

NOTAS DEL CURSO:

PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA

PROFESOR:

LUIS ALBERTO MUÑOZ GÓMEZ

Nota: varias imágenes pertenecen al curso SL-275 de Sun Microsystems, ed. 2002

Introducción al lenguaje

```
public class Introduccion {

    public static void main(String[] args) {
        System.out.println("Hola Mundo Java!");
        byte un=127;
        short dos=32767;
        int cuatro=0x7FFFFFFF;
        long ocho=0x7FFFFFFFFFFFFFFFLL;
        float flotante=2.5F;
        double doble=96.08;
        final double PI=3.1416;
        boolean sonBienvenidos=true;
        System.out.println(un);
        System.out.println(dos);
        System.out.println(cuatro);
        System.out.println(ocho);
        System.out.println(flotante);
        System.out.println(doble);
        System.out.println(PI);
        System.out.println(sonBienvenidos);
        String s1="Bienvenidos.";
        String s2="Estructuras "+
                "de "+
                "Datos "+
                "I";
        if (sonBienvenidos){
            System.out.println(s1);
            System.out.println(s2);
        }
        //else siempre son bienvenidos
        System.out.printf("PI=%f\n",PI);
        //operador condicional
        s1=sonBienvenidos?"Si":"No";
        System.out.println(s1);
        System.out.println(sonBienvenidos?"Si":"No");
    }
}
```

```
//lenguaje C
int a=20,b=10,c;
c=a-b;
printf("la resta %d-%d=%d",a,b,c); //la resta 20-10=10

//lenguaje Java
System.out.println("la resta "+a+"-"+b+"="+c);
System.out.println("var1="+a+",var2="+b)
```

Almacenamiento de Objetos

Arreglos

Almacenamiento de tipos primitivos en arreglos

```
int i,j;
int[] numeros=new int[4];
for(i=0;i<numeros.length;i++){
    numeros[i]=i*10;
}
System.out.println("imprimiendo arreglo de int...");
for(i=0;i<numeros.length;i++){
    System.out.println(numeros[i]);
}
```

```
char[] cars=new char[4];
for(i=0;i<cars.length;i++){
    cars[i]=(char)(65+i);
}
System.out.println("imprimiendo arreglo de chars...");
for(i=0;i<cars.length;i++){
    System.out.println(cars[i]);
}
```

```
byte[] arreglo1;
arreglo1=new byte[4];
for(i=0;i<arreglo1.length;i++){
    arreglo1[i]=(byte)(65+i);
}
System.out.println("imprimiendo arreglo de bytes #1...");
for(i=0;i<arreglo1.length;i++){
    System.out.println(arreglo1[i]);
}
```

```
byte[] arreglo2={65,66,67,68};
System.out.println("imprimiendo arreglo de bytes #2...");
for(i=0;i<arreglo2.length;i++){
    System.out.println(arreglo2[i]);
}
```

```
System.out.println("Conversión de una cadena a sus ASCII...");
String curso="TP00";
System.out.println(curso);
byte[] cursoBytes;
cursoBytes=curso.getBytes();
System.out.println("imprimiendo valores ASCII...");
```

<pre> for(i=0;i<cursoBytes.length;i++){ System.out.println(cursoBytes[i]); } </pre>
<pre> System.out.println("imprimiendo caracteres..."); for(i=0;i<cursoBytes.length;i++){ System.out.println((char)cursoBytes[i]); } </pre>
<pre> System.out.println("o bien..."); for(i=0;i<cursoBytes.length;i++){ System.out.print((char)cursoBytes[i]); } </pre>
<pre> System.out.println("\nCopiando arreglos e invirtiendo..."); byte[] destino=new byte[cursoBytes.length]; j=destino.length; for(i=0;i<cursoBytes.length;i++){ j--; destino[j]=cursoBytes[i]; } for(;j<destino.length;j++){ System.out.print((char)destino[j]); } </pre>

Almacenamiento de objetos en arreglos

<pre> System.out.println("Arreglo de Objetos:"); String cadenas[]=new String[6]; cadenas[0]="Curso"; cadenas[1]="de"; cadenas[2]="Estructuras"; cadenas[3]="de"; cadenas[4]="Datos"; cadenas[5]="I"; for(i=0;i<cadenas.length;i++){ System.out.println(cadenas[i]); } for(i=0;i<cadenas.length;i++){ System.out.print(cadenas[i]); System.out.print(" "); } </pre>
<pre> public static void main(String args[]){ if (args.length>0){ for(i=0;i<args.length;i++){ System.out.println(args[i]); } } } </pre>

Encapsulamiento

<table><tr><th>MiFecha</th></tr><tr><td>+dia : int</td></tr><tr><td>+mes : int</td></tr><tr><td>+anio : int</td></tr><tr><td></td></tr></table>	MiFecha	+dia : int	+mes : int	+anio : int		<pre>class MiFecha{ //Fecha como registro public int dia; public int mes; public int anio; } public class Ocultamiento { //el problema: libre acceso del exterior public static void main(String[] args) { MiFecha f=new MiFecha(); f.dia=31; //es posible, pero de qué mes? System.out.println("dia="+f.dia); } }</pre>
MiFecha						
+dia : int						
+mes : int						
+anio : int						

<table><tr><th>MiFecha</th></tr><tr><td>-dia : int</td></tr><tr><td>-mes : int</td></tr><tr><td>-anio : int</td></tr><tr><td>+dameDia() : int</td></tr><tr><td>+dameMes() : int</td></tr><tr><td>+dameAnio() : int</td></tr><tr><td>+fijaDia(in d : int) : boolean</td></tr><tr><td>+fijaMes(in m : int)</td></tr><tr><td>+fijaAnio(in a : int)</td></tr><tr><td>-verificaDia(in d : int) : boolean</td></tr></table>	MiFecha	-dia : int	-mes : int	-anio : int	+dameDia() : int	+dameMes() : int	+dameAnio() : int	+fijaDia(in d : int) : boolean	+fijaMes(in m : int)	+fijaAnio(in a : int)	-verificaDia(in d : int) : boolean	<pre>class MiFecha{ //Fecha como clase private int dia; private int mes; //completar dame() y fija() private int anio; //completar dame() y fija() public boolean fijaDia(int diax){ dia=diax; //requiere validar antes return true; //no siempre es válido } public int dameDia(){ return dia; } } public class Encapsulamiento{ //la solución public static void main(String[] args) { MiFecha f=new MiFecha(); //f.dia=31; //no compila ocultamiento f.fijaDia(31); System.out.println("dia="+f.dameDia()); } }</pre>
MiFecha												
-dia : int												
-mes : int												
-anio : int												
+dameDia() : int												
+dameMes() : int												
+dameAnio() : int												
+fijaDia(in d : int) : boolean												
+fijaMes(in m : int)												
+fijaAnio(in a : int)												
-verificaDia(in d : int) : boolean												

<table><tr><th>MiFecha</th></tr><tr><td>-fecha : long</td></tr><tr><td>+dameDia() : int +dameMes() : int +dameAnio() : int +fijaDia(in d : int) : boolean +fijaMes(in m : int) +fijaAnio(in a : int) -verificaDia(in d : int) : boolean</td></tr></table>	MiFecha	-fecha : long	+dameDia() : int +dameMes() : int +dameAnio() : int +fijaDia(in d : int) : boolean +fijaMes(in m : int) +fijaAnio(in a : int) -verificaDia(in d : int) : boolean	<pre>class MiFecha{ //fecha en milisegundos a 64 bits private long fecha; public boolean fijaDia(int diax){ //validar diax if (diaValido){ //si diax es válido //calcular fechax nueva fecha=fechax; } return diaValido; } public int dameDia(){ //calcular a partir fecha en el pasado return dia; } //... }</pre>
MiFecha				
-fecha : long				
+dameDia() : int +dameMes() : int +dameAnio() : int +fijaDia(in d : int) : boolean +fijaMes(in m : int) +fijaAnio(in a : int) -verificaDia(in d : int) : boolean				

Constructores

MiFecha
-dia : int
-mes : int
-anio : int
+dameDia() : int
+dameMes() : int
+dameAnio() : int
+fijaDia(in d : int) : boolean
+fijaMes(in m : int)
+fijaAnio(in a : int)
-verificaDia(in d : int) : boolean

```
public class Constructor{
    public static void main(String[] args) {
        MiFecha fecha1,fecha2;
        fecha1=new MiFecha();
        fecha2=new MiFecha(2009,9,25);
        System.out.println("dia="+fecha1.dameDia());
        System.out.println("dia="+fecha2.dameDia()); }
}

class MiFecha{
    private int dia;
    private int mes; //completar dame() y fija()
    private int anio; //completar dame() y fija()

    //constructor sin parametros
    public MiFecha(){
        anio=1979;
        mes=9;
        dia=12;
    }

    //constructor con parametros
    public MiFecha(int aniox,int mesx,int diax){
        anio=aniox;
        mes=mesx;
        dia=diax;
    }

    //este es el constructor por defecto si no se
    declara uno
    /*
    public MiFecha(){
    }
    */
    ...
}
```

Comunicación entre objetos

```
class Circulo{
    private double radio;
    private int colorBorde;
    private int colorFondo;
    private final double PI=3.1416;

    public void fijaRadio(double radiox){
        radio=radiox;
    }

    public void fijaColorBorde(int colorBordex){
        colorBorde=colorBordex;
    }

    public void fijaColorFondo(int colorFondox){
        colorFondo=colorFondox;
    }

    public double dameRadio(){
        return radio;
    }

    public int dameColorBorde(){
        return colorBorde;
    }

    public int dameColorFondo(){
        return colorFondo;
    }

    public double dameArea(){
        return PI*radio*radio;
    }

    public boolean esIgualA(Circulo c){
        boolean resultado;
        if (radio==c.dameRadio() &&
            colorBorde==c.dameColorBorde() &&
            colorFondo==c.dameColorFondo()){
            resultado=true;
        }
        else{
            resultado=false;
        }
        return resultado;
    }
}
```

```

public class Comunicacion {

    public static void main(String[] args) {
        Circulo c1=new Circulo();
        c1.fijaRadio(10);
        c1.fijaColorBorde(11);
        c1.fijaColorFondo(14);
        Circulo c2=new Circulo();
        c2.fijaRadio(10);
        c2.fijaColorBorde(11);
        c2.fijaColorFondo(14);
        Circulo c3=new Circulo();
        c3.fijaRadio(10);
        c3.fijaColorBorde(4);
        c3.fijaColorFondo(14);
        if (c1.esIgualA(c2)){
            System.out.println("c1 y c2 son iguales");
        }
        else{
            System.out.println("c1 y c2 son diferentes");
        }
        if (c2.esIgualA(c3)){
            System.out.println("c2 y c3 son iguales");
        }
        else{
            System.out.println("c2 y c3 son diferentes");
        }
    }
}

```


1.5 Generalización y Especialización de Clases

1.5.1 Tipos de Herencia

Herencia Simple

Abstracción base:

<table><tr><th>Empleado</th></tr><tr><td>+nombre : String</td></tr><tr><td>+salario : double</td></tr><tr><td>+fechaNacimiento : MiFecha</td></tr><tr><td>+dameInfo() : String</td></tr></table>	Empleado	+nombre : String	+salario : double	+fechaNacimiento : MiFecha	+dameInfo() : String	<pre>class Empleado{ public String nombre; public double salario; public MiFecha fechaNacimiento; public String dameInfo(){ String info="Nombre: "+nombre+"\n"+ "Salario: "+salario+"\n"+ "Fecha Nacimiento: "+fechaNacimiento; return info; } } class MiFecha{ public int dia; public int mes; public int anio; public MiFecha(int aniox,int mesx,int diax){ anio=aniox; mes=mesx; dia=diax; } public String toString(){ //sobreescritura return anio+"/"+mes+"/"+dia; } } public class Herencial { public static void main(String[] args){ Empleado e=new Empleado(); e.nombre="Patriota"; e.salario=10000; e.fechaNacimiento=new MiFecha(1979,9,16); System.out.println(e.dameInfo()); } }</pre>
Empleado						
+nombre : String						
+salario : double						
+fechaNacimiento : MiFecha						
+dameInfo() : String						

Problema:

Encargado

+nombre : String
+salario : double
+fechaNacimiento : MiFecha
+departamento : String
+dameInfo() : String

```
class Encargado{
    public String nombre;
    public double salario;
    public MiFecha fechaNacimiento;
    public String departamento;

    public String dameInfo(){
        String info="Nombre: "+nombre+"\n"+
            "Salario: "+salario+"\n"+
            "Fecha Nacimiento: "+fechaNacimiento+"\n"+
            "Departamento: "+departamento;
        return info;
    }
}

public class Herencia3 {

    public static void main(String[] args) {
        Encargado e=new Encargado();
        e.nombre="Jefazo";
        e.salario=30000;
        e.fechaNacimiento=new MiFecha(1979,9,16);
        e.departamento="Computación";
        System.out.println(e.dameInfo());
    }
}
```

Solución:

Empleado

+nombre: String
+salario: double
+fechaNacimiento: MiFecha
+dameInfo(): String

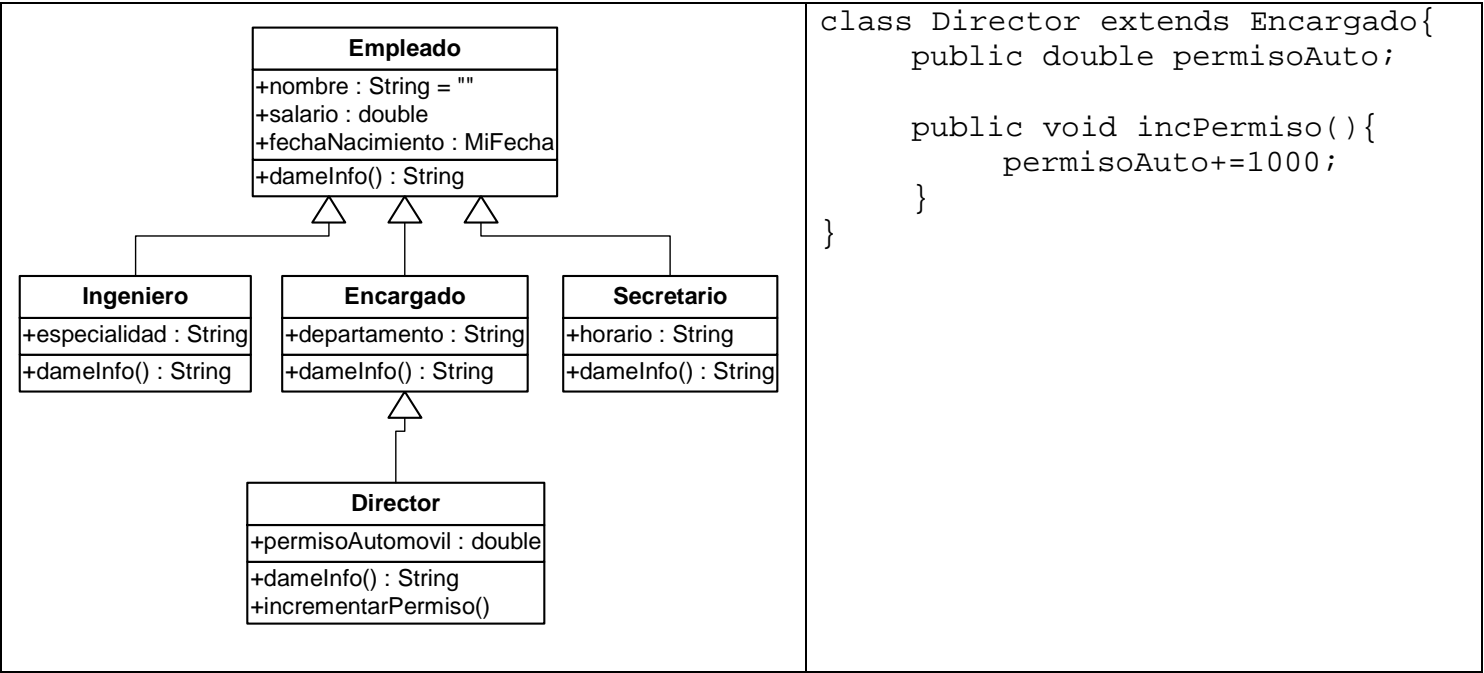
Encargado

+departamento: String
+dameInfo(): String

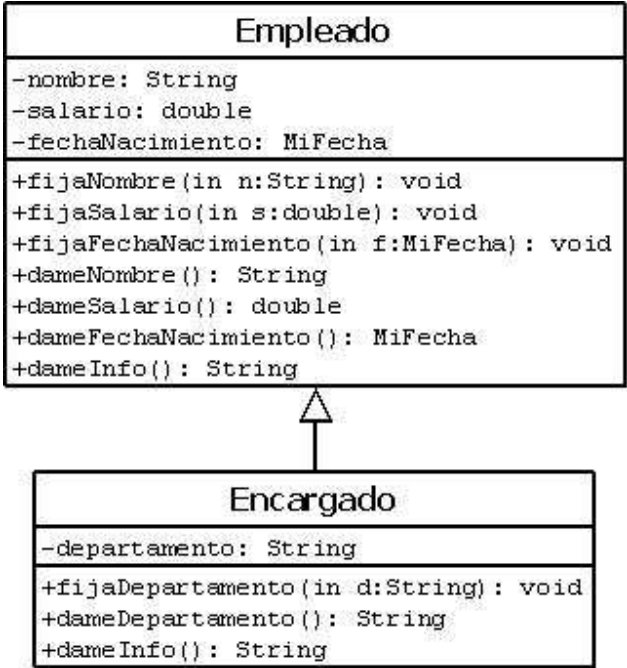
```
class Encargado extends Empleado{
    public String departamento;

    public String dameInfo(){ //sobreescritura
        String info=super.dameInfo()+"\n"+
            "Departamento: "+departamento;
        return info;
    }
}
```

Reutilización de código mediante la herencia:



Ahora con atributos privados:



Access Control

Modifier	Same Class	Same Package	Subclass	Universe
<code>private</code>	Yes			
<i>default</i>	Yes	Yes		
<code>protected</code>	Yes	Yes	Yes	
<code>public</code>	Yes	Yes	Yes	Yes

The Object Class

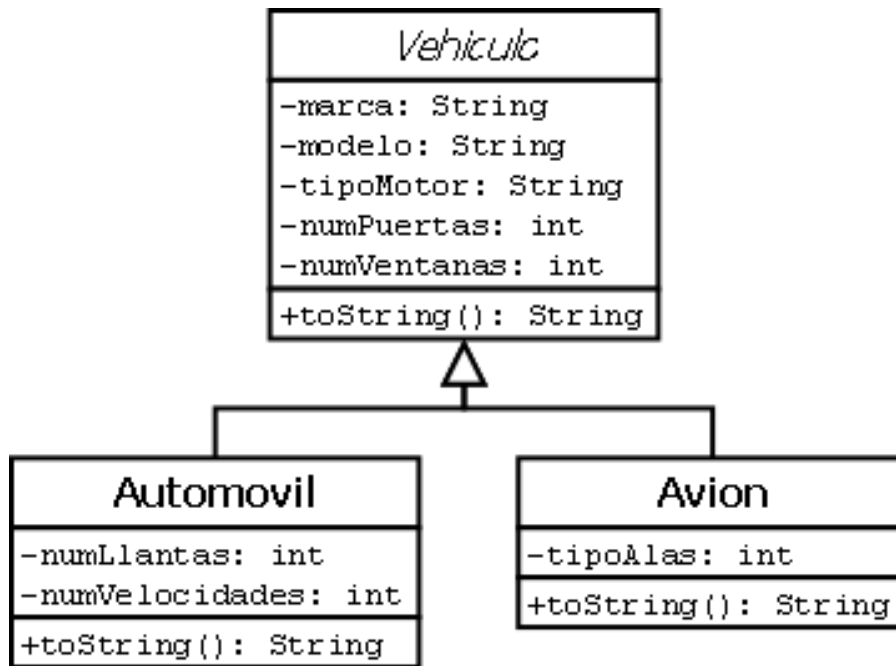
- The Object class is the root of all classes in Java
- A class declaration with no extends clause, implicitly uses “extends the Object”

```
public class Employee {  
    ...  
}
```

is equivalent to:

```
public class Employee extends Object {  
    ...  
}
```

1.5.2 Clase Abstracta



```
public class ClaseAbstracta {

    public static void main(String[] args) {
        //Vehiculo v=new Vehiculo(); //no compila
        Automovil auto=new Automovil("Ford","Mustang","Chido",2,4,4,6);
        Avion avi=new Avion("USAF","F-14","Unico",1,1,2);
        System.out.println(auto);
        System.out.println(avi);
    }
}

abstract class Vehiculo{
    private String marca;
    private String modelo;
    private String tipoMotor;
    private int numPuertas;
    private int numVentanas;

    public Vehiculo(String marcax,String modelox,String tipoMotorx,
        int numPuertasx,int numVentanasx){
        marca=marcax;
        modelo=modelox;
        tipoMotor=tipoMotorx;
        numPuertas=numPuertasx;
        numVentanas=numVentanasx;
    }

    public String toString(){ //sobreescritura de Object.toString()
        return marca+" "+modelo+" con motor "+tipoMotor+" con "+
            numPuertas+" puerta(s) y "+numVentanas+" ventana(s)";
    }
}
```

```

class Automovil extends Vehiculo{
    private int numLlantas;
    private int numVelocidades;

    public Automovil(String marcax,String modelox,String tipoMotorx,
        int numPuertasx,int numVentanasx,int numLlantasx,int
numVelocidadesx){
        super(marcax,modelox,tipoMotorx,numPuertasx,numVentanasx);
        numLlantas=numLlantasx;
        numVelocidades=numVelocidadesx;
    }

    public String toString(){ //sobreescritura de Vehiculo.toString()
        return super.toString()+" y además "+numLlantas+" llantas y "+
            numVelocidades+" velocidades";
    }
}

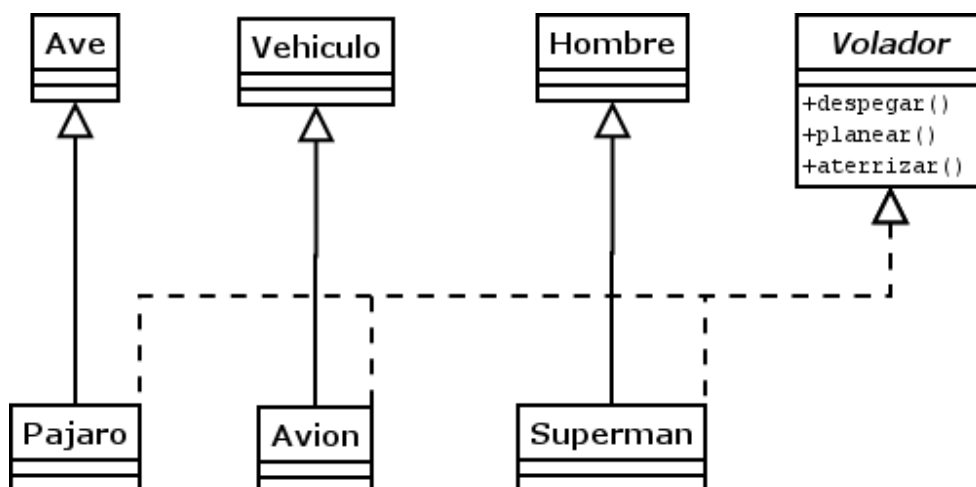
class Avion extends Vehiculo{
    private int tipoAlas;

    public Avion(String marcax,String modelox,String tipoMotorx,
        int numPuertasx,int numVentanasx,int tipoAlasx){
        super(marcax,modelox,tipoMotorx,numPuertasx,numVentanasx);
        tipoAlas=tipoAlasx;
    }

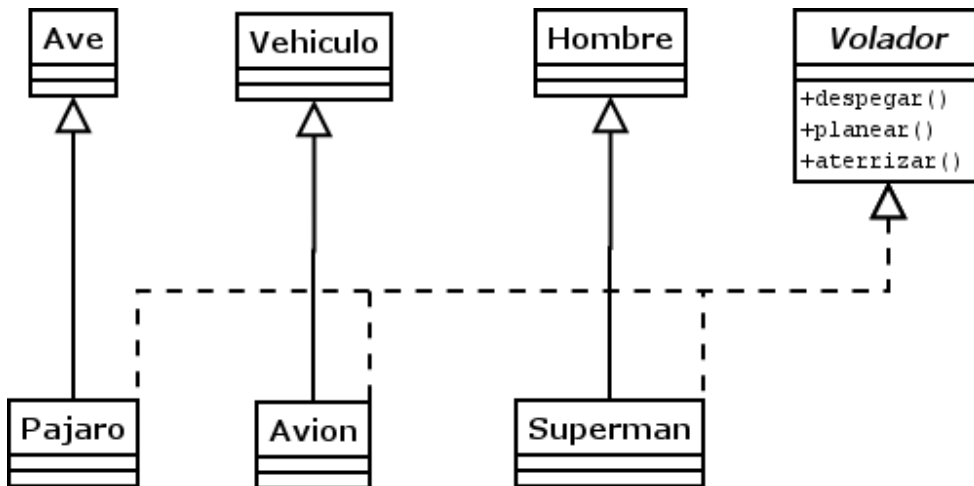
    public String toString(){ //sobreescritura de Vehiculo.toString()
        return super.toString()+" y además alas tipo "+tipoAlas;
    }
}

```

Herencia Múltiple



1.6 Expresiones del Polimorfismo



1.6.1 Definición del Polimorfismo

- Polimorfismo es la habilidad de tener muchas formas diferentes; por ejemplo la clase Encargado tiene acceso a los métodos de la clase Empleado
- Un objeto tiene únicamente una forma
- Una variable de instancia puede referenciar a objetos de diferentes formas

```
Empleado e1=new Empleado();
Empleado e2=new Encargado();
Empleado e3=new Ingeniero();
```

- Si una variable de instancia es de tipo Empleado, solo le son válidos de invocar los atributos y métodos (públicos) de la clase Empleado, aunque el objeto instanciado fuese por ejemplo de tipo Ingeniero

```
Employee employee = new Manager(); //legal
```

```
// Illegal attempt to assign Manager attribute
employee.department = "Sales";
// the variable is declared as an Employee type,
// even though the Manager object has that attribute
```

- Virtual method invocation:

```
Employee e = new Manager();
e.getDetails();
```

- Compile-time type and runtime type

Rules About Overridden Methods

- Must have a return type that is identical to the method it overrides
- Cannot be less accessible than the method it overrides

Algo que no funcionaría...

```
public class Parent {  
    public void doSomething() {}  
}  
  
public class Child extends Parent {  
    private void doSomething() {}  
}  
  
public class UseBoth {  
    public void doOtherThing() {  
        Parent p1 = new Parent();  
        Parent p2 = new Child();  
        p1.doSomething();  
        p2.doSomething();  
    }  
}
```

Polymorphic Arguments

- Because a Manager is an Employee:

```
// In the Employee class  
public TaxRate findTaxRate(Employee e) {  
}  
// Meanwhile, elsewhere in the application class  
Manager m = new Manager();  
:  
TaxRate t = findTaxRate(m);
```


The instanceof Operator

```
public class Employee extends Object
public class Manager extends Employee
public class Engineer extends Employee
-----

public void doSomething(Employee e) {
    if (e instanceof Manager) {
        // Process a Manager
    } else if (e instanceof Engineer) {
        // Process an Engineer
    } else {
        // Process any other type of Employee
    }
}
```

1.6.2 Sobrecarga de métodos

```
public class SobreCargal {
    public void imprimeBits(byte b){
        byte test=(byte)0x80;
        System.out.print(b+"=");
        for(int i=0;i<8;i++){
            if ((b&test)!=0){
                System.out.print("1");
            }
            else{
                System.out.print("0");
            }
            test=(byte)((test>>1)&0x7F);
        }
        System.out.println("");
    }
    public void imprimeBits(short s){
        short test=(short)0x8000;
        System.out.print(s+"=");
        for(int i=0;i<16;i++){
            if ((s&test)!=0){
                System.out.print("1");
            }
            else{
                System.out.print("0");
            }
            test=(short)((test>>1)&0x7FFF);
        }
        System.out.println("");
    }
    public static void main(String[] args) {
        SobreCargal sc=new SobreCargal();
        byte b1;
        short s1;
        int limitePositivo=16,limiteNegativo=-112;
        System.out.println("Positivos byte");
        for(b1=0;b1<limitePositivo;b1++){
            sc.imprimeBits(b1);
        }
        System.out.println("Negativos byte");
        for(b1=-128;b1<limiteNegativo;b1++){
            sc.imprimeBits(b1);
        }
        System.out.println("Positivos short");
        for(s1=0;s1<limitePositivo;s1++){
            sc.imprimeBits(s1);
        }
        System.out.println("Negativos short");
        for(s1=-128;s1<limiteNegativo;s1++){
            sc.imprimeBits(s1);
        }
    }
}
```

Otro ejemplo:

```
public class SobreCarga2 {

    public static void main(String[] args) {
        Empleado e=new Empleado();
        System.out.println(e.dameInfo());
        System.out.println("- - -");
        e.fijaInfo("Luis");
        System.out.println(e.dameInfo());
        System.out.println("- - -");
        e.fijaInfo("Alberto",1000);
        System.out.println(e.dameInfo());
        System.out.println("- - -");
        e.fijaInfo("Luis Alberto",4400,new MiFecha(1979,9,12));
        System.out.println(e.dameInfo());
    }

}

class Empleado{
    private String nombre;
    private double salario;
    private MiFecha fechaNacimiento;

    public Empleado(){
        nombre="staff";
        salario=0;
        fechaNacimiento=new MiFecha(2007,8,1);
    }

    public void fijaInfo(String nombrex){
        nombre=nombrex;
    }

    public void fijaInfo(String nombrex,double salariox){
        fijaInfo(nombrex);
        salario=salariox;
    }

    public void fijaInfo(String nombrex,double salariox,
        MiFecha fechaNacimientox){
        fijaInfo(nombrex,salariox);
        fechaNacimiento=fechaNacimientox;
    }

    public String dameInfo(){
        String info="Nombre: "+nombre+"\n"+
            "Salario: "+salario+"\n"+
            "Fecha Nacimiento: "+fechaNacimiento;

        return info;
    }

}
```

Overloading Method Names

- Use as follows:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

- Argument lists *must* differ.
- Return types *can* be different.

Overloading Constructors

- As with methods, constructors can be overloaded.
- Example:

```
public Employee(String name, double salary, Date DoB)
public Employee(String name, double salary)
public Employee(String name, Date DoB)
```

- Argument lists *must* differ.
- You can use the `this` reference at the first line of a constructor to call another constructor.

```
1  public class Employee {
2      private static final double BASE_SALARY = 15000.00;
3      private String name;
4      private double salary;
5      private Date    birthDate;
6
7      public Employee(String name, double salary, Date DoB) {
8          this.name = name;
9          this.salary = salary;
10         this.birthDate = DoB;
11     }
12     public Employee(String name, double salary) {
13         this(name, salary, null);
14     }
15     public Employee(String name, Date DoB) {
16         this(name, BASE_SALARY, DoB);
17     }
18     public Employee(String name) {
19         this(name, BASE_SALARY);
20     }
21     // more Employee code...
22 }
```

Constructors Are Not Inherited

- A subclass inherits all methods and variables from the superclass (parent class).
- A subclass does not inherit the constructor from the superclass.
- Two ways to include a constructor are:
 - ▼ Use the default constructor
 - ▼ Write one or more explicit constructors

Invoking Parent Class Constructors

- To invoke a parent constructor, you must place a call to `super` in the first line of the constructor
- You can call a specific parent constructor by the arguments that you use in the call to `super`
- If no `this` or `super` call is used in a constructor, then the compiler adds an implicit call to `super()` that calls the parent no argument constructor (which could be the "default" constructor)
 - ▼ If the parent class defines constructors, but does not provide a no argument constructor, then a compiler error message is issued

```
1  public class Employee {
2      private static final double BASE_SALARY = 15000.00;
3      private String name;
4      private double salary;
5      private Date    birthDate;
6
7      public Employee(String name, double salary, Date DoB) {
8          this.name = name;
9          this.salary = salary;
10         this.birthDate = DoB;
11     }
12     public Employee(String name, double salary) {
13         this(name, salary, null);
14     }
15     public Employee(String name, Date DoB) {
16         this(name, BASE_SALARY, DoB);
17     }
18     public Employee(String name) {
19         this(name, BASE_SALARY);
20     }
21     // more Employee code...
22 }
```

```

1  public class Manager extends Employee {
2      private String department;
3
4      public Manager(String name, double salary, String dept) {
5          super(name, salary);
6          department = dept;
7      }
8      public Manager(String n, String dept) {
9          super(name);
10         department = dept;
11     }
12     public Manager(String dept) { // This code fails: no super()
13         department = dept;
14     }
15 }

```

1.6.3 Sobre-escritura de métodos

```

public class Employee {
    protected String name;
    protected double salary;
    protected Date birthDate;

    public String getDetails() {
        return "Name: " + name + "\n" +
            "Salary: " + salary;
    }
}

public class Manager extends Employee {
    protected String department;

    public String getDetails() {
        return "Name: " + name + "\n" +
            "Salary: " + salary + "\n" +
            "Manager of: " + department;
    }
}

```

o bien usando la palabra clave *super* ...

```
public class Manager extends Employee {
    private String department;

    public String getDetails() {
        // call parent method
        return super.getDetails() +
            "\nDepartment: " + department;
    }
}
```

1.6.4 Método Abstracto

```
1 public abstract class Vehicle {
2     public abstract double calcFuelEfficiency();
3     public abstract double calcTripDistance();
4 }

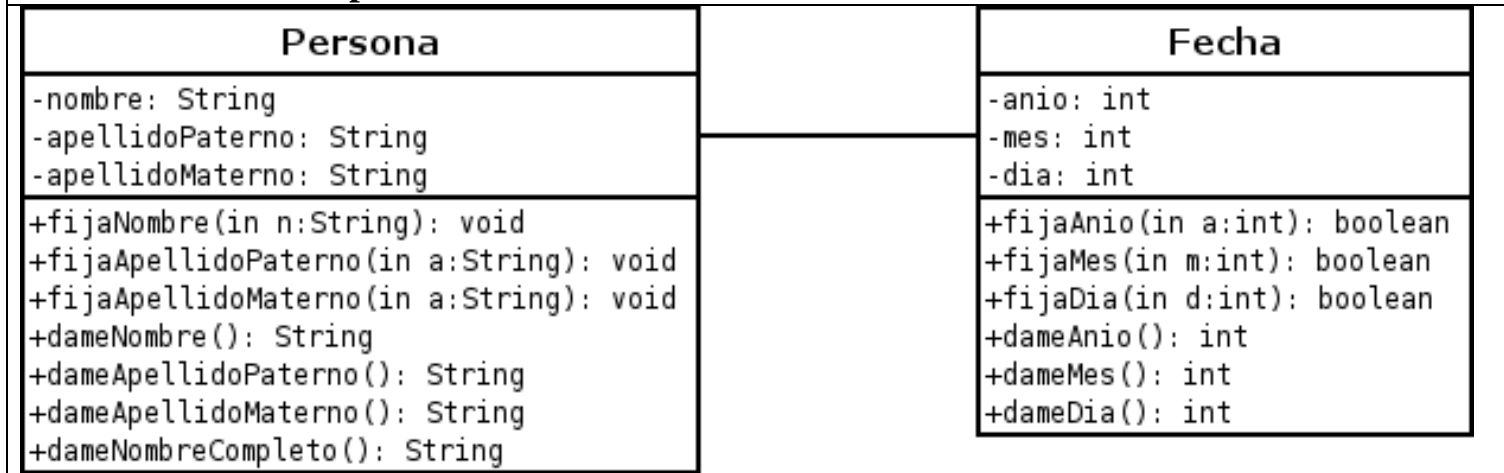
1 public class Truck extends Vehicle {
2     public Truck(double max_load) {...}
3
4     public double calcFuelEfficiency() {
5         /* calculate the fuel consumption of a truck at a given load */
6     }
7     public double calcTripDistance() {
8         /* calculate the distance of this trip on highway */
9     }
10 }

1 public class RiverBarge extends Vehicle {
2     public RiverBarge(double max_load) {...}
3
4     public double calcFuelEfficiency() {
5         /* calculate the fuel efficiency of a river barge */
6     }
7     public double calcTripDistance() {
8         /* calculate the distance of this trip along the river-ways */
9     }
10 }
```

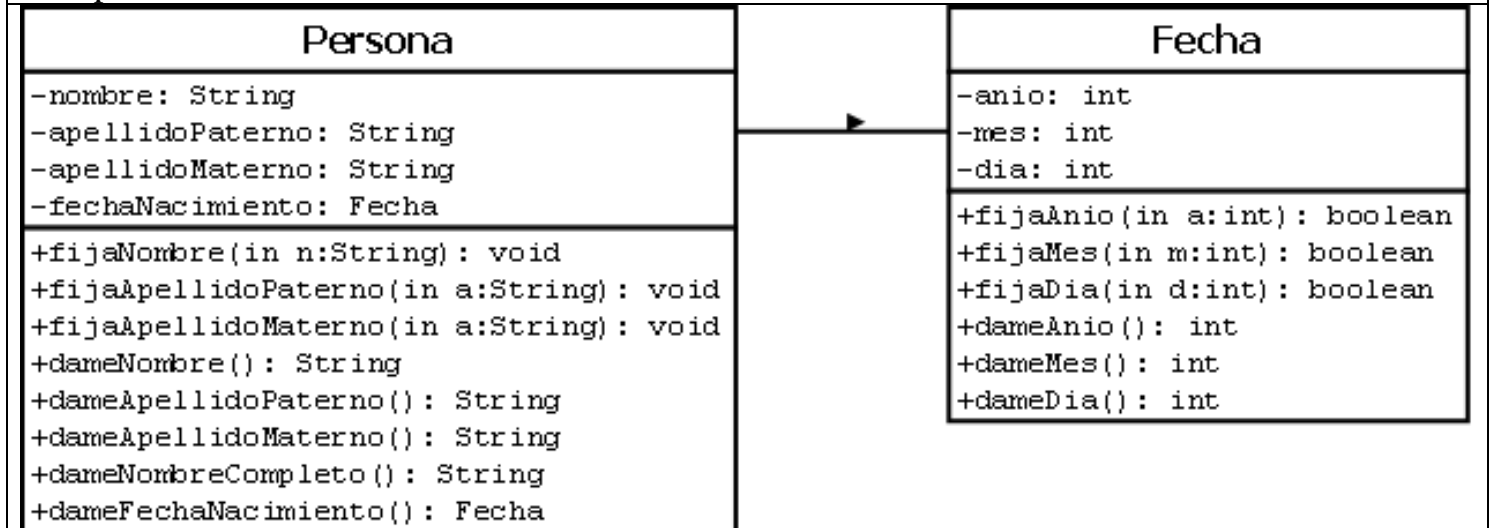

1.7 Relaciones entre clases

Asociación de clases:

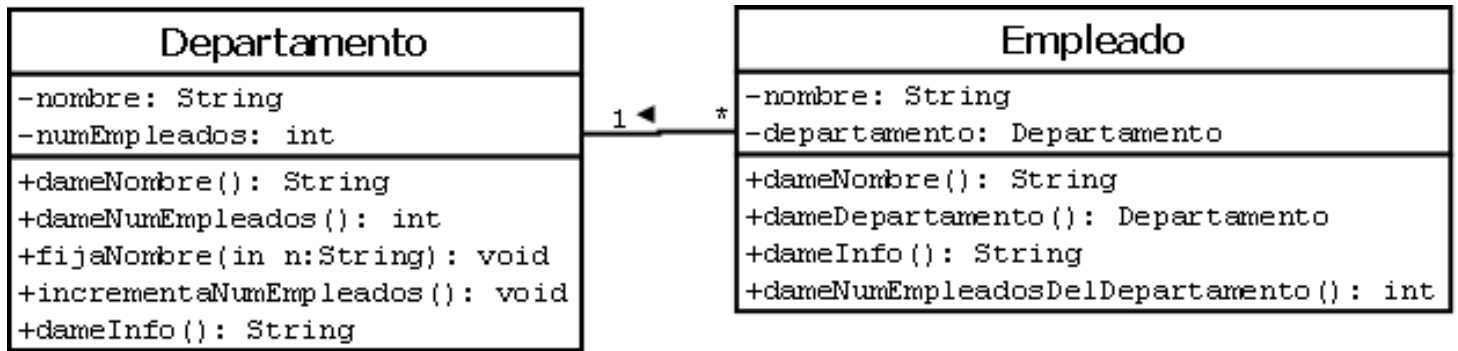
En el siguiente ejemplo dos clases están asociadas pero no es claro cual tiene la **referencia** hacia la otra o si las dos se **apuntan** mutuamente.



En el siguiente ejemplo, es claro quién tiene la **referencia** del otro, y esto se logra ya sea colocando el atributo de tipo Fecha o bien usando el triángulo sobre la asociación. La asociación es una relación débil, incluso aplicable si un método cualquiera y no un constructor, recibe un parámetro tipo clase, por lo tanto la asociación no precisamente obliga a que se tenga un atributo del tipo de la otra clase.



En el siguiente ejemplo se ilustra una relación de un Departamento hacia muchos empleados



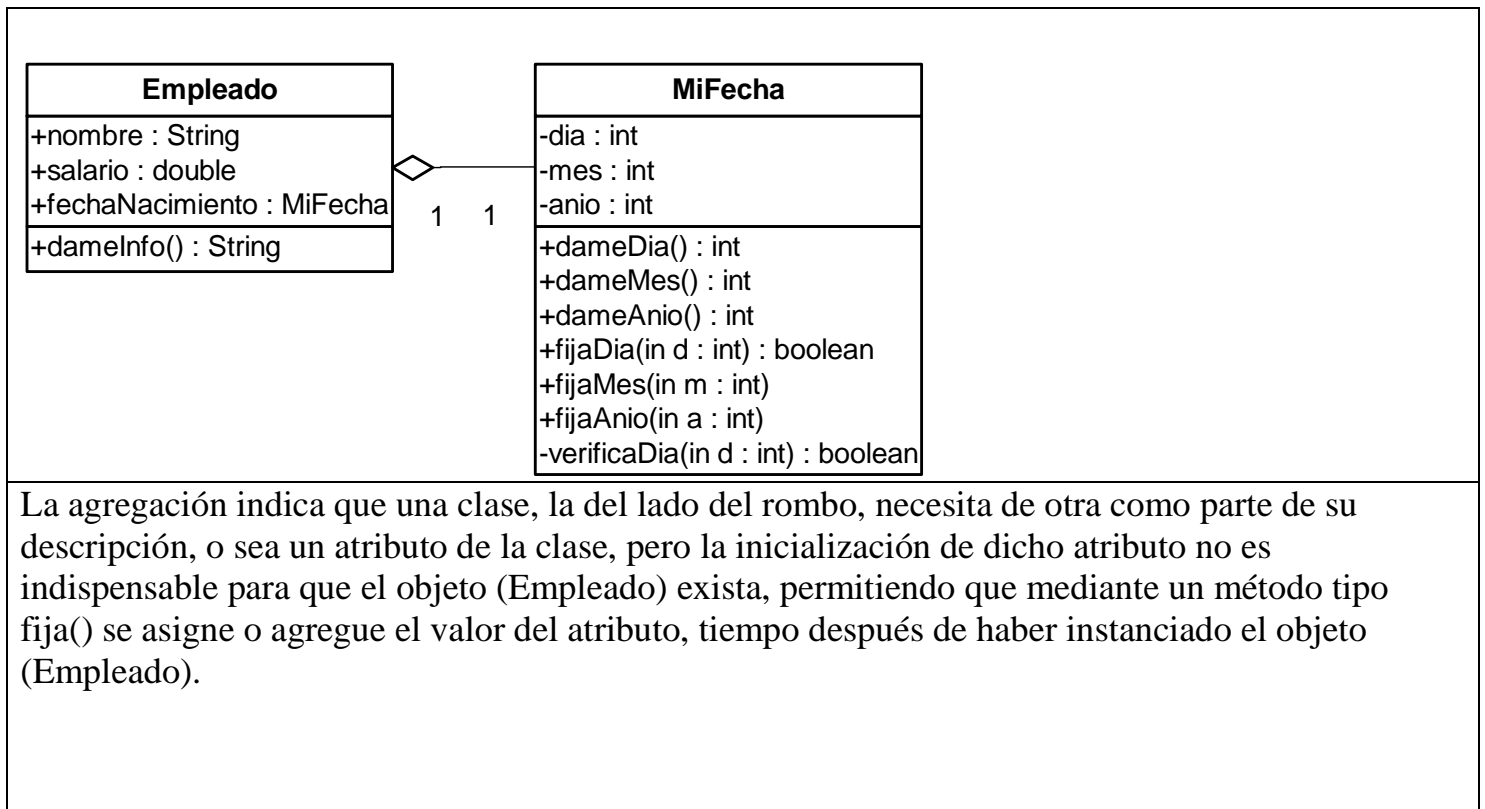
Cardinalidad

Podemos usar cardinalidad para especificar cuántos objetos de otra clase se relacionan con uno; dependiendo del mundo en abstracción se decide la cuantificación.

Para cardinalidad se pueden usar los siguientes en cualquiera de los extremos de la relación: 1, 2, 3, etc., M, N, donde estos son constantes, y si la cantidad es infinita se usa un *. La cardinalidad aplica para las relaciones de asociación, agregación y composición.

1.7.1 Agregación o composición de clases

Agregación de clases:



```

class Empleado{
    private String nombre;
    private double salario;
    private MiFecha fechaNacimiento=null;    //la agregación es opcional

    public Empleado(String nombrex, double salariox){
        nombre=nombrex;
        salario=salariox;
    }

    public void fijaFechaNacimiento(MiFecha fecha){
        fechaNacimiento=fecha;
    }

    public String dameInfo(){
        String info="Nombre: "+nombre+"\n"+
                    "Salario: "+salario+"\n"+
                    "Fecha Nacimiento: "+fechaNacimiento;

        return info;
    }
}

public class Agregacion {

    public static void main(String[] args) {
        Empleado e=new Empleado("Luis Alberto",4400);
        System.out.println(e.dameInfo());
        e.fijaFechaNacimiento(new MiFecha(1979,9,12));
        System.out.println(e.dameInfo());
    }
}

```

Asociación bidireccional:

Hombre		Mujer
-nombre: String -esposa: Mujer -dineroDisponible: float	1	-nombre: String -marido: Hombre -listadoPlatillosConocidos: String
+fijaNombre(in n:String): void +fijaEsposa(in m:Mujer): void +dameNombre(): String +dameEsposa(): Mujer +incrementaDineroDisponible(in dinero:float): void +dameDineroParaComida(in montoRequerido:float): float +quererComer(): void	1	+fijaNombre(in n:String): void +fijaMarido(in h:Hombre): void +dameNombre(): String +dameMarido(): Hombre +dameComida(): String +conocePlatillo(in nombrePlatillo:String, in costoPreparacion:float): void

En una asociación podemos tener a dos objetos que pueden existir por separado, pero que en algún momento necesitan comunicación el uno con el otro para la ejecución satisfactoria de sus métodos; aquí no hay una relación de contenedor-componente por lo que no se coloca el rombo de agregación; nótese que podemos usar cardinalidad para especificar cuántos objetos de otra clase se relacionan con uno; dependiendo del mundo en abstracción se decide la cuantificación.

```

public class ComunicacionEntreDosTiposDistintos {

    public static void main(String args[]){
        Mujer ella=new Mujer();
        Hombre luis=new Hombre();
        System.out.println("El es "+luis.dameNombre());
        System.out.println("Ella es "+ella.dameNombre());
        ella.fijaMarido(luis);
        luis.fijaEsposa(ella);
        ella.conocePlatillo("Pizza", 50);
        ella.conocePlatillo("Papas al Horno", 20);
        luis.incrementaDineroDisponible(123);
        luis.quererComer();
        luis.quererComer();
        luis.quererComer();
        luis.quererComer();
    }
}

class Mujer{
    private String nombre;
    private Hombre marido;
    private String listadoPlatillosConocidos;

    public Mujer(){
        nombre="Ella";
        marido=null;
        listadoPlatillosConocidos="";
    }

    public void fijaNombre(String n){
        nombre=n;
    }

    public void fijaMarido(Hombre h){
        marido=h;
    }

    public String dameNombre(){
        return nombre;
    }

    public Hombre dameMarido(){
        return marido;
    }

    public void conocePlatillo(String nombrePlatillo,float
costoPreparacion){
        listadoPlatillosConocidos+=nombrePlatillo+" "+costoPreparacion+" ";
        System.out.println(dameNombre()+" ahora sabe preparar
"+nombrePlatillo+
        " y eso cuesta $" +costoPreparacion);
    }
}

```

```

public String dameComida(){
    String platillosElegibles,comidaPosible,comidaElegida;
    float dineroSolicitado,dineroObtenido;
    boolean escogerOtroPlatillo;
    int pos;
    platillosElegibles=listadoPlatillosConocidos;
    dineroObtenido=-1;
    do{
        pos=platillosElegibles.indexOf(";");
        if (pos>0){
            comidaPosible=platillosElegibles.substring(0,pos);
            System.out.println(dameNombre()+" posiblemente prepare
"+comidaPosible);
            platillosElegibles=platillosElegibles.substring(pos+1);
            pos=platillosElegibles.indexOf(";");

            dineroSolicitado=Float.parseFloat(platillosElegibles.substring(0,pos));
            if (dineroObtenido== -1){

                dineroObtenido=marido.dameDineroParaComida(dineroSolicitado);
            }
            //else en una iteración anterior su marido le entregó todo
lo que le quedaba
            if (dineroObtenido>=dineroSolicitado){
                escogerOtroPlatillo=false;
                comidaElegida=comidaPosible;
                if (dineroObtenido>dineroSolicitado){

                    marido.incrementaDineroDisponible(dineroObtenido-dineroSolicitado);
                }
                //else no le devuelve nada
            }
            else{

                platillosElegibles=platillosElegibles.substring(pos+1);
                escogerOtroPlatillo=true;
                comidaElegida=null;
            }
        }
        else{
            System.out.println(dameNombre()+" no conoce ningun
platillo o se le acabaron las opciones");
            escogerOtroPlatillo=false;
            comidaElegida=null;
            if (dineroObtenido>0){
                marido.incrementaDineroDisponible(dineroObtenido);
            }
        }
    }while(escogerOtroPlatillo);
    System.out.println(dameNombre()+" preparo "+comidaElegida);
    return comidaElegida;
}
}

```

```
class Hombre{
    private String nombre;
    private Mujer esposa;
    private float dineroDisponible;

    public Hombre(){
        nombre="Luis Alberto";
        esposa=null;
        dineroDisponible=0;
    }

    public void fijaNombre(String n){
        nombre=n;
    }

    public void fijaEsposa(Mujer m){
        esposa=m;
    }

    public String dameNombre(){
        return nombre;
    }

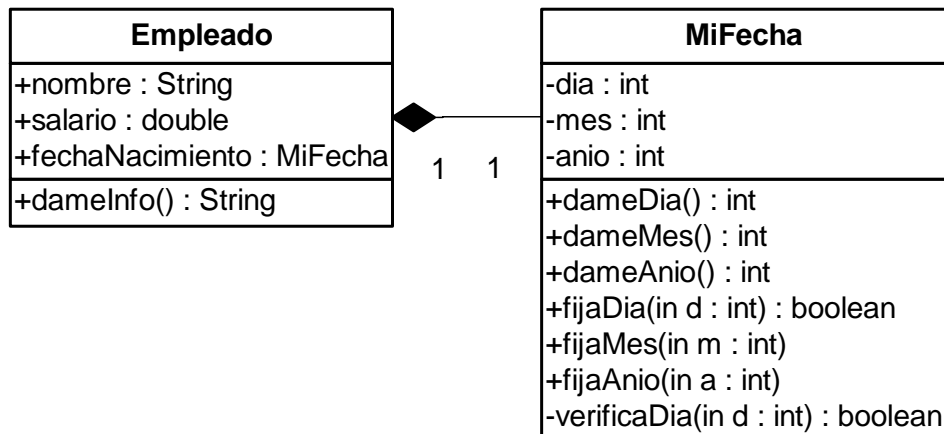
    public Mujer dameEsposa(){
        return esposa;
    }

    public void incrementaDineroDisponible(float dinero){
        System.out.println(dameNombre()+" incrementa su dinero disponible
en $" +dinero);
        dineroDisponible+=dinero;
    }

    public float dameDineroParaComida(float montoRequerido){
        float dinero;
        System.out.println(dameNombre()+" cuenta con $" +dineroDisponible);
        if (dineroDisponible>=montoRequerido){
            dinero=montoRequerido;
        }
        else{
            dinero=dineroDisponible;
        }
        dineroDisponible-=dinero;
        System.out.println(dameNombre()+" otorga $" +dinero+" y le quedan
$" +dineroDisponible);
        return dinero;
    }
}
```

```
public void quererComer(){
    String comida;
    System.out.println(dameNombre()+" quiere comer");
    comida=esposa.dameComida();
    if (comida!=null){
        System.out.println(dameNombre()+" comio "+comida);
    }
    else{
        System.out.println(dameNombre()+" debe trabajar mas para comer
=)");
    }
}
```

Composición de clases:



La composición es una relación de tipo contenedor-a-componente, obliga en diseño a que una instancia de una clase requiere de la inicialización de un atributo, ya sea que reciba la referencia al componente como parámetro en su constructor o bien que inicialice su componente con valores por defecto durante la ejecución del constructor del contenedor

```
class Empleado{
    private String nombre;
    private double salario;
    private MiFecha fechaNacimiento; //la composición es obligatoria

    public Empleado(String nombrex, double salariox,
        MiFecha fechaNacimientox){
        nombre=nombrex;
        salario=salariox;
        fechaNacimiento=fechaNacimientox;
    }

    public void fijaFechaNacimiento(MiFecha fecha){
        fechaNacimiento=fecha;
    }

    public String dameInfo(){
        String info="Nombre: "+nombre+"\n"+
            "Salario: "+salario+"\n"+
            "Fecha Nacimiento: "+fechaNacimiento;

        return info;
    }
}

public class Composicion {

    public static void main(String[] args) {
        Empleado e=new Empleado("Luis Alberto",4400,
            new MiFecha(1979,9,12));

        System.out.println(e.dameInfo());
    }
}
```


1.7.2 Delegación

```
class Secretario extends Empleado{
    private String horario;
    public Secretario(String nombrex, double salariox,
        MiFecha fechaNacimientoox, String horarioox){
        super(nombrex,salariox,fechaNacimientoox);
        horario=horarioox;
    }
    public void imprimeInfoEmpleado(Empleado e){
        System.out.println("- - - - -");
        System.out.println(e.dameInfo());
        System.out.println("- - - - -");
    }
    public String dameInfo(){ //sobreescritura
        String info=super.dameInfo()+"\n"+
            "Horario: "+horario;
        return info;
    }
}
class Encargado extends Empleado{
    private String departamento;
    private Secretario secretario;
    public Encargado(String nombrex, double salariox,
        MiFecha fechaNacimientoox, String departamentox){
        super(nombrex,salariox,fechaNacimientoox);
        departamento=departamentox;
    }
    public void fijaSecretario(Secretario s){
        secretario=s;
    }
    public void imprimeInfoEmpleado(Empleado e){
        secretario.imprimeInfoEmpleado(e);
    }
    public String dameInfo(){ //sobreescritura
        String info=super.dameInfo()+"\n"+
            "Departamento: "+departamento;
        return info;
    }
}
public class Delegacion {
    public static void main(String[] args) {
        Empleado emp=new Empleado("Luis",9000,new MiFecha(1980,2,1));
        Secretario sec=new Secretario("Alberto",5000,
            new MiFecha(1981,3,2),"7-15hrs.");
        Encargado jefe=new Encargado("Luis Alberto",20000,
            new MiFecha(1979,9,12),"Computación");
        System.out.println("El secretario sabe como imprimir...");
        sec.imprimeInfoEmpleado(emp);
        jefe.fijaSecretario(sec);
        System.out.println("El jefe solicita al secretario imprimir...");
        jefe.imprimeInfoEmpleado(emp);
    }
}
```

Casting Objects

- Use instanceof to test the type of an object
- Restore full functionality of an object by casting
- Check for proper casting using the following guidelines:
 - ▼ Casts up hierarchy are done implicitly.
 - ▼ Downward casts must be to a subclass and checked by the compiler.
 - ▼ The object type is checked at runtime when runtime errors can occur.

Retomando las clases Empleado e Ingeniero (tema 1.5.1)...

[illegible]

Heterogeneous Collections

- Collections of objects with the same class type are called *homogenous* collections.

```
MyDate[] dates = new MyDate[2];  
dates[0] = new MyDate(22, 12, 1964);  
dates[1] = new MyDate(22, 7, 1964);
```

- Collections of objects with different class types are called *heterogeneous* collections.

```
Employee [] staff = new Employee[1024];  
staff[0] = new Manager();  
staff[1] = new Employee();  
staff[2] = new Engineer();
```

```
A a=new A(); //suponiendo que A tiene el método andar()  
B b=new B(); //suponiendo que B hereda de A  
C c=new C(); //suponiendo que C hereda de A  
a.andar();  
b.andar();  
c.andar();
```

```
A[] andadores=new A[3];  
andadores[0]=new A();  
andadores[1]=new B();  
andadores[2]=new C();  
for(i=0;i<andadores.length;i++){  
    andadores[i].andar();  
}
```

```
public class ColeccionesHeterogeneas {

    public static void main(String[] args) {
        Empleado emp=new Empleado("Luis",9000,new MiFecha(1980,2,1));
        Secretario sec=new Secretario("Alberto",5000,
            new MiFecha(1981,3,2),"7-15hrs.");
        Ingeniero ing=new Ingeniero("Luis Alberto",15000,
            new MiFecha(1979,9,12),"Software de Sistemas");
        Encargado jefe=new Encargado("LA",20000,new MiFecha(1979,9,16),
            "Computación");
        Empleado empleados[]=new Empleado[4];
        empleados[0]=emp;
        empleados[1]=sec;
        empleados[2]=ing;
        empleados[3]=jefe;
        for(int i=0;i<empleados.length;i++){
            System.out.println(empleados[i].dameInfo());
        }
    }
}
```

1.8.1.3 Almacenamiento de objetos en arreglos dinámicos

```
import java.util.Vector;

public class ArreglosDinamicos {

    public static void imprimeVector(Vector v){
        String s;
        for(int i=0;i<v.size();i++){
            s=(String)v.get(i);
            System.out.println(s);
        }
        System.out.println("- - - - -");
    }

    public static void main(String[] args) {
        Vector v=new Vector();
        String s="Taller ";
        v.add(s);
        s="de ";
        v.add(s);
        s="Programación ";
        v.add(s);
        s="Orientada ";
        v.add(s);
        s="a ";
        v.add(s);
        s="Objetos ";
        v.add(s);
        ArreglosDinamicos.imprimeVector(v);
        v.remove(1);
        ArreglosDinamicos.imprimeVector(v);
        v.remove(3);
        ArreglosDinamicos.imprimeVector(v);
    }
}
```

The == Operator Compared With the equals Method

- The == operator determines if two references are identical to each other (that is, refer to the same object).
- The equals method determines if objects are “equal” but not necessarily identical.
- The Object implementation of the equals method uses the == operator.
- User classes can override the equals method to implement a domain-specific test for equality.
- Note: You should override the hashCode method if you override the equals method.

```

1 public class MyDate {
2     private int day;
3     private int month;
4     private int year;
5
6     public MyDate(int day, int month, int year) {
7         this.day    = day;
8         this.month  = month;
9         this.year   = year;
10    }
11
12    public boolean equals(Object o) {
13        boolean result = false;
14        if ( (o != null) && (o instanceof MyDate) ) {
15            MyDate d = (MyDate) o;
16            if ( (day == d.day) && (month == d.month)
17                && (year == d.year) ) {
18                result = true;
19            }
20        }
21        return result;
22    }
23
24    public int hashCode() {
25        return (day ^ month ^ year);
26    }
27 }

```

//la edición 2002 del SL-275 tiene esto, funciona así con atributos `private`?

Wrapper Classes

- Look at primitive data elements as objects

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

```
int pInt = 500;  
Integer wInt = new Integer(pInt);  
int p2 = wInt.intValue();
```

Interno de Integer.toString()

```
int a=10;  
char[] c=new char[2]  
c[0]=//codigo ascii del "1"  
c[1]=//codigo ascii del "0"  
System.out.println(""+a);
```


Notas de clase:

//acerca de la representación de datos y objetos en la memoria

Variable u objeto	a (10)				b (256)				inge (20012)				(instancia de Ingeniero)										maestro (20012)			
Dirección	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	21500	20501	20502	20503	20504
Memoria	0	0	0	10	0	0	1	0	0	0	78	44	#	#	#	#	#	#	#	...	#	0	0	78	44	

```
int a=10,b=256;
Ingeniero inge=new Ingeniero();
Ingeniero maestro=inge;
Ingeniero otroInge=new Ingeniero();

if (inge==maestro){ //comparación de direcciones de memoria
    //somos el mismo objeto
}
else{
    //somos distintos objetos,
}
if (inge==otroInge){ //comparación de direcciones de memoria
    //somos el mismo objeto
}
else{
    //somos distintos objetos,
}
```

Clase Administradora

```
public class UsaVector {
    public static void main(String[] args) {
        ClaseAdministradora admin=new ClaseAdministradora();
        Materia m;
        admin.alta("CC200","Programación Orientada a Objetos");
        admin.alta("CC201","Taller de POO");
        m=admin.consulta(1); //asumo que existe
        System.out.println(m.dameInfo());
        System.out.println(m);
        System.out.println("registro:"+m);
        admin.alta("CC202","Estructuras de Datos");
        admin.alta("CC203","Taller de Estructuras de Datos");
        System.out.println(admin.dameInfo());
        if (admin.baja("CC202")){
            System.out.println(admin.dameInfo());
        }
        else{
            System.out.println("No se pudo eliminar");
        }
    }
}

class Materia{
    public String clave;
    public String nombre;
    /*aquí constructor*/
    //este método ya se explicó como funciona
    public String dameInfo(){
        String s;
        s=clave+"\t"+nombre;
        return s;
    }
    //y que tal si se sobrescribiera de Object?
    public String toString(){
        String s;
        s=clave+"\t"+nombre;
        return s;
    }
}
```

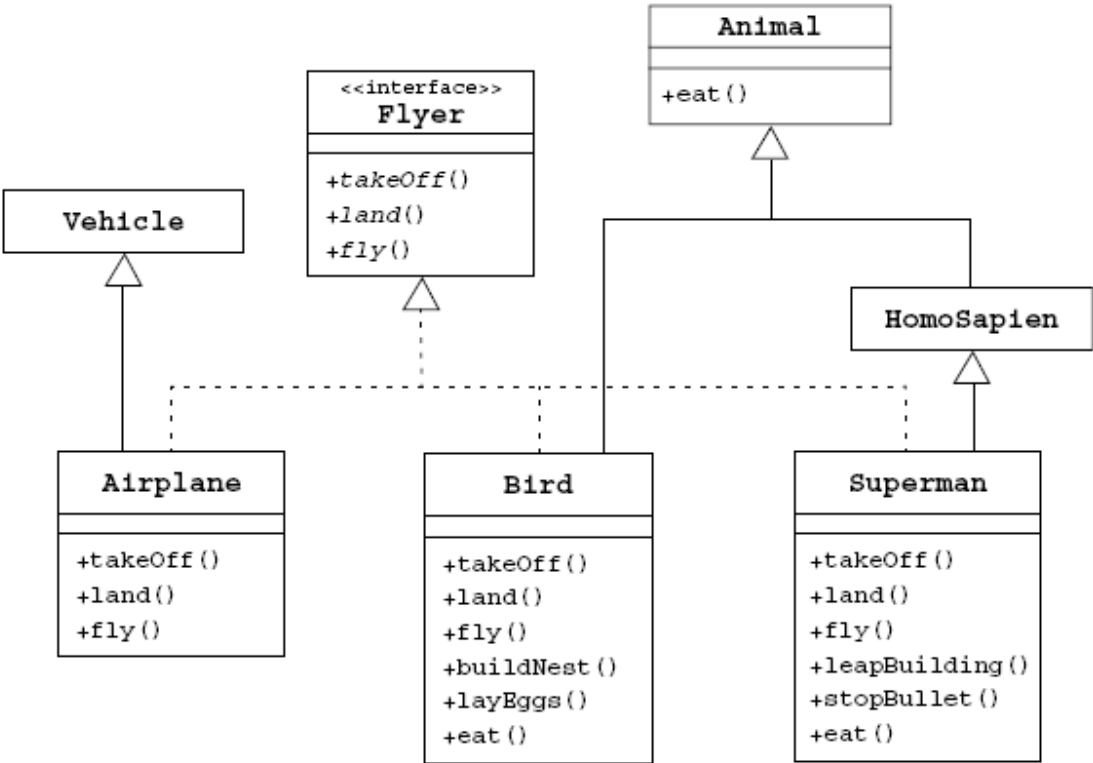
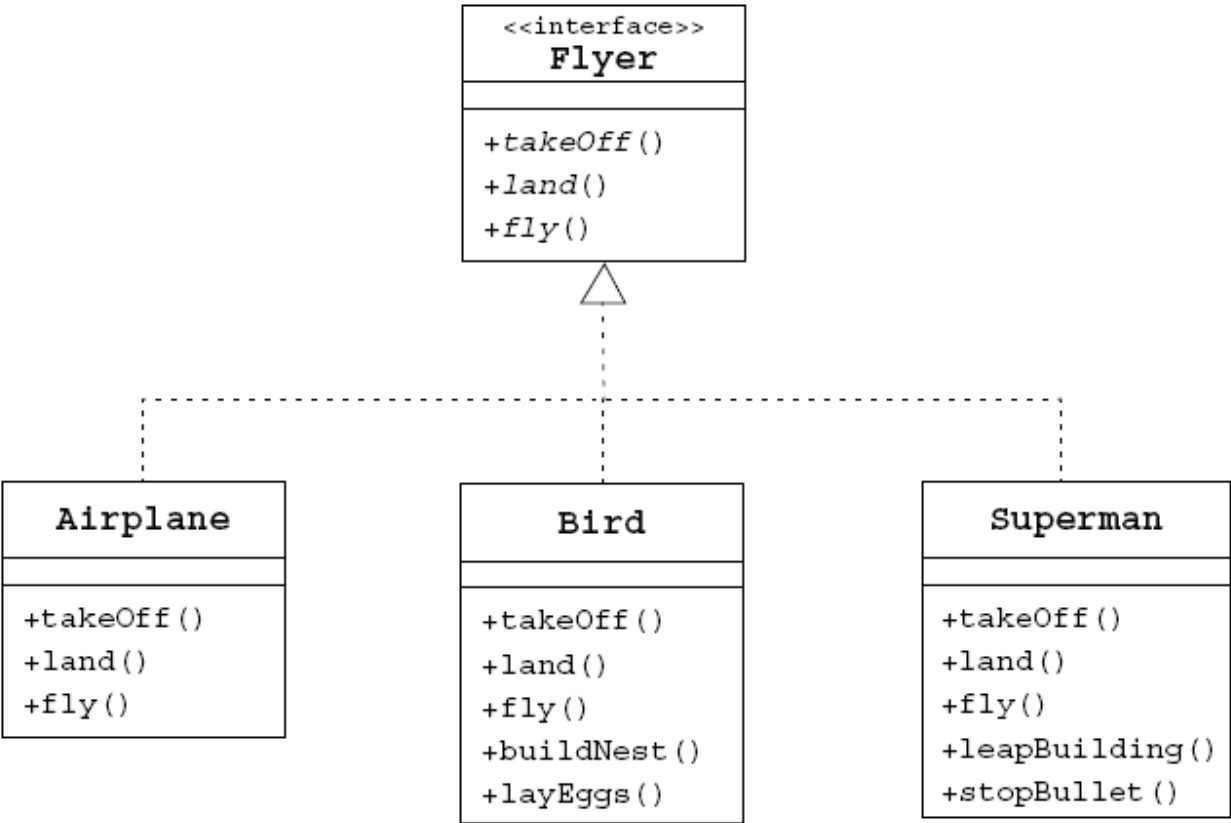
```

class ClaseAdministradora{
    private Vector lista;
    /*un constructor inicializaría lista*/
    public void alta(String clave,String nombre){
        Materia m;
        m=new Materia(clave,nombre);
        lista.add(m);
        /* se podría usar
        * lista.add(posicion,m);
        * para dar de alta en orden por clave
        */
    } //el método podría regresar boolean
    /* se podría implementar un método:
    * public void alta(String clave,String nombre,int posicion)
    */
    public Materia consulta(int posicion){
        Materia m;
        if (posicion<lista.size()){
            m=(Materia)lista.get(posicion);
        }
        else{
            m=null;
        }
        return m;
    }
    /*podría implementarse un método:
    * public Materia consulta(String clave);
    */
    public boolean baja(String clave){
        boolean encontro=false; //hay casos así
        int i,j;
        Materia m;
        for(i=0,j=lista.size();i<j;i++){
            m=(Materia)lista.get(i);
            if (m.clave.equals(clave)){
                lista.remove(i);
                encontro=true;
                break;
            }
            else{
                //seguir buscando
            }
        }
        return encontro;
    }
}

```

```
/* podría implementarse un método:
 * public boolean baja(int posicion);
 */
public String dameInfo(){
    String s="";
    Materia m;
    int i,j;
    for(i=0,j=lista.size();i<j;i++){
        m=(Materia)lista.get(i);
        s+=m.dameInfo()+"\n";
        // y que tal? si...
        //s+=m+"\n";
        //funciona igual?
    }
    return s;
}
//sería bueno sin dameInfo() y sobrescribir toString()?
}
```

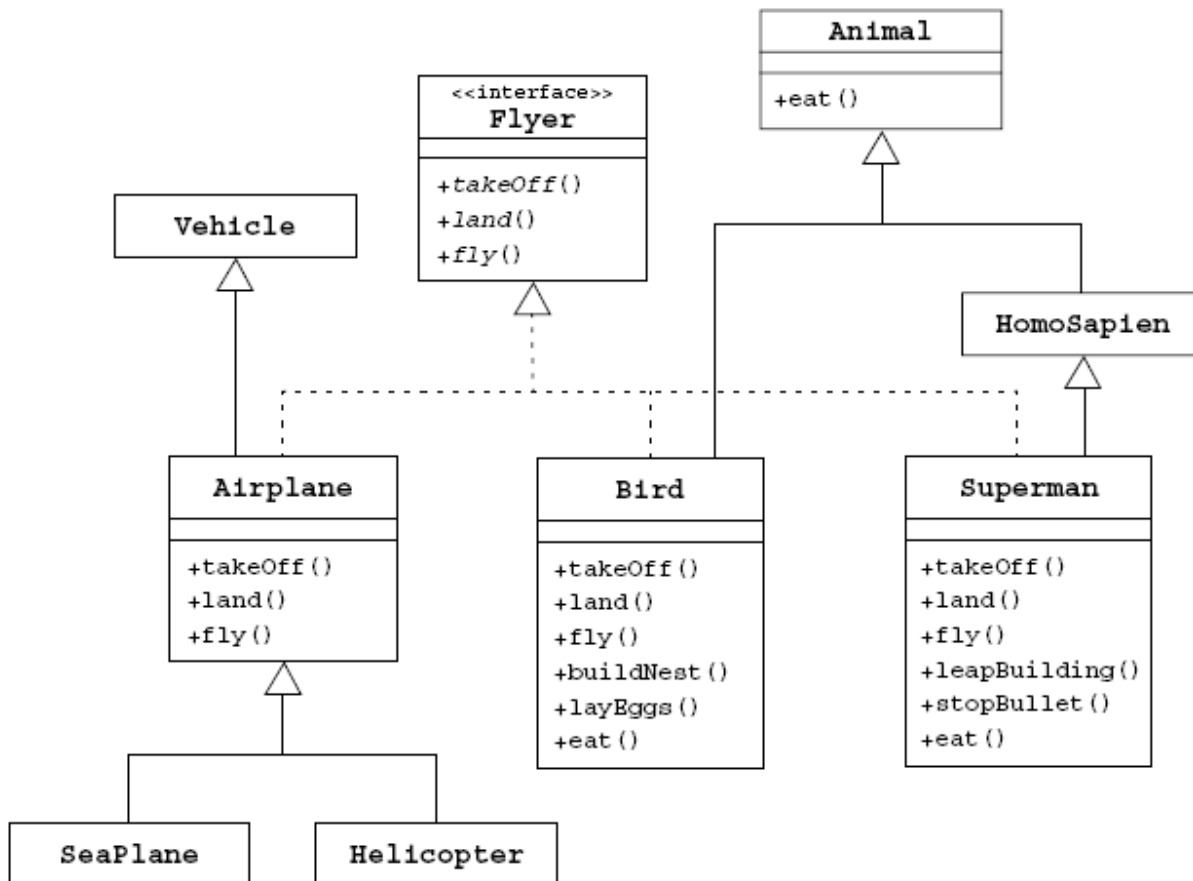
Uso de interfaces



```

public class Bird extends Animal implements Flyer {
    public void takeOff() { /* take-off implementation */ }
    public void land() { /* landing implementation */ }
    public void fly() { /* fly implementation */ }
    public void buildNest() { /* nest building behavior */ }
    public void layEggs() { /* egg laying behavior */ }
    public void eat() { /* override eating behavior */ }
}

```



```

public class Airport {
    public static void main(String[] args) {
        Airport metropolisAirport = new Airport();
        Helicopter copter = new Helicopter();
        SeaPlane sPlane = new SeaPlane();
        Flyer S = Superman.getSuperman(); // Superman is a Singleton

        metropolisAirport.givePermissionToLand(copter);
        metropolisAirport.givePermissionToLand(sPlane);
        metropolisAirport.givePermissionToLand(S);
    }

    private void givePermissionToLand(Flyer f) {
        f.land();
    }
}

```

Múltiples interfaces

