Programación Orientada a Objetos

La POO intenta promover la reutilización de clases. No solo se trata de crear clases, sino de usar o adaptar clases ya hechas. Hay tres formas fundamentales de reutilizar clases:

- Mediante composición/agregación
- Mediante herencia
- Mediante uso/dependencia

Composición/agregación

La reutilización mediante composición consiste en formar objetos a partir de otros. Cuando entre dos objetos de distintas clases se produce la relación "se compone de" o "es parte de" o "tiene un", estamos en presencia de una relación de agregación. Por ejemplo, un auto tiene un precio, el precio es un dato real, simple. Pero un auto tiene un motor, y un motor tiene a su vez una serie de particularidades, es más no todos los autos tienen el mismo motor. Además los motores no solo se usan en los autos, también en barcos, aviones, electrodomésticos, etc. Es decir, motor es una clase. La relación se puede describir:

- *un auto "se compone de" un motor*
- *un auto "tiene" un motor*
- *un motor "es parte" un auto*

Se implementa de la siguiente manera:

```
class Auto
{
    String marca;
    Motor mot;
     -----

   Auto()
    { //código }

  ---
}
```

- La agregación implica que en la clase Auto hay al menos un atributo de tipo Motor.
- En algún método de Auto mot debe ser creado, ya que esta declarado como atributo.
- Significa que desde cualquier método de Auto se pueden enviar mensajes al objeto mot.

Dependencia

Cuando entre dos objetos de distintas clases se produce la relación "usa" o "depende", estamos en presencia de una relación de dependencia. Por ejemplo, un auto tiene 4 ruedas, y si es necesario cambiar una rueda, Auto deberá tener un método cambiarRueda. Para poder cambiar una rueda es necesario usar un cricket, que no es un componente de un auto, pero se usa para cambiar la rueda. Es decir, un objeto de la clase Auto, usará un objeto de la clase Cricket, cada vez que se ejecute el método "cambiarRueda".
La relación se puede describir: *un auto "usa" un cricket*

Se implementa de la siguientes maneras:

```
class Auto
{
     ---
    void cambiarRueda(Cricket c)
    {
    //código
    }
    ---
}
```

- La dependencia significa que una clase A tiene la posibilidad de enviar mensajes a un objeto de una clase B, que recibe por parámetro o crea como variable local.

```
class Auto
{
     ___
    void cambiarRueda(Cricket c)
    {
     Cricket c = new Cricket()
      //código
    }
    ___
}
```

➢ La dependencia significa que una clase A tiene la posibilidad de enviar mensajes a un objeto de una clase B, que recibe por parámetro o crea como variable local.

Herencia

La reutilización mediante herencia consiste en crear una nueva clase partiendo de las operaciones (métodos y atributos) de otra ya existente. A la nueva clase se la llama clase hija, derivada o subtipo. A la clase original se la llame clase madre, ancestro, base o supertipo. La clase hija tendrá todas las operaciones no privadas de la clase base.

Sintaxis

```
class <claseSubtipo> extends <claseSupertipo>
{
  .........
 }
```

Ejemplo

```
class A
{
  int atributoA;
  void metodoA() {System.out.println("metodoA");}
}
class B extends A
{
.....................
}

class Ejemplo {
  public static void main (String args[])
  {
   A a = new A();
   a.setAtributoA(5);
   a.metodoA();

   B b = new B();
   b.setAtributoA(7);
   b.metodoA();
  }
}
```

Una clase derivada podrá añadir nuevas operaciones (ampliar el número de mensajes que entiende el objeto)

```
class B extends A
{
  void metodoB(){ System.out.println("metodoB");}
}

class Ejemplo {
  public static void main (String args[])
  {
   A a = new A();
   a.metodoB();          // ERROR: "a" no entiende

   B b = new B();
   b.metodoB();         // OK
  }
}
```

Al **redefinir** un método se puede reutilizar la implementación sustituida. La palabra "super" se utiliza para referirse a los métodos de la clase base.

```
class B extends A
{
  void metodoA(){
      System.out.println("antes");
      Super.metodoA();
      System.out.println("despues");
  }
}

class Ejemplo {
  public static void main (String args[])
  {
   B b = new B();
   b.metodoA();
  }
}
```

Cuando una clase no deriva explícitamente de otra clase entonces deriva implícitamente de la clase java.lang.Object. Por tanto, **toda clase de Java es un Object** y tendrá las operaciones de este último. Algunos métodos de la clase Object:

```
boolean equals(Object)
void finalize()
int hashCode()
String toString()
```

> Un aspecto fundamental de la herencia es la relación entre el tipo base y el tipo derivado. La clase derivada tiene, al menos, las operaciones de la clase base. Es decir, un objeto de la primera clase *entiende* todos los mensajes que pudiera *entender* un objeto de la segunda. Por tanto *cualquier referencia del tipo base podrá apuntar a un objeto del tipo derivado* (entiende todos sus mensajes)

```
class Vehiculo
{
  void anda(){ System.out.println("andando");  }
}

class Coche extends Vehiculo { }

class Bicicleta extends Vehiculo { }

class Ejemplo
{
  public static void conduce (Vehiculo v)
  {
   ......
  }

  public static void main (String args[])
  {
    Vehiculo v;
    Coche c = new Coche();

    v = c;          // El coche es un subtipo de Vehiculo
    v = new Bicicleta();

    conduce(c);    /* El metodo "conduce" quiere un objeto de un
                      determinado tipo (que entienda ciertos mensajes).
```

```
                        El coche que se la pasa entenderá todos esos
                        Mensajes */
  }
}
```

Polimorfismo

La herencia permite que distintas clases puedan compartir un conjunto común de operaciones (las de su clase base).
El polimorfismo (dynamic bynding) permite que aunque dos objetos acepten los mismos mensajes puedan realizarlos de forma diferente.

```
class Vehiculo
{
  void anda(){ System.out.println("andando");  }
}

class Coche extends Vehiculo {
   void anda(){ System.out.println("acelerando");}
}

class Bicicleta extends Vehiculo {
   void anda(){ System.out.println("pedaleando");}
}

class Ejemplo
{
  public static void conduce (Vehiculo v)
  {
    v.anda();     //¿A qué método le llega el mensaje?
  }

  public static void main (String args[])
  {
    Vehiculo h = new Vehiculo();
    Coche c = new Coche();
    Bicicleta b = new Bicicleta();

    conduce(h);
    conduce(c);
    conduce(b);
  }
}
```

El polimorfismo permite que aunque "v" sea una referencia de tipo Vehiculo se tenga en cuenta el tipo del objeto al que se apunta (dinamic binding). Es importante identificar los dos tipos que participan en el envío de un mensaje:

- El tipo de la referencia
- El tipo del objeto al que apunta la misma

El tipo de la referencia es el que determina los mensajes que ésta pueda transmitir a un objeto asociado. Por tanto, solo se podrá asociar a objetos *que entiendan*, al menos, dichos mensajes (objetos del mismo tipo o de un tipo derivado).
El tipo de objeto asociado es el que determina que método es el *que despachará* el mensaje recibido.

```
class A {
   void mensaje1()  { System.out.println("mensaje1");}
}
class B extends A {
   void mensaje2()  { System.out.println("mensaje2");}
}

class Ejemplo {
   public static void main (String args[])
```

```
    {
     A a;
     B b;

     a = new A();
       // OK. Se asocia "a" con un objeto que entiende todos sus mensajes

     a.mensaje1();
       // "a" envía un mensaje perteneciente a su tipo
       /*  El objeto asociado es de tipo A por lo que será despachado por
           el metodo de esta clase */

     a = new B();
       // OK. Se asocia "a" con un objeto que entiende todos sus mensajes

     a.mensaje1();
       // "a" emite un mensaje de su tipo
       /*  El objeto asociado es de tipo B por lo que será despachado por
           el metodo de esta clase */

     a.mensaje2();
       // ERROR : el mensaje2 no está entre los que "a" puede transmitir

     b = new A();
       /* ERROR : el tipo del objeto asociado no es compatible con el de
        la referencia (no entendera el mensaje2)*/

     b = new B();
       // OK. El objeto asociado entiende todos los mesnajes que "b" emita

     b.mensaje1();
     b.mensaje2();

       // "b" emite mensajes de su tipo y son despachados.
    }
}
```

Resumen y ejemplo de Polimorfismo

Polimorfismo = "muchas formas"

- Es la capacidad de un solo operador o nombre de subprograma (método) para referirse a varias definiciones de función, de acuerdo con los tipos de datos de los argumentos y resultados. El polimorfismo es un mecanismo que permite a un método realizar distintas acciones al ser aplicado sobre distintos tipos de objetos que son instancias de una mismas jerarquía de clases.
- El polimorfismo se realiza en tiempo de ejecución debido a la capacidad del lenguaje de realizar "binding" ligadura dinámica.

Supongamos que debemos trabajar con rectángulos, circunferencias y triángulos. Todos estos elementos tienen en común cierto comportamiento (superficie, perímetro, etc.) ya que son figuras geométricas, aunque se diferencian por su definición. Si debemos hacer programas que manipulen figuras geométricas, la jerarquía de herencia seria la que abajo se describe, luego el polimorfismo a través de herencia y binding dinámico resuelve la situación de una manera muy eficiente:

```
class Figura
{
   float Superficie(){}
   float Perimetro(){}
   void Dibujar(){}
}
```

- Figura no tiene atributos
- Métodos sin implementación

```
class Rectangulo extends Figura
{
  // declaración de los atributos del
     rectángulo
float Superficie()
  { // código que calcula la superficie
    // del rectángulo  }
float Perimetro()
 { // código que calcula el perímetro
   // del  rectángulo  }
void Dibujar()
 { // código que dibuja el rectángulo  }
}
```

```
class Circunferencia extends Figura
{
   // declaración de los atributos de la
   // Circunferencia
  float Superficie()
   { // código que calcula la superficie
    // de la circunferencia  }
  float Perimetro()
   { // código que calcula el perímetro
     // de la circunferencia  }
  void Dibujar()
   { // código que dibuja la circ. }
 }
```

```
class Triangulo extends Figura
{
   // declaración de los atributos del
   //  Triangulo
float Superficie()
  { // código que calcula la superficie
    // del triangulo   }
float Perimetro()
  { // código que calcula el perímetro
    //  del  triangulo   }
void Dibujar()
  { // código que dibuja el triangulo }
}
```

Ejemplo 1: Supongamos que en un programa se trabaja con un conjunto de Figuras, para lo cual se utilizará un arreglo:

```
Figura [ ] geometría = new Figura(10);  // crear el arreglo
geometria[0] = new Triangulo(....);      // agregar figuras al arreglo
geometria[1] = new Rectangulo(....);
geometria[2] = new Triangulo(....);
----
geometria[9] = new Circunferencia(....);

// muestra la superficie de cada figura
for (int i=0; i<10;i++)
   {
     int s = geometria[i].Superficie();
     System.out.println(s);
   }
// muestra el perímetro de cada figura
for (int i=0; i<10;i++)
   {
     int p = geometria[i].Perimetro();
     System.out.println(p);
   }
// dibuja cada figura
for (int i=0; i<10;i++)
   geometria[i].Dibujar();
```

En tiempo de compilación geometria se ha definido como un arreglo de figuras, pero en ejecución contiene objetos triangulo, circunferencia y rectángulo, que por herencia son también figura. **Lo importante es que al invocar el método superficie, se ejecutará el algoritmo particular del objeto realmente almacenado.**
Además en tiempo de ejecución nunca se crearán objetos de clase Figura. Las ventajas son obvias:
- al agregar una nueva figura (clase), solo se debe agregar la clase nueva como derivada de Figura

El *binding dinámico* se refiere a la capacidad de enlazar en tiempo de ejecución con los métodos y atributos del objeto realmente almacenado, que no es el mismo que el objeto definido en tiempo de compilación.

El beneficio es aun mayor en el siguiente ejemplo: supongamos el caso de un objeto que denominamos Pizarra que tiene la facultad de trabajar con conjuntos de figuras. (se aplica agregación) Ejemplo 2.

```
     class Pizarra
     {
     Figura [ ] elementos
     int indice;

     Pizarra()
     {
        elementos = new Figura(10);
        indice=0;
     }
     void agregarFigura(Figura f)
     {
      if (indice<10)
          {elementos[indice] = f;
           indice++;}
     }
     float superficiePizarra()
     {
      int s = 0;
      for (int i = 0; i< indice; i++)
          s = s+ elementos[indice].Superficie();
      return s;
     }

     void dibujaPizarra()
     {
     for (int i = 0; i<indice; i++)
        elementos[indice].Dibujar();
     }
}
```

Supongamos que en un programa se trabaja con la pizarra:

```
Pizarra mi_pizarra = new Pizarra();

mi_pizarra.agregarFigura(new Triangulo(....));
mi_pizarra.agregarFigura(new Rectangulo(....));
mi_pizarra.agregarFigura(new Circunferencia(....));
```

En tiempo de compilación, se ha definido que el método agregarFigura, recibe por parámetro un objeto de tipo Figura, pero en ejecución se envían objetos de tipo Triangulo, etc. Es decir subclases de Figura.

```
____

mi_pizarra.agregarFigura(new Rectangulo(....));

// dibuja las figuras de la pizarra
mi_pizarra.dibujaPizarra();

System.out.println(mi_pizarra.superficiePizarra());
```

Para el cálculo de la superficie como para dibujar cada figura, se toma la implementación del objeto creado en ejecución.

No se debe confundir polimorfismo con sobrecarga. El polimorfismo es la misma función en distintos tipos (clases), en ejecución se envía distintos objetos a los declarados (se resuelve en tiempo de ejecución) en cambio la sobrecarga se refiere a distintas funciones en el mismo tipo (clase), en ejecución se envía el mismo objeto a los declarados, y se resuelve en tiempo de compilación. Concluyendo, la sobrecarga se resuelve en tiempo de compilación, dado que los métodos se deben diferenciar en el tipo o en el número de parámetros. El polimorfismo se resuelve en tiempo de ejecución todos los métodos tienen los mismos parámetros, las acciones cambian en función del objeto al que se aplica.

Upcasting
Asignar una referencia de un tipo derivado a un tipo base es válido, pero ¿y lo contrario?

```
class Vehiculo
{
  void anda(){   }
}

class MercedesDeLujo extends Vehiculo
{
  void anda(){   }
}
class Canoa extends Vehiculo {
  void echarAlAgua(){   }
  void anda(){   }
  void sacarDelAgua(){   }
}

Vehiculo vehiculo;
if (condicion)
   vehiculo = new Canoa();
else
   vehiculo = new MercedesDeLujo();

Canoa canoa = vehiculo;      // valido????
canoa.echarAlAgua();         // que ocurrirá???
```

Sin embargo hay ocasiones en las que se sabe con certeza que el objeto es del tipo adecuado. En este caso se podrá forzar la asignación mediante un *cast.*

```
vehiculo = new Canoa();
Canoa c = (Canoa)vehiculo;
```

El cast comprueba que el objeto es del tipo adecuado y realiza la asignación. Si no es asi se produce una excepción.

Operador instanceof

Este operador permite averiguar si un objeto pertenece a una clase.

```
if (ref instanceof MiClase)
```

De esta manera se puede saber si un cast puede aplicarse.

```
if (vehiculo instanceof Canoa)
Canoa c = (Canoa)vehiculo;
```

También devuelve true si la clase es un supertipo de la clase de la instancia.

```
if (vehiculo instanceof Object)
 // siempre true
```

Constructores y Herencia

Los constructores **NO** se heredan.

```
class A {
   A(int i) {  }
   A()  {  }
}
class B extends A { }
-
-
B b = new B(14);   // ERROR
```

Una clase solo tendrá los constructores que declare. Si no declara ninguno se le asignara el constructor por defecto.

Un objeto tiene los atributos de su clase más los que herede. Construir un objeto comprende dar un valor inicial a todos. La clase derivada solo tendrá que inicializar los suyos: los heredados se inicializan en el constructor del ancestro. Java invoca (antes) automáticamente el constructor de la clase base en cada constructor de la clase derivada.

```
class A {
   int i;
   A()  { i = 100;}
}
class B extends A {
   int j;
   B() { j = 200;}
}

B b = new B();
System.out.println(b.getI()+ " "+b.getJ());
```

¿Qué pasa si el derivado no tiene constructor?

```
class B extends A {
   int j;
}

B b = new B();
System.out.println(b.getI()+ " "+b.getJ());
```

¿Qué pasa si el ancestro no tiene constructor sin argumentos?

```
class A {
   A(int i) { this.i = i;  }
}
class B extends A { }
-
-
B b = new B(14);
```

Es obligatorio que se invoque algún constructor del ancestro (garantiza su estado). Mediante el operador **super** se puede seleccionar el constructor adecuado. Super solo puede aparecer una vez y debe ser la primera línea del constructor.

```
class A {
   int i;
   A(int i)  { this.i = i;}
}
class B extends A {
   int j;
   B() {
     super(100);
     j = 200;
   }
}
```

Clases abstractas

Los métodos de la clase Figura tenían una implementación vacía. Su objetivo era declarar un conjunto de mensajes común para poder tratar con sus derivados. Por tanto no tiene sentido:

- Instanciar una figura
- Que una clase derivada no redefina sus métodos

Se necesita un mecanismo que haga evidente la intención de este tipo de clases: las clases abstractas.

Una clase abstracta es aquella que tiene uno o más métodos abstractos. Un método abstracto es aquel que no lleva implementación. Deberá ser aportada por cada uno de los subtipos. Son identificados por la palabra *abstract*

```
abstract class Vehiculo
```

```
{
  abstract void anda();
  abstract void para();
}
```

Una clase abstracta no puede instanciarse (tiene métodos sin implementar). Si un subtipo no redefine todos los métodos abstractos será a su vez una clase abstracta.

Resumen de clases abstractas

Creación de clases abstractas
- Identificar las operaciones comunes que deben tener los objetos a manipular.
- Crear con ellos una clase abstracta (tipo)
- Identificar las operaciones a redefinir y dar implementación al resto.

Uso de clases abstractas
- Su único objetivo es la derivación
- Los subtipos tendrán que implementar las operaciones abstractas
- Las demás podrán heredarse, sustituirse o ampliarse.

Clases Final

Puede que se desee que una clase no pueda ser derivada. Razones:
- Seguridad. No se desea que un derivado pueda sustituir código peligroso.
- Eficiencia¿?

Si una clase es final el compilador no permitirá su extensión.

```
class A
{
   .....
}
class B extends A    // ERROR
{
   .....
}
```

Métodos final

También hay un nivel intermedio: que se pueda derivar de la clase pero no ciertos métodos. Un método final no puede ser redefinido.

```
class A
{
 final void validaContrasena()
  { ..... }
 void f()
  { ..... }
}
class B extends A    // ERROR

{
 final void validaContrasena() // ERROR
  { ..... }
 void f()                      // OK
  { ..... }
}
```

Hacer una clase o método final es una gran restricción. Solo deben hacerse en situaciones especiales.

Atributos final

Un atributo final no puede ser modificado una vez que haya sido inicializado.

```
class A
{
 final double x = Math.random();
void f()
  { x = 7; }      // ERROR
}
```

Constantes
Si el valor del atributo final es el mismo para todas las instancias se esta desaprovechando espacio.

```
class A
{
 final double x = Math.random();  // OK
 final int i = 7;                 // REDUNDANTE
}
```

En Java las constantes se declaran combinando los modificadores static y final.

```
class A
{
 static final int i = 7;
 static final int MAX = 100;
 static final double PI = 3.141592;
}
```

Las constantes no ocupan memoria.

Interfaces

En una determinada aplicación se necesita ordenar una serie de personas por nombre.

Posible solución:

```
class Persona {
  String name;
  .....
  String getName() {}
  ........
}

public void ordena (Persona [] pers)
{  // insercion
   for ( int i = 0; i < pers.length - 1; i++)
      for ( int j = i+1; j < pers.length; j++)
          if (pers[j].getName().compareTo(pers[i].getName()) < 0)
          {
            Persona temp = pers[i];
            pers[i] = pers[j];
            pers[j] = temp;
          }
 }
```

En una determinada aplicación se necesita ordenar una serie de facturas por número.

Posible solución:

```
class Factura {
  int numero;
  .....
  int getNumero() {}
  ........
}
```

```
public void ordena (Factura [] fact)
{  // insercion
   for ( int i = 0; i < fact.length - 1; i++)
      for ( int j = i+1; j < fact.length; j++)
          if (fact[j].getNumero() < fact[i].getNumero())
          {
            Factura temp = fact[i];
            fact[i] = fact[j];
            fact[j] = temp;
          }
 }
```

¿¿¿Porque no se reutiliza algo tan habitual como los algoritmos de ordenamiento???

- Por que se ha ligado el algoritmo a las estructuras de datos (Factura o Persona) que manipula.
- Aunque los pasos del algoritmo son siempre los mismos se implementan mediante operaciones particulares a cada tipo.
- Es la implementación elegida del algoritmo la que limita la reutilización.
- Dependiendo del tipo a ordenar la operación de comparación debe realizarse de forma distinta.

Como lo podemos solucionar?

Solución 1: Herencia

```
abstract class Ordenable {
  abstract boolean anterior(Ordenable o);
}

public void ordena (Ordenable [] elem)
{  // inserción
   for ( int i = 0; i < elem.length - 1; i++)
      for ( int j = i+1; j < elem.length; j++)
          if (elem[j].anterior(elem[i]))
          {
            Ordenable temp = elem[i];
            elem[i] = elem[j];
            elem[j] = temp;
          }
 }
```

Ejemplo

```
class Persona extends Ordenable{
  String name;
  .....
  String getName() {}
  ........
  boolean anterior (Ordenable o)
  {
   Persona p = (Persona) o;
   return getName().compareTo(p.getName())<0;
}
```

Para ordenar personas

```
Persona [] personas = new Persona[100];
< cargar el array >
ordena(personas)
```

La solución mediante herencia tiene aún inconvenientes que resolver.

Ejemplo. Implementar una búsqueda binaria.

```
abstract class Comparable
{
  abstract boolean igualA(Comparable c);
```

```
  abstarct boolean menorQue(Comparable c);
}

public int busca(Comparable[] elementos, Comparable elemento)
{
 int inferior = 0;
 int superior = elemento.length - 1;
 while (inferior <= superior)
   {
    int medio = (inferior + superior) / 2;
    if (elemento.igualA(elementos[medio]) return medio;
    else
       if (elemento.menorQue(elementos[medio])
          superior = medio - 1;
       else
          superior = medio + 1;
    }
  return - 1;
}
```

En una aplicación necesitamos ordenar un array de Personas y buscar a una persona en el array. ¿Cómo se implementa la clase Persona?

No se puede: JAVA no permite herencia múltiple de implementación.

Herencia Múltiple
La herencia múltiple de implementación consiste en que una clase pueda heredar sus operaciones de varias clases.

Ejemplo C++:

```
class A {
public:
  int atributoA;
  void metodoA() {...}
}
class B {
public
  int atributoB;
  void metodoB() {...}
}

class C: public A, publicB
{
  .... // operaciones que quiera añadir C
}
```

Los lenguajes C++ y Eiffel (entre otros) admiten herencia múltiple. Esa flexibilidad deben pagarla en complejidad o ineficiencia. La herencia múltiple presenta problemas cuando se hereda varias veces de una misma clases. JAVA opta por buscar la solución en otro mecanismo: **los interfaces**.

El problema de la herencia es que ofrece los mensajes del supertipo pero también sus implementaciones (métodos y atributos)

```
class A {
  int atributo;
  void saluda() { System.out.println("HOLA"); }
}
class B extends A { }
```

El tipo B no solo esta obteniendo el mensaje "saluda" sino también un método para tratarlo. El conflicto de la herencia múltiple ocurre porque una clase derivada puede recibir:
- Varias implementaciones para un mismo mensaje.

- Varias copias de un mismo atributo.

Los interfaces al igual que las clases, son una forma de crear tipos. Se caracterizan porque solo **declara** a los mensajes:
- No tiene código asociado a los mensajes
- No tiene atributos

```
interface unInterface {
  void mensaje1();
  void mensaje2(char c, Object o);
}
```

Uso de interface
Al no tener implementación no se puede instanciar (new). Solo se pueden usar para crear nuevos tipos (clases o interfaces)

```
interface unInterface {
  void mensaje1();
  void mensaje2();
}

class UnaClase implements unInterface {
  void mensaje1() { System.out.println("1");}
  void mensaje2() { System.out.println("2");}
}
```

La clase "UnaClase" hereda los mensajes de "UnInterface" (se convierte en un subtipo). Sin embargo tendrá que darles una implementación a los mismos.

El ejemplo anterior se podría haber hecho derivando de una clase abstracta. Sin embargo no se podría haber derivado de dos o más a la vez.

```
interface A {
  void mensaje1();
  void mensaje2();
}
interface B {
  void mensaje3();
  void mensaje4();
}

class C implements A, B {
  void mensaje1() { System.out.println("1");}
  void mensaje2() { System.out.println("2");}
  void mensaje3() { System.out.println("3");}
  void mensaje4() { System.out.println("4");}
}
```

La razón por la que se pueden implementar varios interfaces es que, al no tener implementaciones, no habrá conflictos entre las mismas.
Si A y B tuvieran un mismo mensaje se fundirían en una misma implementación.

```
interface A {
  void mensaje1();
  void mensaje2();
}
interface B {
  void mensaje2();
  void mensaje3();
}
```

```
class C implements A, B {
  void mensaje1() { System.out.println("1");}
  void mensaje2() { System.out.println("2");}
  void mensaje3() { System.out.println("3");}
}
```

Por tanto a la hora de manipular objetos lo adecuado es requerir que cumplan un determinado interface (y no que sean de una determinada clase).

```
interface Comparable {
    boolean igualA(Comparable o);
    boolean menorQue(Comparable o);
}
```

Los interfaces también pueden usarse para crear nuevos interfaces.

```
interface A {
  void mensaje1();
  }
interface B extends A {
  void mensaje2();
  }
```

Una clase que herede del interface B deberá implementar ambos mensajes. De la misma manera tambien se permite herencia múltiple de interface y herencia repetida.

Herencia Múltiple de Interfaces

```
interface A {
  void mensaje1();
  }
interface B extends A {
  void mensaje2();
  }
interface C extends B {
  void mensaje3();
  }
interface D extends B,C {
  void mensaje4();
  }
```

La clase que implemente el interface D deberá implementar los cuatro mensajes.

Resumen

Clases
- Tipo que al extenderlo mediante herencia se obtienen sus mensajes y sus implementaciones (métodos y atributos).
- Inconveniente: Solo se puede derivar de una de ellas
- Ventaja: menor codificación al crear nuevos subtipos ya que los mensajes vienen con sus implementaciones.

Interfaces
- Tipo que al extenderlo mediante herencia se obtienen solamente mensajes.
- Ventaja: Se pueden heredar varios interfaces sin conflictos.
- Inconveniente: Hay que codificar el método de cada mensaje en cada subtipo.

► Preferiblemente se usarán interfaces antes que clases abstractas. De esta manera no se compromete a los objetos en una jerarquía determinada.
► Solo se usarán clases abtractas cuando se este seguro de que los objetos a manipular no necesitan participar de mas tipos y se desee reutilizar las implementaciones.

Controlar el Acceso a los Miembros de la Clase

Uno de los beneficios de las clases es que pueden proteger sus variables y métodos miembros frente al acceso de otros objetos. ¿Por qué es esto importante?
Ciertas informaciones y peticiones contenidas en la clase, las soportadas por los métodos y variables accesibles públicamente en su objeto son correctas para el consumo de cualquier otro objeto del sistema. Otras peticiones contenidas en la clase son sólo para el uso personal de la clase. Estas otras soportadas por la operación de la clase no deberían ser utilizadas por objetos de otros tipos. Se querría proteger esas variables y métodos personales a nivel del lenguaje y prohibir el acceso desde objetos de otros tipos.

En Java se puede utilizar los especificadores de acceso para proteger tanto las variables como los métodos de la clase cuando se declaran. El lenguaje Java soporta cuatro niveles de acceso para las variables y métodos miembros: private, protected, public, y, todavía no especificado, acceso de paquete.

La siguiente tabla le muestra los niveles de acceso permitidos por cada especificador:

	clase	subclase	paquete	mundo
private	X			
protected	X	*	X	
public	X	X	X	X
package	X		X	

La primera columna indica si la propia clase tiene acceso al miembro definido por el especificador de acceso. La segunda columna indica si las subclases de la clase (sin importar dentro de que paquete se encuentren estas) tienen acceso a los miembros. La tercera columna indica si las clases del mismo paquete que la clase (sin importar su parentesco) tienen acceso a los miembros. La cuarta columna indica si todas las clases tienen acceso a los miembros.

La intersección entre protected y subclase tiene un '*' - este caso de acceso particular tiene una explicación en más detalle más adelante.

Niveles de acceso

Private: Es nivel de acceso más restringido. Un miembro privado es accesible sólo para la clase en la que está definido. Se utiliza este acceso para declarar miembros que sólo deben ser utilizados por la clase. Esto incluye las variables que contienen información que si se accede a ella desde el exterior podría colocar al objeto en un estado de inconsistencia, o los métodos que llamados desde el exterior pueden ponen en peligro el estado del objeto o del programa donde se está ejecutando. Los miembros privados son como secretos. Para declarar un miembro privado se utiliza la palabra clave private en su declaración. La clase siguiente contiene una variable miembro y un método privados:

```
class Alpha {
    private int soyPrivado;
    private void metodoPrivado() {
        System.out.println("metodoPrivado");
    }
}
```

Los objetos del tipo Alpha pueden inspeccionar y modificar la variable soyPrivado y pueden invocar el método metodoPrivado(), pero los objetos de otros tipos no pueden acceder. Por ejemplo, la clase Beta definida aquí:

```
class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.soyPrivado = 10;      // ilegal
        a.metodoPrivado();      // ilegal
    }
}
```

no puede acceder a la variable soyPrivado ni al método metodoPrivado() de un objeto del tipo Alpha porque Beta no es del tipo Alpha.
Si una clase esta intentando acceder a una variable miembro a la que no tiene acceso--el compilador mostrará un mensaje de error similar a este y no compilará su programa:

```
Beta.java:9: Variable iamprivate in class Alpha not accessible from class Beta.
        a.iamprivate = 10;     // ilegal
         ^
1 error
```

Y si un programa intenta acceder a un método al que no tiene acceso, generará un error de compilación parecido a este:

```
Beta.java:12: No method matching privateMethod() found in class Alpha.
        a.privateMethod();         // ilegal
1 error
```

Protected : El nivel de acceso es 'protected' permite a la propia clase, las subclases (con la excepción a la que nos referimos anteriormente), y todas las clases dentro del mismo paquete que accedan a los miembros. Este nivel de acceso se utiliza cuando es apropiado para una subclase da la clase tener acceso a los miembros, pero no las clases no relacionadas. Los miembros protegidos son como secretos familiares - no importa que toda la familia lo sepa, incluso algunos amigos allegados pero no se quiere que los extraños lo sepan.
Para declarar un miembro protegido, se utiliza la palabra clave protected. Veamos cómo afecta este especificador de acceso a las clases del mismo paquete.
Consideremos esta versión de la clase Alpha que ahora se declara para estar incluida en el paquete Griego y que tiene una variable y un método que son miembros protegidos:

```
package Griego;

class Alpha {
    protected int estoyProtegido;
    protected void metodoProtegido() {
        System.out.println("metodoProtegido");
    }
}
```

Ahora, supongamos que la clase Gamma, también está declarada como miembro del paquete Griego (y no es una subclase de Alpha). La Clase Gamma puede acceder legalmente al miembro estoyProtegido del objeto Alpha y puede llamar legalmente a su método metodoProtegido():

```
package Griego;

class Gamma {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.estoyProtegido = 10;    // legal
        a.metodoProtegido();    // legal
    }
}
```

Esto es muy sencillo. Pero cómo afecta el especificador protected a una subclase de Alpha. Introduzcamos una nueva clase, Delta, que desciende de la clase Alpha pero reside en un paquete diferente - Latin. La clase Delta puede acceder tanto a estoyProtegido como a metodoProtegido(), pero solo en objetos del tipo Delta o sus subclases. La clase Delta no puede acceder a estoyProtegido o metodoProtegido() en objetos del tipo Alpha. metodoAccesor() en el siguiente ejemplo intenta acceder a la variable miembro estoyProtegido de un objeto del tipo Alpha, que es ilegal, y en un objeto del tipo Delta que es legal. Similarmente, metodoAccesor() intenta invocar a metodoProtegido() en un objeto del tipo Alpha, que también es ilegal:

```
import Griego.*;

package Latin;
class Delta extends Alpha {
```

```
    void metodoAccesor(Alpha a, Delta d) {
        a.estoyProtegido = 10;    // ilegal
        d.estoyProtegido = 10;    // legal
        a.metodoProtegido();      // ilegal
        d.metodoProtegido();      // legal
    }
}
```

Si una clase es una subclase o se cuenta en el mismo paquete de la clase con el miembro protegido, la clase tiene acceso al miembro protegido.

Public: El especificador de acceso más sencillo es 'public'. Todas las clases, en todos los paquetes tienen acceso a los miembros públicos de la clase. Los miembros públicos se declaran sólo si su acceso no produce resultados indeseados si un extraño los utiliza. Para declarar un miembro público se utiliza la palabra clave public. Por ejemplo,

```
package Griego;

class Alpha {
    public int soyPublico;
    public void metodoPublico() {
        System.out.println("metodoPublico");
    }
}
```

Al reescribir la clase Beta una vez más y la colocarla en un paquete diferente que la clase Alpha y asegurarse que no están relacionadas (no es una subclase) de Alpha:

```
import Griego.*;

package Romano;
class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.soyPublico = 10;       // legal
        a.metodoPublico();       // legal
    }
}
```

Como se puede ver en el ejemplo anterior, Beta puede inspeccionar y modificar legalmente la variable soyPublico en la clase Alpha y puede llamar legalmente al método metodoPublico().

Acceso de Paquete
El último nivel de acceso es el que se obtiene si no se especifica ningún otro nivel de acceso a los miembros. Este nivel de acceso permite que las clases del mismo paquete que la clase tengan acceso a los miembros. Este nivel de acceso asume que las clases del mismo paquete son amigas de confianza. Este nivel de confianza es como la que extiende a sus mejores amigos y que incluso no la tiene con su familia.
Por ejemplo, esta versión de la clase Alpha declara una variable y un método con acceso de paquete. Alpha reside en el paquete Griego:

```
package Griego;

class Alpha {
    int estoyEmpaquetado;
    void metodoEmpaquetado() {
        System.out.println("metodoEmpaquetado");
    }
}
```

La clase Alpha tiene acceso a estoyEmpaquetado y a metodoEmpaquetado(). Además, todas las clases declaradas dentro del mismo paquete como Alpha también tienen acceso a estoyEmpaquetado y metodoEmpaquetado(). Supongamos que tanto Alpha como Beta son declaradas como parte del paquete Griego:

```
package Griego;

class Beta {
    void metodoAccesor() {
        Alpha a = new Alpha();
        a.estoyEmpaquetado = 10;     // legal
        a.metodoEmpaquetado();     // legal
    }
}
```

Entonces Beta puede acceder legalmente a estoyEmpaquetado y metodoEmpaquetado().

Sobrecarga

- Se produce cuando existen en la misma clase dos o mas métodos con el mismo identificador (nombre) pero con diferentes listas de parámetros (en número y/o en tipo).
- Cada método realiza una operación en función de los argumentos que recibe. Semánticamente la sobrecarga resuelve el problema de que una determinada función se puede realizar de diferentes formas (algoritmos) con distintos datos (parámetros)
- La sobrecarga se resuelve en tiempo de compilación, dado que los métodos se deben diferenciar en el tipo o en el número de parámetros.

Ejemplo: supongamos una clase Punto que se utilizara para definir la clase Rectángulo.

```
class Punto
{
     int x,y;

    Punto(int x, int y)
    {
     this.x = x;
     this.y = y;
    }
    void getX()
    {
     return x;
    }
    void getY()
    {
     return y;
    }
    ---
 }
```

Un rectángulo se define por 4 puntos. Hay una agregación, es decir un rectángulo se compone de 4 puntos. Pero esto se puede crear de diferentes maneras:

a) se tienen los 4 puntos
b) se tiene el punto superior izquierdo, el ancho y alto y mediante cálculos se obtiene el valor de los puntos restantes.
c) se tienen los puntos superior izquierdo e inferior derecho y mediante cálculos se obtiene el valor de los puntos restantes.

```
class Rectangulo
{
     Punto si, sd, ind, ii;

    Rectangulo(Punto p1, Punto p2, Punto p3, Punto p4)
    {
     si = p1;
     sd = p2;
     ii = p3;
     id = p4;
```

```
        }

        Rectangulo(Punto p1, int ancho, int alto)
        {
         si = p1;
         sd = new Punto(si.getX(),si.getY()+ancho);
         ii = new Punto (si.getX()+alto,si.getY());
         id = new Punto(ii.getX(),ii.getY()+ancho);
        }

        Rectangulo(Punto p1, Punto p2)
        {
         si = p1;
         id = p2;
         int ancho = p2.getY()-p1.getY();
         int alto = p2.getX()-p1.getX();
         sd = new Punto(si.getX(),si.getY()+ancho);
         ii = new Punto (si.getX()+alto,si.getY());
        }
        ---
    }
```

El método constructor de la clase Rectangulo esta sobrecargado, es decir un rectángulo se puede construir de 3 maneras diferentes. En cada método, la implementación (algoritmo) como la lista de parámetros es diferente, pero el resultado es el mismo.

Operador this

Al acceder a variables de instancia de una clase, la palabra clave this hace referencia a los miembros de la propia clase en el objeto actual; es decir, this se refiere al objeto actual sobre el que está actuando un método determinado y se utiliza siempre que se quiera hace referencia al objeto actual de la clase. Volviendo al ejemplo de MiClase, se puede añadir otro constructor de la forma siguiente:

```
class MiClase {
   int i;
   MiClase() {
      i = 10;
   }
     // Este constructor establece el valor de i
   MiClase( int valor ) {
      this.i = valor;       // i = valor
   }
     // Este constructor también establece el valor de i
   MiClase( int i ) {
      this.i = i;
   }
   void Suma_a_i( int j ) {
      i = i + j;
   }
}
```

Aquí this.i se refiere al entero i en la clase MiClase, que corresponde al objeto actual. La utilización de this en el tercer constructor de la clase, permite referirse directamente al objeto en sí, en lugar de permitir que el ámbito actual defina la resolución de variables, al utilizar i como parámetro formal y después this para acceder a la variable de instancia del objeto actual.

La utilización de this en dicho contexto puede ser confusa en ocasiones, y algunos programadores procuran no utilizar variables locales y nombres de parámetros formales que ocultan variables de instancia. Una filosofía diferente dice que en los métodos de inicialización y constructores, es bueno seguir el criterio de utilizar los mismos nombres por claridad, y utilizar this para no ocultar las variables de instancia. Lo cierto es que es más una cuestión de gusto personal que otra cosa el hacerlo de una forma u otra.

La siguiente aplicación de ejemplo, utiliza la referencia this al objeto para acceder a una variable de instancia oculta para el método que es llamado.

```
class java509 {
      // Variable de instancia
      int miVariable;

      // Constructor de la clase
      public java509() {
          miVariable = 5;
          }

      // Metodo con argumentos
      void miMetodo(int miVariable) {
          System.out.println( "La variable Local miVariable contiene "
              + miVariable );
          System.out.println(
              "La variable de Instancia miVariable contiene "
              + this.miVariable );
          }

      public static void main( String args[] ) {
          // Instanciamos un objeto del tipo de la clase
          java509 obj = new java509();
          // que utilizamos para llamar a su unico metodo
          obj.miMetodo( 10 );
          }
      }
```