir el método Word, aunque éste se quede vacío?**

**Lo que dicta este principio es que las impresiones se deben separar en dos interfaces de tal manera que sean suficientemente funcionales y generales.**

## **Dependency Inversion Principle**

**El principio de inversión de dependencia dicta que se debe depender de lo abstracto, no de lo concreto.**

**Una aplicación se debe diseñar de tal manera que varios módulos puedan estar separados entre sí, pero se puedan unir a través de una capa abstracta.**



**Los principios SOLID para el diseño de clases forman las mejores prácticas a seguir para crear las clases de una aplicación.**

Nombre del principio	¿Qué dicta?
Single Responsibility	Una clase debe tener una y sólo una razón.
Open Closed	Los componentes de software se deben abrir para poder extender su comportamiento, pero se deben cerrar a las modificaciones.
Liskov's Substitution	Los tipos derivados deben poder ser sustituidos por sus tipos base.
Interface Segregation	Los clientes no deben ser forzados a implementar métodos innecesarios, los cuales no van a utilizar.
Dependency Inversion	Dependencia en lo abstracto, no en lo concreto.

**Principios SOLID para el diseño de clases.**



## 3.8 PAQUETES Y DOCUMENTACIÓN.

# PAQUETES

Las clases de las bibliotecas estándar del lenguaje están organizadas en jerarquías de paquetes. Esta organización en jerarquías ayuda a que las personas encuentren clases particulares que requieren utilizar.

Está bien que varias clases tengan el mismo nombre si están en paquetes distintos. Así, encapsular grupos pequeños de clases en paquetes individuales permite reusar el nombre de una clase dada en diversos contextos.

**Un paquete es una agrupación de clases. La API de Java cuenta con muchos paquetes, que contienen clases agrupadas bajo un mismo propósito, por ello, es necesaria una estructura en la organización de la biblioteca.**

**Java utiliza paquetes para organizar las clases de la biblioteca en grupos que permanecen juntos. Las clases del API están organizadas en jerarquías de paquetes.**

## **Los paquetes se utilizan con las finalidades siguientes:**

- **Para agrupar clases relacionadas.**
- **Para evitar conflictos de nombres. En caso de conflicto de nombres entre clases el compilador obliga diferenciarlos usando su nombre cualificado.**
- **Para ayudar en el control de la accesibilidad de clases y miembros.**



**Un paquete, por tanto, es una agrupación de clases afines, equivalente al concepto de librería existente en otros lenguajes.**

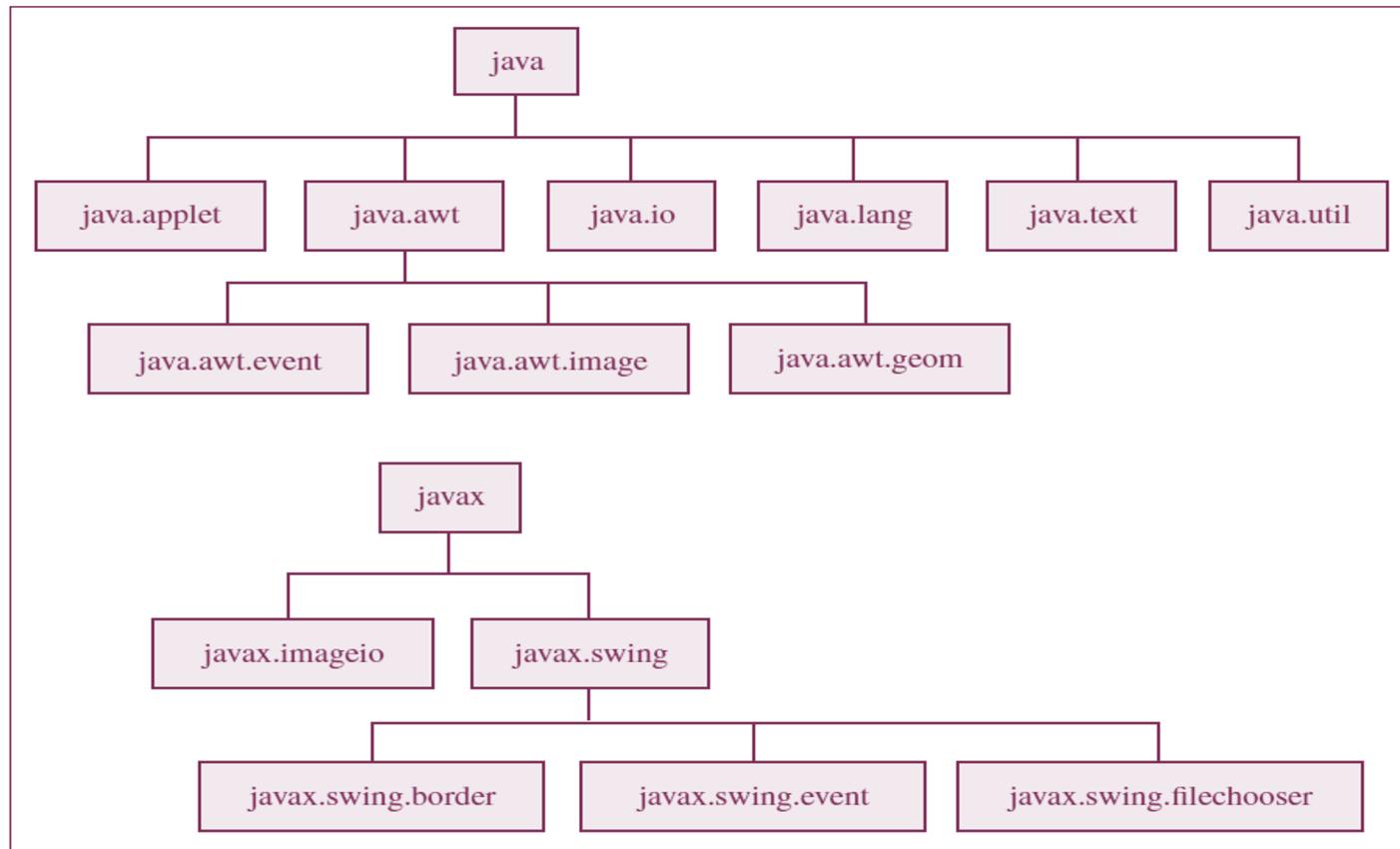
**Una clase puede agruparse en un paquete y puede usar otras clases definidas en ese o en otros paquetes.**

**Los paquetes delimitan el espacio de nombres (space name) de las clases. El nombre de una clase debe ser único dentro del paquete donde se define.**



**El nombre completo o nombre calificado (Fully Qualified Name) de una clase debe ser único y está formado por el nombre de la clase precedido por los nombres de los sub-paquetes en donde se encuentra hasta llegar al paquete principal, separados por puntos.**

**Por lo tanto, dos clases con el mismo nombre en dos paquetes distintos pueden coexistir e incluso pueden ser usadas en el mismo programa.**



**Figura. Jerarquía de paquetes de la API de Java.**

**En Java para definir que una clase pertenece a un paquete en específico se utiliza la palabra reservada package, esta es su sintaxis:**

```
package nombrepaquete;
```

**La cláusula package debe ser la primera sentencia del archivo fuente. Por lo tanto, cualquier clase declarada dentro de ese archivo pertenece al paquete indicado.**

**La cláusula package es opcional, si no se utiliza las clases declaradas en el archivo fuente simplemente no pertenecen a ningún paquete en concreto (pertenecen a un paquete por defecto sin nombre).**

**Los paquetes se utilizan para organizar las clases y, por ende, mantener en una ubicación común las clases relacionadas. También resultan importantes para el alcance de los modificadores de acceso vistos anteriormente.**

Por ejemplo, si se define la siguiente clase:

```
package aeropuerto;  
  
public class Aerolinea {  
    List<Avion> flotillaAviones;  
}
```

El compilador y la máquina virtual asumen que la clase Avion pertenece al paquete aeropuerto.

**Si la clase Avion no pertenece al mismo paquete, es necesario hacer accesible el espacio de nombres donde se define la clase Avion en la clase Aerolinea. Esto se realiza mediante el uso de la clausula import.**

```
package aeropuerto;  
import flotilla.*;  
  
public class Aerolinea {  
    List<Avion> flotillaAviones;  
}
```

**Con la sentencia import flotilla.\*; se hacen accesibles todas las clases que pertenecen al paquete flotilla. Es importante aclarar que \* solo importa las clases de ese paquete, es decir, si existe un sub paquete dentro del paquete, esas clases no se importarían.**

**Por otro lado, si únicamente se quisiera importar la clase Avion, se podría utilizar la sentencia: import flotilla.Avion;**

También es posible hacer accesibles las clases de un paquete sin utilizar la clausula import y en tiempo de ejecución de la siguiente manera:

```
package aeropuerto;  
  
public class Aerolinea {  
    List<flotilla.Avion> flotillaAviones;  
}
```

Sin embargo, de esta manera cada vez que se genere una instancia de la clase Avion hay que especificar el nombre completo de la misma.

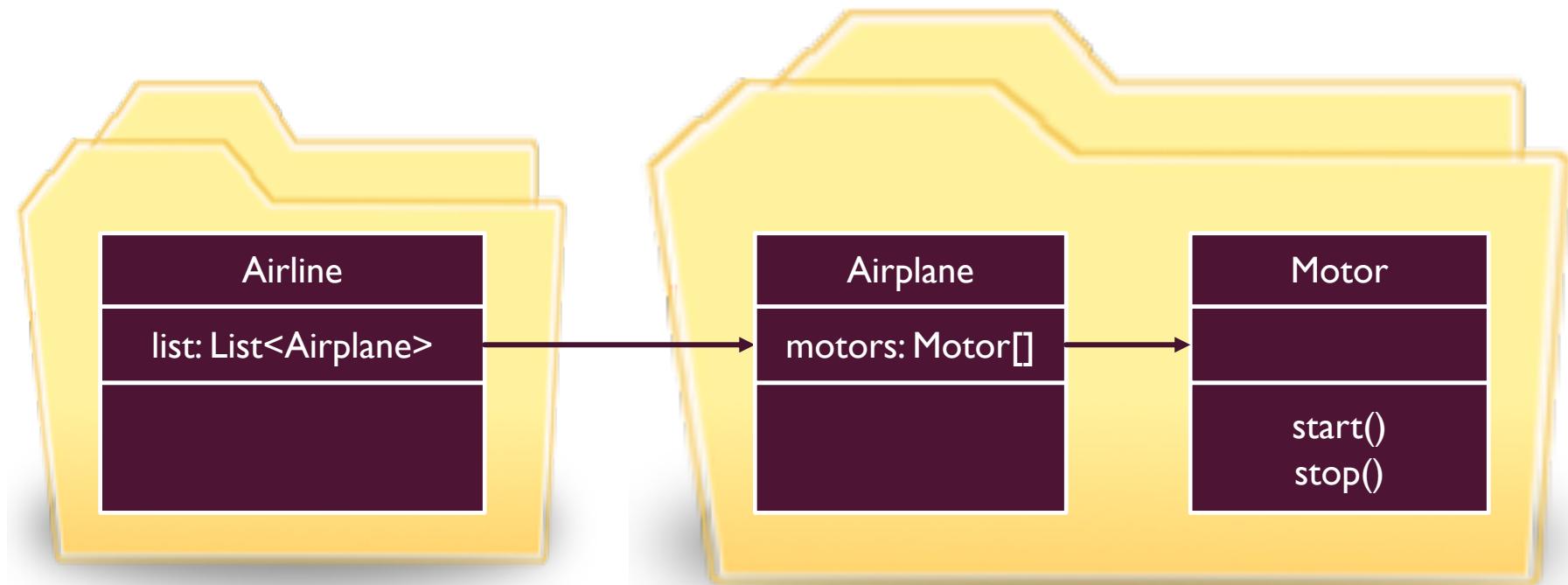
**Los paquetes que utilizan nombres compuestos se separan por puntos, de manera similar a como se componen las direcciones URL, por ejemplo:**

```
package flotilla.aviones;
```

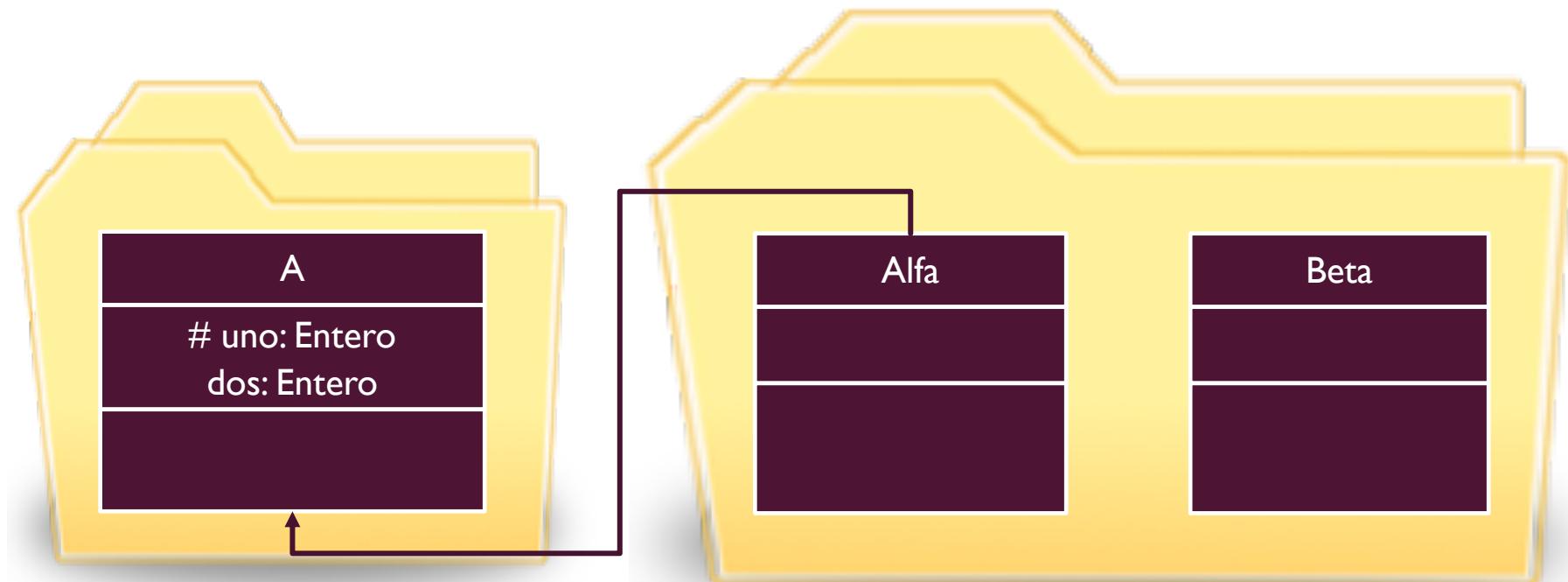
**El API de java está estructurado de esta forma, con un primer calificador (java o javax) que indica la base, un segundo calificador (awt, util, swing, etc.) que indica el grupo funcional de las clases y subpaquetes en un tercer nivel, dependiendo de la amplitud del grupo.**

**Los paquetes también tienen un significado físico ya que sirven para almacenar los módulos ejecutables (archivos con extensión .class) en el sistema de archivos de la máquina.**

### Diagrama de clases de la Aerolínea, el Avión y el Motor:



### Diagrama de clases de la Aerolínea, el Avión y el Motor:



## Ejemplo 24

---

```
package airport.cdmx;

import flotilla.airplane.Airplane;
import java.util.List;
import java.util.ArrayList;

public class Airline {
    public List<Airplane> airplanes = new ArrayList<Airplane>();
    public Airline(){}
}
```

```
package flotilla.airplane;

public class Airplane {
    public Motor [] motors;
    public Airplane(){}
    public Airplane(int numMotors){
        motors = new Motor[numMotors];
        for (int cont=0 ; cont<numMotors ; cont++){
            motors[cont] = new Motor();
        }
    }
}
```

## Ejemplo 24

---

```
package flotilla.airplane;

public class Motor {
    public void start(){
        System.out.println("Arranca motor.");
    }
    public void stop(){
        System.out.println("Detiene motor.");
    }
}
```

```
package test;

import airport.cdmx.Airline;
import flotilla.airplane.Airplane;
import flotilla.airplane.Motor;

public class TestAirline {
    public static void main (String [] args){
        Airline mexicana = new Airline();
        mexicana.airplanes.add(new Airplane(4));
        mexicana.airplanes.add(new Airplane(2));
        mexicana.airplanes.add(new Airplane(3));

        for (Airplane air : mexicana.airplanes){
            System.out.println(air);
            for (Motor m : air.motors){
                m.start();
                m.stop();
            }
        }
    }
}
```

**Para compilar una clase que pertenece a un paquete se utiliza la bandera -d seguido de la ruta donde se quiere crear el paquete:**

```
javac -d . NombreClase.java
```



**“Code never lies, comments sometimes do.”**

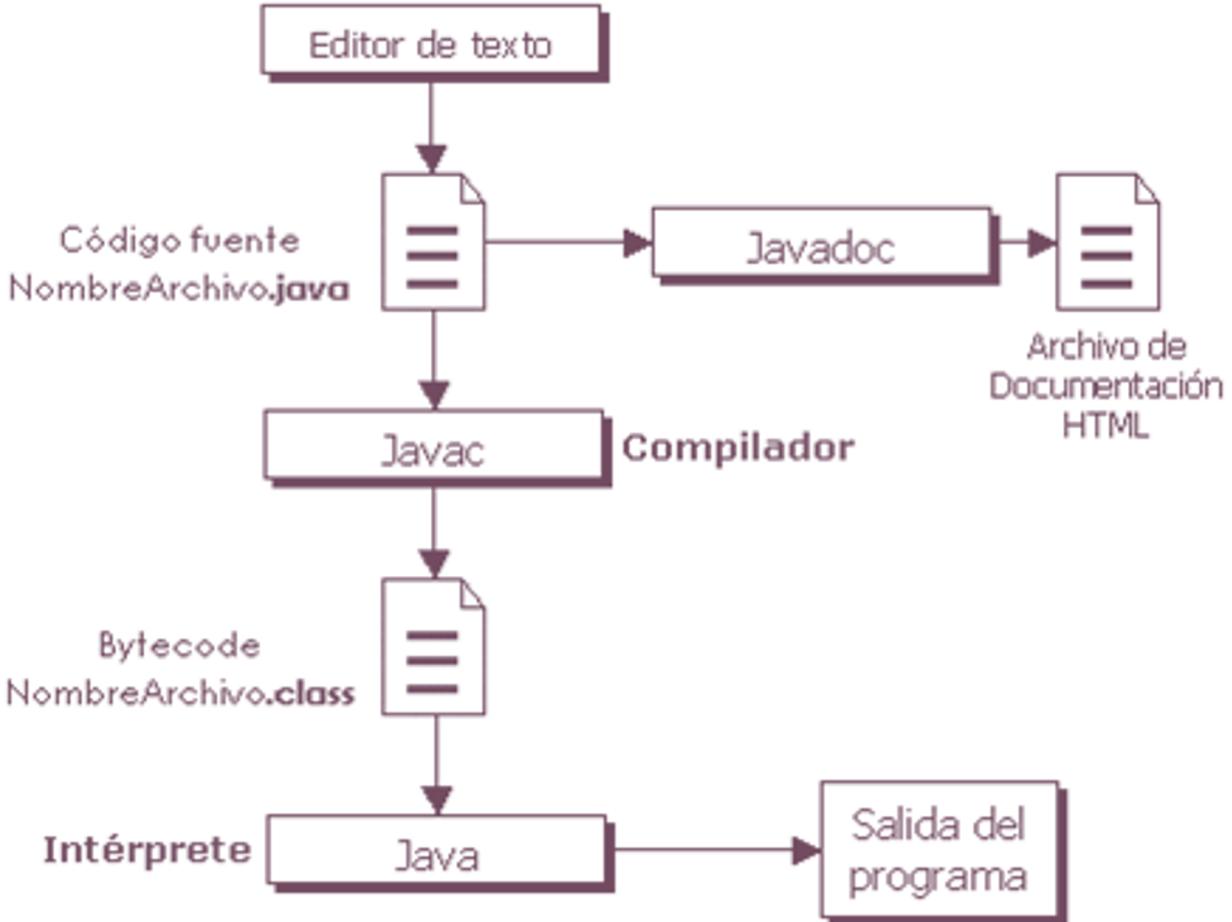
**Ron Jeffries**  
**(one of the three founders of the Extreme Programming  
software development methodology circa 1996)**

# DOCUMENTACIÓN

JDK proporciona una herramienta para generar la documentación de las clases creadas a partir de los comentarios incluidos en el código fuente.

Los comentarios de documentación deben empezar con `/**` y terminar con `*/`. Se pueden documentar clases, atributos (datos miembro) y métodos.

La documentación se genera en código HTML utilizando el comando javadoc.



## Existen etiquetas para documentar a través de javadoc.

<b>Etiqueta</b>	<b>Formato</b>	<b>Descripción</b>
Todos	@see	Permite crear una referencia a la documentación de otra clase o método.
Clases	@version	Comentario con datos indicativos del número de versión.
Clases	@author	Nombre del autor.
Clases	@since	Versión desde la que está presente la clase.
Métodos	@param	Parámetros que recibe el método.
Métodos	@return	Significado del dato devuelto por el método
Métodos	@exception	Comentario sobre las excepciones que lanza.
Métodos	@throws	Comentario sobre las excepciones que lanza.
Métodos	@deprecated	Indicación de que el método es obsoleto.

**La documentación se puede guardar en cualquier ruta. Para especificar la ruta de destino se utiliza la bandera -d.**

**javadoc -d ruta -version -author -use NombreClase.java**

**Es posible generar la documentación de todas las clases (\*.java) que se encuentran en la ruta actual. La documentación se genera en HTML.**

## Ejemplo 25

```
package airport.cdmx;
import flotilla.airplane.Airplane;
import java.util.List;
import java.util.ArrayList;

/**
 * Clase que crea una flotilla de aviones para una aerol&iacute;nea.
 * @author Jorge A. Solano
 * @author jorge.a.solano@hotmail.com
 * @version 1
 * @see Visita: <a href="microsoft.fi-b.unam.mx" target=_blank>LMSR</a>
 */

public class Airline {
    public List<Airplane> airplanes = new ArrayList<Airplane>();

    /**
     * Constructor sin argumentos
     * @param Ninguno
     * @exception Ninguna
     */
    public Airline(){}
}
```

## Ejemplo 25

```
package flotilla.airplane;

/**
 * Clase que crea un avión con cierto número de motores
 * @author Jorge A. Solano
 * @author jorge.a.solano@hotmail.com
 * @version 1
 * @see Visita: <a href="microsoft.fi-b.unam.mx" target=_blank>LMSR</a>
 */
public class Airplane {
    public Motor [] motors;

    /** M&acute;todo constructor sin argumentos
     * @param Ninguno
     * @exception Ninguna
     */
    public Airplane(){}

    /** M&acute;todo constructor que inicia los motores del avión
     * @param numMotores que indica el número de motores del avión
     * @throws Nada
     */
    public Airplane(int numMotors){
        motors = new Motor[numMotors];
        for (int cont=0 ; cont<numMotors ; cont++){
            motors[cont] = new Motor();
        }
    }
}
```

## Ejemplo 25

```
package flotilla.airplane;
/**
 * Clase que crea un motor de avión.
 * @author Jorge A. Solano
 * @author jorge.a.solano@hotmail.com
 * @version 1
 * @see Visita: <a href="microsoft.fi-b.unam.mx" target=_blank>LMSR</a>
 */
public class Motor {
    /** Método que permite iniciar el motor
     * @param Ninguno
     * @return Ninguno
     * @exception Ninguna
     */
    public void start(){
        System.out.println("Arranca motor.");
    }
    /** Método que permite detener el motor
     * @param Ninguno
     * @return Ninguno
     * @throws Nada
     */
    public void stop(){
        System.out.println("Detiene motor.");
    }
}
```

## Ejemplo 25

The screenshot shows a Java API documentation interface with two main sections:

**Top Section (Left Column):**

- All Classes
- Packages
  - airport.cdmx
  - flotilla.airplane

**Bottom Section (Right Column):**

**Top Bar:** OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP

**Navigation:** PREV NEXT FRAMES NO FRAMES

**Packages Section:**

Package	Description
airport.cdmx	
flotilla.airplane	

**Bottom Bar:** OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP

**Navigation:** PREV NEXT FRAMES NO FRAMES

**Realizar una aplicación para una jerarquía de clases con, por lo menos, 4 niveles, utilizando:**

- **Los cimientos del paradigma**
- **Los principios SOLID**
- **Todos los modificadores vistos**
- **Espacio de nombres**
- **Documentación**

# 3 Herencia y polimorfismo

**Objetivo:** Aplicar las diferentes propiedades de la programación orientada a objetos para la resolución de problemas.

- 3.1 Herencia.
- 3.2 Método constructor.
- 3.3 Polimorfismo (moldeado o casting entre tipos referencia o instancias).
- 3.4 Referencias a this y a la clase base.
- 3.5 Modificadores de acceso (encapsulamiento).
- 3.6 Tipos de clases: abstractas, comunes y finales.
- 3.7 Interfaces.
- 3.8 Paquetes y documentación.

# ANÁLISIS

- Para crear aplicaciones con el paradigma orientado a objetos es importante tener bases sólidas del paradigma de programación orientado a objetos.
- Para implementar aplicaciones en un lenguaje de programación orientado a objetos solo es necesario conocer bien un lenguaje, ya que se pueden encontrar similitudes entre los diferentes lenguajes, lo que hace más sencillo programar en uno u en otro.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace SP {
    public class Logica {
        SqlConnection con = new SqlConnection();
        SqlCommand comando = new SqlCommand();
        string servidor, tabla, seguridad, usuario, pwd;
        public string error;

        public Logica() {
            iniciaCadCon();
        }
    }
}
```

```
public void iniciaCadCon() {  
    servidor = "omega";  
    tabla = "AdventureWorks";  
    seguridad = "True";  
    usuario = "labmicrosoft";  
    pwd = "qIw2e3r4t5y6,k.l";  
}  
  
public void conexion() {  
    con.ConnectionString = "Data Source=" + servidor +  
        "; Initial Catalog=" + tabla + "; Persist Security Info=" +  
        seguridad + "; User ID=" + usuario + "; Password=" + pwd;  
    con.Open();  
}
```

```
public DataSet buscarId(int id) {  
    try {  
        conexion();  
        comando.CommandType = CommandType.StoredProcedure;  
        comando.Connection = con;  
        comando.CommandText = "HumanResources.sp_getEmployeed";  
        comando.CommandTimeout = 10;  
        comando.Parameters.Clear();  
        comando.Parameters.AddWithValue("@EmployeedID", id);  
        SqlDataAdapter adapter = new SqlDataAdapter(comando);  
        DataSet ds = new DataSet();  
        adapter.Fill(ds);  
        con.Close();  
        return ds;  
    } catch (Exception e){  
        return null;  
    } finally {  
        if (con != null)  
            con.Close();  
    }  
}
```

```
public DataSet mostrarTodo() {
    try {
        conexion();
        comando.CommandType = CommandType.StoredProcedure;
        comando.Connection = con;
        comando.CommandText = "seleccionaTodo";
        comando.CommandTimeout = 10;
        comando.Parameters.Clear();
        SqlDataAdapter adapter = new SqlDataAdapter(comando);
        DataSet ds = new DataSet();
        adapter.Fill(ds);
        con.Close();
        return ds;
    } catch (Exception){
        return null;
    } finally {
        if (con != null)
            con.Close();
    }
}
}
```