Mecanismos de abstracción. Herencia

Introducción

La verdadera potencia de la POO radica en su capacidad para reflejar la abstracción que el cerebro humano realiza automáticamente durante el proceso de aprendizaje y el proceso de análisis de información. Desde la temprana infancia los seres humanos adquieren capacidades de abstracción: aprenden a clasificar individuos dentro de especies, aprecian que ciertas cosas están compuestas de otras más pequeñas y también que hay parecidos entre especies. Todos estos mecanismos de abstracción fueron transportados al paradigma de objetos. El más interesante es la herencia, basado en el mecanismo de generalización.

Los defensores de la POO afirman que esta técnica se adecua mejor al funcionamiento del cerebro humano. La capacidad de descomponer un problema o concepto en un conjunto de objetos relacionados entre sí, y cuyo comportamiento es fácilmente identificable, puede ser muy útil para el desarrollo de programas informáticos.

En POO la representación del conocimiento se hace mediante clases. Una clase representa un concepto en el dominio del problema. Su propósito es brindar un mayor nivel de abstracción que los objetos. Las clases se usan para representar parte del conocimiento adquirido del dominio del problema.

Puede darse la situación de que las clases compartan parte del conocimiento al cual representan. Y ante esta situación, es necesario un mecanismo que permita reunir ese conocimiento común en un solo sitio, para que luego pueda ser reutilizado por las clases que lo compartan. Es así como surge la noción de subclasificación, potente mecanismo utilizado en este paradigma.

Mecanismos de abstracción

Abstracción significa organizar la realidad para hacerla más aprehensible. Centrar la atención en las características relevantes al problema a resolver, y **disponerlas del modo más conveniente para su mejor comprensión**. Existen diferentes mecanismos para lograr este propósito:

- **Clasificación**: es el más habitual, y permite definir un objeto como perteneciente a un tipo o clase mediante el mecanismo de instanciación. Es el que permite definir a "Corrientes" como una provincia y a "María" como una persona.
- **Generalización**: permite definir una clase o tipo como un caso general de varias clases o tipos. Está relacionado con el concepto de especies o familias. Por ejemplo, se puede definir cuadriláteros a partir de los paralelogramos, rectángulos y rombos. La operación inversa se llama especialización, y es la que permite definir clases como casos particulares de otras más generales.
- **Agregación**: permite definir a un objeto como una simple agrupación de otros. Por ejemplo, podríamos definir a un curso como el conjunto de los alumnos y un profesor. En esencia, representa la relación de un objeto con otros, teniendo los segundos existencia propia. Ejemplo: un objeto "libro" tiene datos de su autor, almacenados en un objeto "autor". Sin embargo, la eliminación de un libro no implica la desaparición del objeto autor, que puede tener otros libros escritos.
- **Composición**: similar al mecanismo anterior, permite definir un objeto como compuesto por otros objetos, de modo que si el objeto compuesto deja de existir los objetos componentes dejan de tener sentido. Es el que permite definir a una empresa como un conjunto de elementos empleados. Si la empresa deja de existir, los empleados dejan de ser tales.

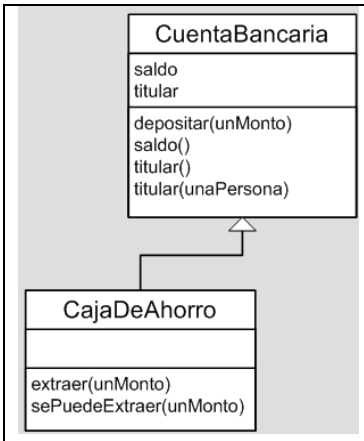
Existen diferencias importantes entre estos mecanismos de abstracción:

- La clasificación es una relación entre un objeto y una clase (un objeto es una instancia de una clase). Es una relación conceptual, porque hay que basarse en el **conocimiento** de características y comportamiento del objeto para ubicarlo en una clase. Ejemplo: si el objeto tiene cuatro patas y ladra, pertenece a la clase perro.
- La agregación y la composición son relaciones entre objetos (un objeto es una agrupación de otros). Dos objetos distintos están involucrados, uno es parte del otro. Son relaciones físicas, porque no depende de la definición de las características de cada objeto, sino que indica que ciertos objetos ***están contenidos en*** o ***son referenciados por*** otro objeto.

- La generalización es una relación entre clases (una clase es un caso general de varias clases). Es una manera de estructurar la descripción de **un único** objeto. Tanto la superclase como la subclase referencian las propiedades de un único objeto. En una generalización, un objeto es simultáneamente una instancia de la superclase y de la subclase. Por ejemplo, en una generalización Figura/Cuadrilátero, un objeto cuadrilátero es también una figura. Es una relación conceptual, ya que determina si una clase es un caso especial o general de otra en base al **conocimiento** de sus características y comportamiento.

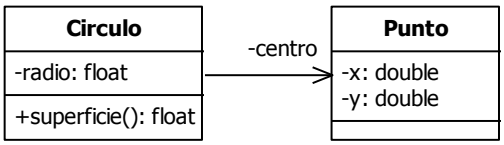
Tanto la relación de agregación como la de generalización forman jerarquías. Una jerarquía de agregación está compuesta de instancias de **objetos** que son parte de un objeto compuesto. Una jerarquía de herencia está compuesta de **clases** que describen un objeto.

Mecanismo de Abstracción	Tipo de relación	Participan	Jerarquía
Clasificación	conceptual	un objeto y una clase	--
Generalización/especializacion	conceptual	clases	Compuesta de **clases** que describen **un objeto**
Agregación y Composición	físicas	objetos	Compuesta con **dos o más objetos**



Generalización/especialización: la jerarquía está compuesta de **clases** que describen **un objeto**

Ejemplo: el objeto en este caso es **una caja de ahorro**, que a su vez es una cuenta bancaria



Agregación y Composición: la jerarquía está compuesta con **dos o más objetos**

Ejemplo: el objeto círculo está compuesto por **un objeto punto**

Reutilización y mecanismos de abstracción

La ventaja de los mecanismos de abstracción es que favorecen la reutilización del código de las clases.

Se llama **reutilización** al uso de clases u objetos desarrollados y probados en un determinado contexto, para incorporar esa funcionalidad en una aplicación diferente a la de origen. Se basa en aprovechar las clases desarrolladas para una aplicación, utilizándolas en la construcción de nuevas clases para la misma u otra aplicación.

Todos los mecanismos de abstracción mencionados son herramientas claves para lograr la reutilización:

1. Clasificación

Es la forma más simple de **reutilización**. Es simplemente hacer uso de las clases como están, mediante instanciación. Por ej.: cada vez que se instancia un objeto punto, se reutiliza la definición de la clase Punto.

```
public class Punto {
  private double x;
  private double y;

        // contenido de la clase
}

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

        Punto punto_1 = new Punto(2, 3);
        Punto punto_2 = new Punto(0, 5.3);
        ...
        Punto punto_n = new Punto(-2, -4.6);
}
```

2. Agregación/Composición

Es una relación estructural entre objetos: un objeto se construye a partir de otros. En el diagrama de clases, una clase tiene en su estructura a otra clase, es decir, se construye una clase a partir de otras. Al momento de codificar, esta relación se representa como un atributo que es una instancia de otra clase.

Este mecanismo se debe entender como la actividad de construir elementos complejos a partir de otros elementos más simples, que es justamente la verdadera esencia de la POO.

Es una relación que representa objetos compuestos. Los objetos son **reutilizados** en distintas "composiciones".

Dos objetos tienen una relación de agregación si existe entre ellos una relación **todo-parte, continente-contenido**.

Representación en *UML*:

Agregación **Composición**

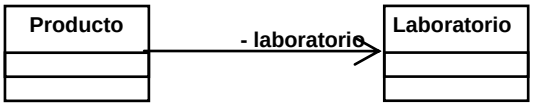
Esta relación se lee ***Parte*** "es parte de" ***Todo*** ó ***Todo*** "está-formado-por" ***Parte.*** En ella los objetos que representan los componentes de algo son asociados a un objeto que representa el todo. [Rumbaugh]

Cada objeto parte tiene su propio comportamiento y el objeto compuesto es tratado como una unidad cuya responsabilidad es realizar la coordinación de las partes.

Esta relación también puede ser vista desde el otro sentido, como "tiene un". Por ejemplo: un documento tiene un (o más) párrafo, que a su vez está formado por oraciones. Una factura tiene una (o más) línea, que a su vez está formada por un artículo, una cantidad y un precio.

Esta relación representa **el conocimiento de un objeto con sus variables de instancia**. Es un modo de indicar gráficamente, por ejemplo, que un Producto tiene-un laboratorio, o que una factura tiene líneas de venta.

Frecuentemente la relación de conocimiento por variables de instancia es representada gráficamente con las clases unidas por una flecha cuya punta abierta apunta a la clase contenida:

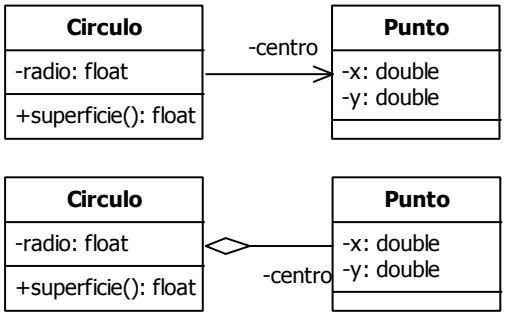


Las partes pueden o no existir fuera del objeto compuesto o pueden aparecer en varios objetos compuestos. Por lo tanto, la existencia del objeto componente (LineaVenta) puede depender de la existencia del objeto compuesto (Factura) del cual forma parte, o en otro caso, el objeto componente (Laboratorio) puede tener una existencia independiente. En el primer caso se habla de una "composición", que es una

relación de agregación más fuerte, como el caso de una factura y sus líneas de venta: si la factura desaparece, las líneas dejan de tener sentido. En cambio el Laboratorio tiene existencia independiente de Producto, y puede ser utilizado como parte de otros objetos, por ejemplo, en un libro de pagos a proveedores. Por otra parte, puede pensarse también que si el producto deja de existir, el laboratorio continúa elaborando otros productos.

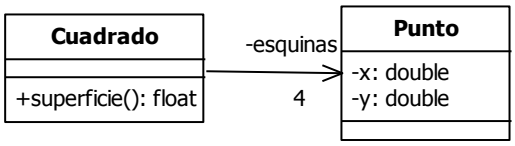
Ejemplo 1: un Circulo *tiene-un* Punto

```
public class Circulo {
   private Punto centro;
   private float radio;

   Circulo(Punto p_centro, float p_radio){
       ...
   }

   public float superficie() {
       return 3.14F * radio * radio;
   }
}
```



Ejemplo 2: un Cuadrado *tiene-cuatro* Punto (s)

```
public class Cuadrado{
   private Punto esquinas[];

   Cuadrado(Punto p_esquinas[]){
      this.esquinas]= new Punto[4];
      this.setEsquinas(p_esquinas);
   }

   public float superficie() {
       // ** efectuar calculo **;
   }
}
```



3. Generalización/Subclasificación

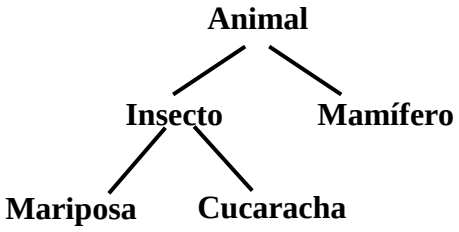
La generalización consiste en reunir el comportamiento y la estructura común en una clase, la cual cumplirá el rol de *superclase*. Otras clases pueden cumplir el rol de *subclases*, heredando ese comportamiento y estructura en común. Estas clases también reciben el nombre de clases derivadas, o hijas.

Generalización y especialización corresponden a dos puntos de vista diferentes de la misma relación, desde la perspectiva de las superclases o de las subclases. La palabra generalización deriva del hecho de que la superclase generaliza las subclases. Especialización se refiere al hecho de que las subclases refinan o especializan la superclase.

Se conforma así una jerarquía de clases. Los elementos de la jerarquía cumplen la relación ***es-un*** (también conocida como relación **IsA**).

En este ejemplo de subclasificación, todo insecto y todo mamífero **es-un** animal. Toda mariposa y toda cucaracha **es-un** insecto.

Así mismo, puede observarse la transitividad, dado que toda mariposa a su vez **es-un** animal



Toda instancia de una clase también *es-una* instancia de su superclase, en el sentido de que responde al mismo comportamiento o protocolo.

El hecho de heredar implica que se **reutiliza** todo lo ya definido. La herencia es una técnica clave de reusabilidad. Sin ella, cada clase debería implementar completamente todos los servicios que ofrece. La herencia provee la posibilidad de que todos los servicios de una superclase puedan ser reutilizados por la subclase, sin necesidad de definirlos nuevamente.

Herencia

Al realizar un análisis del dominio de un problema, la estructura interna y el comportamiento que son iguales para clases diferentes tenderá a migrar hacia superclases comunes. Este es el motivo por el cual se dice que la herencia es una jerarquía de generalización/especialización. Habitualmente se utiliza *generalización* para referirse a la relación entre clases, mientras que *herencia* se refiere al mecanismo de compartir atributos y métodos utilizando la relación de *generalización*.

Las superclases representan abstracciones generalizadas, y las subclases representan especializaciones en las cuales atributos y métodos de la superclase son agregados, modificados o aun ocultados. Por lo tanto, la herencia representa una jerarquía de abstracciones.

Ejemplo: en el dominio de un sistema bancario, al realizar el análisis, se consideran en principio dos tipos de cuenta: las cajas de ahorro y las cuentas corrientes. Estas, además de compartir una estructura básica (titular, saldo), tienen un comportamiento similar: ambas permiten realizar depósitos y extracciones.

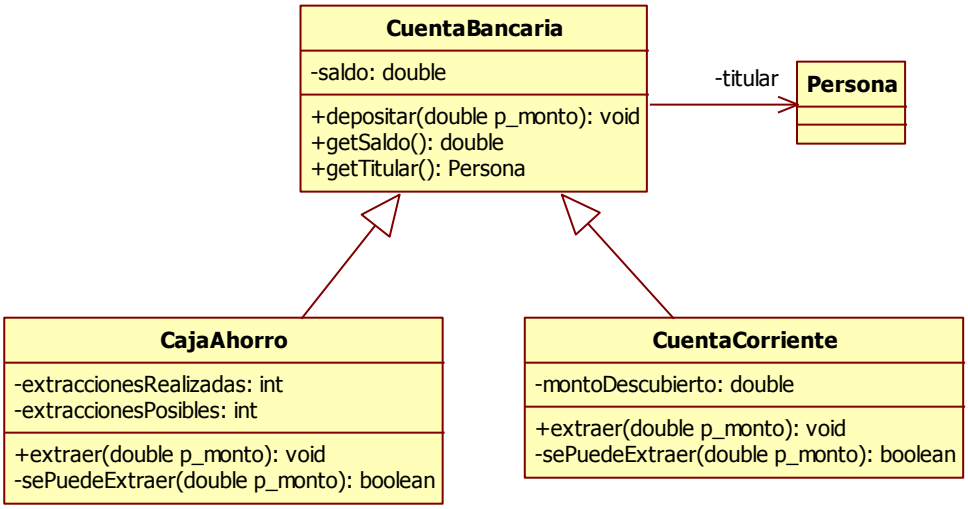
Sin embargo, cada una de las cuentas tiene un tipo de restricción distinto en cuanto a las extracciones. Las cuentas corrientes permiten que el cliente gire en descubierto, con un tope pactado con cada cliente, mientras que las cajas de ahorro poseen una cantidad máxima de extracciones mensuales (que es igual para todos los clientes) y además no permiten girar en descubierto.

¿Qué se puede hacer para reutilizar el comportamiento común a las dos cuentas?

Se puede hacer uso de la generalización y establecer una jerarquía de herencia para resolver este problema: la solución consiste en crear una jerarquía de clases de cuentas bancarias. Tanto las cajas de ahorro como las cuentas corrientes son cuentas bancarias, y esto queda reflejado en la jerarquía.

En la clase *CuentaBancaria* está el comportamiento común a todas las cuentas, mientras que la *CajaDeAhorro* especializa su comportamiento, definiendo su método *extraer(unMonto)*, al igual que la *CuentaCorriente*.

Además, cada una agrega atributos, que le serán necesarios a la hora de resolver su comportamiento, como ser el número de extracciones realizadas y posibles, y el monto de giro descubierto autorizado.



Al crear una jerarquía de clases, se debe respetar la relación *es-un* entre una clase y su superclase. Este concepto es fundamental a la hora de diseñar jerarquías de clase. Por ejemplo, una caja de ahorro *es-una* cuenta bancaria.

La herencia permite:
• Abstraer las similitudes
• Crear una nueva clase como refinamiento de otra (subclase)
• Diseñar e implementar sólo la diferencia que presenta la subclase

Para diseñar la jerarquía se pueden aplicar dos metodologías:

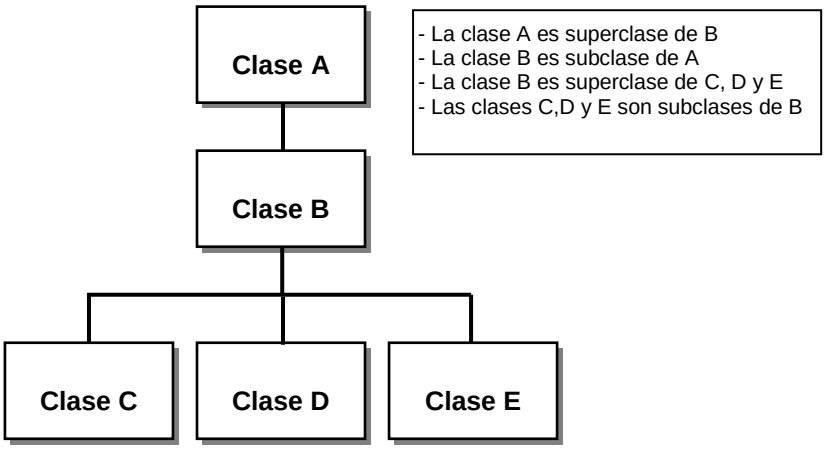
- **Top-down:** en el cual se plantea primero el concepto más abstracto en una superclase y luego, por medio de un proceso de refinación se crean las subclases. Es decir, se parte de un concepto general y luego se especializa en uno o más conceptos particulares.
- **Bottom-up:** en el cual se modelan los conceptos en forma separada y luego, al encontrar similitudes en el comportamiento de los objetos, se realiza un proceso de generalización, en el cual se crean abstracciones que engloban las clases previamente modeladas.

Herencia simple

Es el mecanismo por el cual una clase puede extender las características de otra, pero puede heredar o tomar atributos de **una sola** clase padre.

Según lo expresado, la herencia permite reutilizar tanto comportamiento como estructura. Esto significa que, si la clase B es subclase de A, B heredará todas las variables de instancia y los métodos definidos en A.

La herencia tiene carácter transitivo: si B es subclase de A y C es subclase de B, entonces C entiende los mensajes y atributos definidos en B, mas los atributos y mensajes definidos en A.



A la hora de subclasificar es fundamental tener en cuenta que lo importante es basarse en el comportamiento y no en la estructura. Si se hace foco en los mensajes a los que debe responder cada clase se asegura mantener el principio *es-un* en la jerarquía.

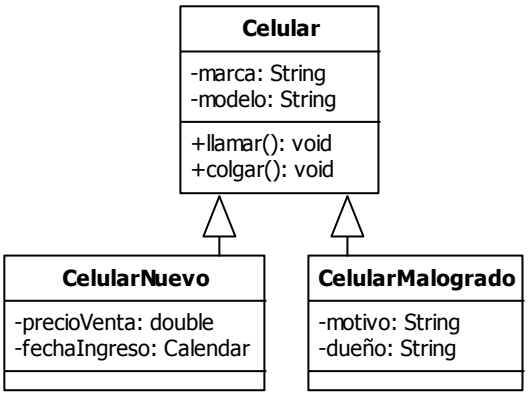
A pesar que el mecanismo de herencia permite reutilizar tanto estructura como comportamiento, existen algunas diferencias entre éstos:

Herencia de comportamiento:
- Una subclase hereda todos los métodos definidos en su superclase.
- **Una subclase puede redefinir un método heredado, refinando la forma en la que se comportan sus instancias ante el envío de ese mensaje.**
- Una subclase no puede anular un método que esté definido en su superclase. Notar que eso es el equivalente a redefinir un método para que dispare un error, lo cual, a pesar de ser posible en la práctica, es conceptualmente incorrecto, ya que significaría que una subclase no *es-un* de su superclase.

Herencia de estructura:
- Una subclase hereda todas las variables de instancia definidas en su superclase.
- **Una subclase no puede redefinir un atributo o colaborador. Esto llevaría a tener que utilizar dos variables de instancia con el mismo nombre, lo cual sería ambiguo.**
- Una subclase no puede anular la herencia de una variable de instancia.

Notación UML

En UML la herencia se representa mediante una flecha con punta cerrada y sin relleno, que parte de la subclase y que apunta a la superclase. Por ejemplo, en el análisis de un sistema para una tienda que vende y repara equipos celulares, la representación podría ser la de la figura adjunta. En el gráfico se observan dos clases que se extienden de la clase Celular.

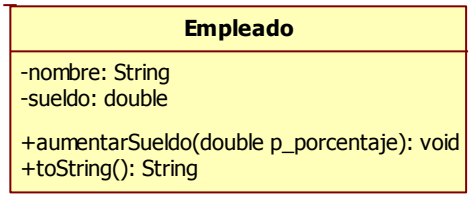


Los lenguajes de POO proveen el modo de indicar la herencia entre clases. Por ejemplo, en java, para indicar que una clase deriva de otra, se usa el término *extends*.

```
public class Celular{
   private String marca;
   private String modelo;

  public void llamar(){
       // contenido del método
  }
  public void colgar(){
       // contenido del método
  }
}

public class CelularMalogrado extends Celular{
   private String motivo;
   private String dueño;

       // contenido de la clase
}

public class CelularNuevo extends Celular{
   private int precioVenta;
   private Calendar fechaIngreso;

       // contenido de la clase
}
```

La palabra clave *extends* se utiliza para indicar que se desea crear una subclase de la clase que es nombrada a continuación.

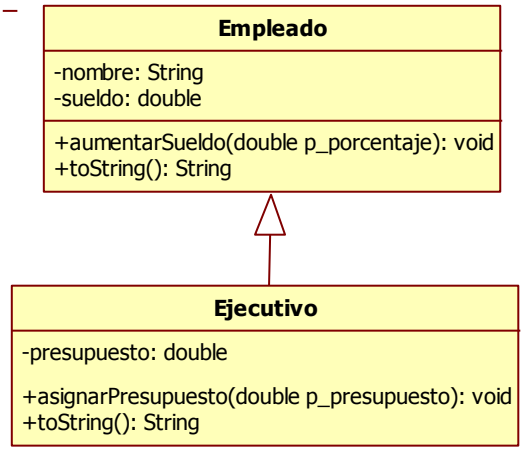
Ejemplo: Una empresa define la clase Empleado, con las características generales (nombre y sueldo), que permite calcular un aumento de sueldo, y devuelve una representación de los datos del empleado en una cadena.




```
      public class Empleado {
          private String nombre;
          private double sueldo;

          Empleado(String p_nombre, double p_sueldo) {
              this.setNombre(p_nombre);
              this.setSueldo(p_sueldo);
          }


      // *** definir accessors ***

          public void aumentarSueldo(double p_porcentaje) {
              this.setSueldo(this.getSueldo() + (this.getSueldo() * p_porcentaje));
          }

          public String toString() {
              return this.getNombre() + " -$" + this.getSueldo();
          }
}
```

Luego la empresa necesita manipular un tipo de empleado *Ejecutivo*, que reune todas las características de *Empleado*, pero que tiene asignado un presupuesto para el desempeño del departamento que dirige. Se dice que la clase *Ejecutivo extiende* o hereda la clase *Empleado*. La definición de la clase es la siguiente:

```
public class Ejecutivo extends Empleado{
   private double presupuesto;

   // Atención !!  Falta definir el constructor (se verá más
adelante)

   public void asignarPresupuesto(double
p_presupuesto) {
         this.presupuesto = p_presupuesto;
    }
}
```



Con esta definición, un Ejecutivo *es-un* Empleado que además tiene un rasgo distintivo propio. El cuerpo de la clase Ejecutivo incorpora sólo los miembros que son específicos de esta clase, pero implícitamente tiene todo lo que tiene la clase Empleado. Empleado es la clase base o superclase y Ejecutivo es la clase derivada o subclase.

Los objetos de las clases derivadas se crean igual que los de la clase base y pueden acceder tanto a sus datos y métodos como a los de la clase base. Por ejemplo:

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

       Ejecutivo jefe = new Ejecutivo( "Juan Perez", 6000);
       jefe.asignarPresupuesto(1500);
       jefe.aumentarSueldo(0.05);
}
```

<u>Nota</u>: el uso de los constructores en una jerarquía de herencia se verá un poco más adelante.

Un Ejecutivo *es-un* Empleado, por tanto entiende el mensaje *aumentarSueldo*. Sin embargo lo contrario no es cierto. Si se escribe:

```
Empleado emple = new Empleado ("José Romero", 1000);
emple.asignarPresupuesto(5000);  // error
```

se producirá un error de compilación pues en la clase Empleado no existe ningún método llamado *asignarPresupuesto*.

Redefinición de métodos

En ocasiones es necesario redefinir métodos de la super clase, de modo que métodos con el mismo nombre y características se comporten de forma distinta en la subclase (manteniendo la semántica. Es decir, el método *extraer* tendrá siempre el mismo significado, aunque se realice de distinta manera).

Para redefinir un método en la subclase, basta con declarar un método miembro con la misma firma. Por ejemplo, el método *toString()* de la clase Empleado puede añadir las características propias de la clase Ejecutivo:

```
public class Ejecutivo extends Empleado{
   private double presupuesto;

   public void asignarPresupuesto(double p_presupuesto) {
        this.presupuesto = p_presupuesto;
   }

   public String toString() {
        String s = super.toString();
        s = s + " Presupuesto: " + this.getPresupuesto();
        return s;
   }
}
```

Se observa en este ejemplo en lenguaje Java el uso de la seudovariable `super`, que representa referencia interna implícita a la clase base (superclase). Mediante `super.toString()` se invoca el método *toString()* de la clase Empleado. Sin embargo, cuando se invoque `jefe.toString()` se usará el método *toString()* de la clase Ejecutivo en lugar del existente en la clase Empleado.

Constructores y destructores considerando herencia

Para asegurar que toda la asignación de memoria e inicialización se hace en la forma correcta, algunos lenguajes de programación, dentro del constructor de una subclase llaman en primer lugar al *constructor* de la clase ancestro inmediato, y dentro de un destructor, llaman al final al *destructor* de la clase ancestro inmediato.

En Java la primera regla se cumple en forma automática, en el caso del **constructor por defecto**, por lo que no es necesario incluir la invocación al constructor del ancestro, y se crea implícitamente un objeto de la clase base, que se inicializa con su constructor correspondiente. Como en Java no hay destructores, la segunda regla no se aplica.

Sin embargo, si en la clase ancestro hay más de un constructor, o éste tiene parámetros, en el constructor de la subclase es necesario invocarlo explícitamente, usando la seudovariable *super*, sin indicar el nombre de método (asume el constructor).

En el ejemplo dado es necesario un constructor para la clase Ejecutivo, que podemos completar así:

```
public class Ejecutivo extends Empleado{
   private double presupuesto;

    Ejecutivo (String p_nombre, double p_sueldo, double p_presupuesto) {
        super(p_nombre, p_sueldo); // invoca al constructor de la superclase
        this.asignarPresupuesto(p_presupuesto);
    }
    public void asignarPresupuesto(double p_presupuesto) {
        this.presupuesto = p_presupuesto;
    }

    public String toString() {
        String s = super.toString();
        s = s + " Presupuesto: " + this.getPresupuesto();
        return s;
    }
}
```

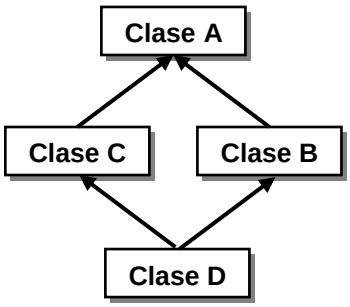
Herencia múltiple

Cuando una clase tiene más de una superclase, combinando comportamiento de varias fuentes, y agregando solo alguna porción, para producir su propio y único tipo de objeto, se dice que hay una herencia múltiple. Por ejemplo: la clase A hereda de las clases B1, B2, ..., Bn..

La herencia múltiple permite a una clase tomar funcionalidades de otras clases. Por ejemplo, la clase *MusicoOficinista* puede heredar de una clase llamada *Musico* y una clase llamada *Trabajador*.

Esto puede presentar **conflictos de nomenclatura**, si las superclases definen propiedades con el mismo nombre (atributos o métodos). Por ejemplo, el atributo "nombre" en Trabajador y en Músico (considerando que los músicos en ocasiones usan un nombre artístico diferente de su nombre real). ¿Qué atributo debería ser heredado?

Un tipo especial de conflicto de nomenclatura se presenta si una clase *D* hereda en forma múltiple de las superclases *B* y *C* que a su vez son derivadas de una superclase *A*. Esto es conocido como el **problema del diamante,** debido a que presenta una gráfica de herencia como se muestra en la figura:



Cabe la pregunta acerca de qué propiedades hereda la clase *D* de sus superclases *B* y *C*, y si **dos** copias de las propiedades de *A*: una heredada de *B* y otra de *C*.

Continuando con el ejemplo, *Musico* y *Trabajador* podrían heredar de *Persona,* y en este caso el atributo "edad" de *Persona* sería heredado por ambos caminos.

Aunque la herencia múltiple es un poderoso mecanismo en orientación a objetos, los problemas que se presentan con los conflictos de nomenclatura han llevado a varios autores a condenarla. Estos problemas pueden aumentar cuando se fuerza el concepto de herencia en un caso que no es natural. Por ejemplo, en ocasiones se establecen relaciones de herencia múltiple sólo para reutilizar en una clase, capacidades de otras. Esto hace que el diseño de la aplicación sea difícil de comprender y de mantener, ya que no existe un "vínculo natural" entre clases. Una relación de herencia múltiple debe surgir sin forzarla.

Debido a estas ambigüedades, sólo algunos lenguajes implementan herencia múltiple, como ser C++, Eiffel, Perl y Python.

Lenguajes como Java, Ada, Objective-C y C# sólo permiten herencia simple. Java por su parte compensa de cierta manera la inexistencia de herencia múltiple con un concepto denominado *interfaces*.

Interfaces

Las interfaces son un mecanismo para declarar operaciones que deben implementar ciertas clases, separando claramente la interfaz de la implementación. El papel de las interfaces es el de describir algunas de las características de una clase. Por ejemplo, el hecho de que una persona sea un futbolista no define su personalidad completa, pero hace que tenga ciertas características que la distinguen de otras. Definen un comportamiento (o funcionalidad) genérico, ignorando los aspectos relacionados con su implementación.

En Java, las interfaces son expresiones puras de diseño. Se trata de auténticas conceptualizaciones no implementadas que sirven de guía para definir un determinado concepto y lo que debe hacer, pero sin desarrollar un mecanismo de solución. Supongamos, por ejemplo, que se desea definir la operación **imprimir**, de forma tal que pueda servir para muchas clases, aunque tenga distintas implementaciones en cada una. Por ejemplo, no es lo mismo imprimir una matriz que una fecha o un vector. Las interfaces proponen una solución elegante y consistente a este problema.

Una interfaz es un conjunto de operaciones que especifican un servicio. Sirve para encapsular un conjunto de métodos, sin asignar esta funcionalidad a ningún objeto en particular ni expresar nada respecto del código que las va a implementar. Como las operaciones de una interfaz no se implementan, se dice que una interfaz **expone métodos.**

En Java, para declarar una interfaz se utiliza la sentencia *interface*, de la misma manera que se usa la sentencia *class*:

```
interface Imprimible {
   public void imprimir();
}
```

Como las interfaces carecen de funcionalidad por no estar implementados sus métodos, es necesario algún mecanismo para dar cuerpo a sus métodos. A través de la palabra reservada *implements* utilizada en la declaración de una clase se indica que ésta implementa la interfaz, es decir, que se compromete a implementar (codifica) todos los métodos de la interfaz.

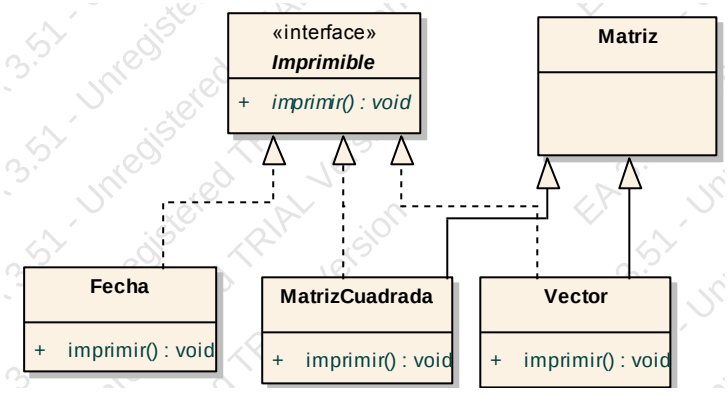
Para implementar una interfaz en una clase hay que especificarlo a continuación de la clase ancestro inmediato, colocando la palabra *implements*:

```
public class Triangulo extends Figura implements Imprimible {
    // declaraciones de atributos y métodos

    public void imprimir(){
          // codificar el modo de imprimirse de un triángulo
    }
}


public class Fecha implements Imprimible {
    // declaraciones de atributos y métodos

    public void imprimir(){
          // codificar el modo de imprimirse de una fecha
    }
}
```

Una clase puede implementar más de una interfaz, lo que permite algo parecido a la herencia múltiple:

```
public class A implements Iz1, Iz2, ... Izn {
    // declaraciones de atributos y métodos

    // implementar métodos de todas las interfaces
    }
}
```

En UML, las interfaces se representan con el estereotipo «interface». La implementación de una interfaz por una clase puede representarse como la herencia, aunque con línea punteada. La figura muestra usos de la interfaz Imprimible:



Las interfaces están presentes sólo en algunos lenguajes de POO, como Java, C# y Object Pascal. En otros, como Eiffel y C++, se implementa herencia múltiple.

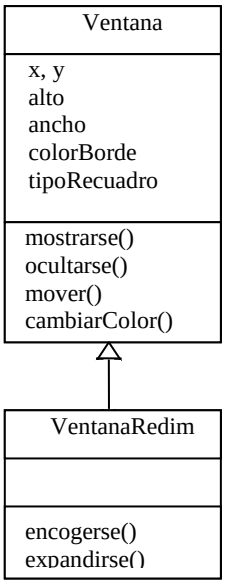
El OMG recomienda la implementación de interfaces para garantizar la interoperabilidad de objetos distribuidos. Por ello es que las tecnologías CORBA, COM+, JavaBeans, .NET, etc., hacen un uso intenso de interfaces.

Herencia y reutilización

En orientación a objetos la herencia es el mecanismo fundamental para implementar la reutilización y extensibilidad del software. A través de ella los diseñadores pueden construir nuevas clases partiendo de una jerarquía de clases ya existente (comprobadas y verificadas) evitando con ello el rediseño, la modificación y verificación de la parte ya implementada. Es decir, se reutiliza todo lo ya definido, por lo que herencia se ha transformado en sinónimo de reusabilidad de código dentro de la comunidad de orientación a objetos, y en consecuencia, una técnica clave en el proceso de construcción de software.

Esto favorece un estilo de desarrollo de software completamente diferente de los enfoques tradicionales. En vez de tratar de resolver cada problema nuevo desde cero, la idea es construir sobre logros previos y extender sus resultados. Con esto se logra **economía**, porque no se rehace lo que ya está hecho, como consecuencia de ello **productividad**, ya que se acortan los tiempos de desarrollo, y **robustez**, porque se construye a partir de componentes de software convenientemente probados y libres de error.

Ejemplo: se dispone de una clase Ventana de una aplicación de Windows. Se presenta la necesidad de contar tanto con ventanas de dimensiones fijas como de dimensiones variables. Por lo tanto, se deben definir dos clases distintas: ventana y ventana redimensionable. Como ambas clases tienen comportamiento y estructura interna comunes se define a la primera como superclase de la segunda.



Todo objeto de la clase Ventana tendrá dimensiones fijas especificadas en el momento de su creación (posición, alto, ancho, etc), y podrá mostrarse, ocultarse, moverse y cambiar de color.

Todo objeto de la clase VentanaRedim tendrá como estado interno heredado de Ventana: x, y, alto, ancho, colorBorde, tipoRecuadro; y tendrá como comportamiento heredado: mostrarse, ocultarse, mover, cambiarColor. Agregará como comportamiento propio *encogerse()* y *expandirse()*.

Del ejemplo se desprende que en la construcción de la segunda clase se podrá reutilizar todo el código desarrollado para la clase Ventana (que como ventaja adicional ya estará libre de errores), y sólo será necesario codificar los dos nuevos métodos.

Mecanismo de respuesta a mensaje en una jerarquía de clases

Debido a la relación de herencia, el mecanismo de respuesta a un mensaje es del siguiente modo:

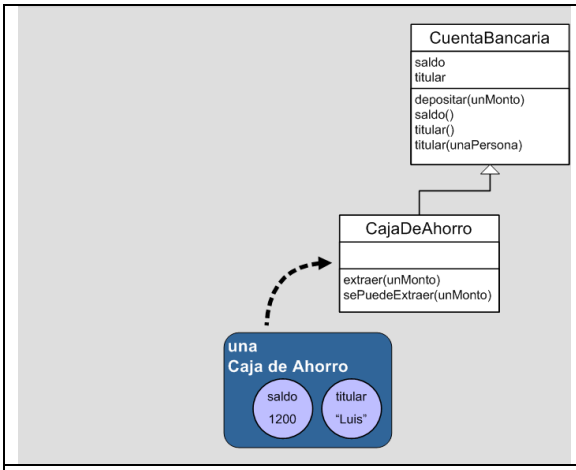
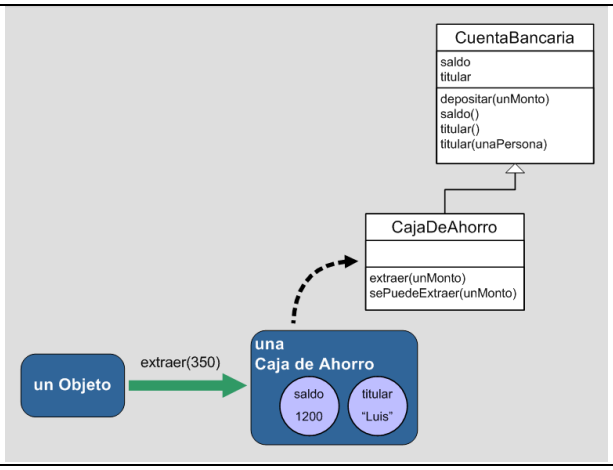
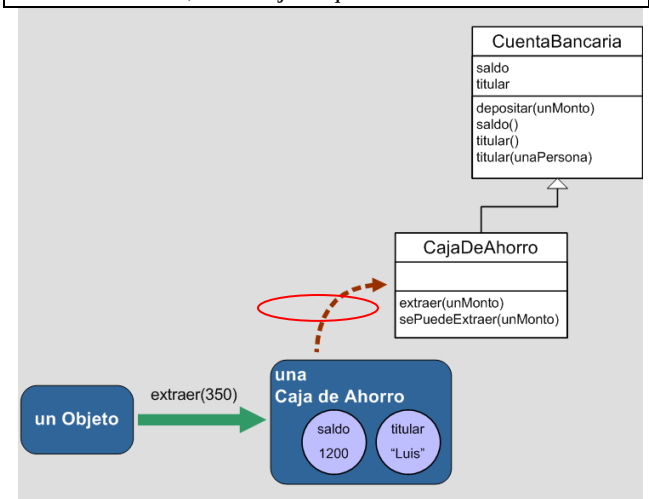
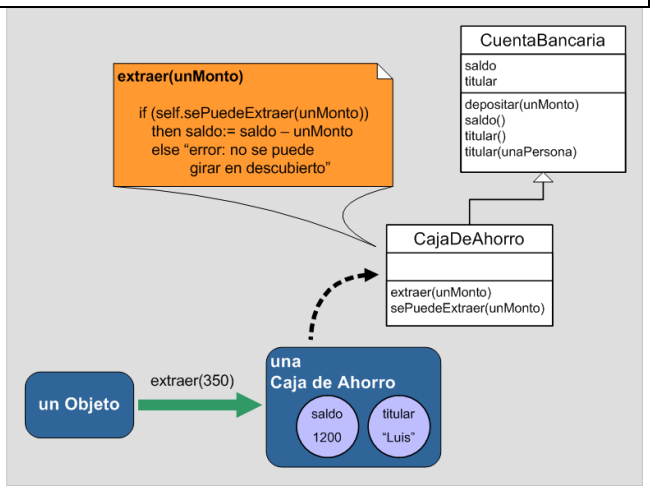
1. El objeto recibe un determinado mensaje.
2. Se busca el mensaje en la clase del objeto receptor.
3. Si el mensaje está definido en esa clase se ejecuta.
4. Si el mensaje no está definido en esa clase lo buscará en la superclase. Si tampoco lo encuentra allí, lo buscará en la siguiente superclase, y así sucesivamente hasta encontrarlo.
5. Si llega a la raíz de la jerarquía (Object en el caso de java) y nunca lo encuentra, se produce una excepción, pues el objeto no sabe responder a ese mensaje.

Este proceso se denomina method lookup. Esta frase **method look up** significa "búsqueda del método", y consiste básicamente en buscar en la cadena de superclases, partiendo de la clase del objeto receptor, un método para responder al envío de un mensaje.

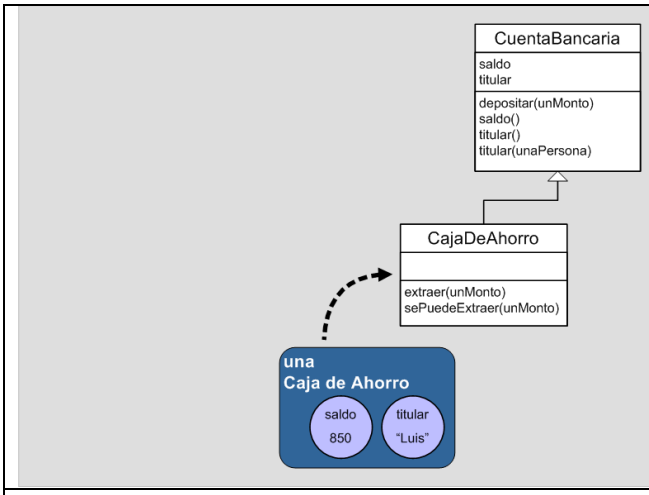
Si existieran dos métodos con el mismo nombre y distinto código, uno en la clase y otro en una superclase, se ejecutaría el de la clase, no el de la superclase (redefinición). Por lo general, siempre se puede acceder explícitamente al método de la clase superior mediante una sintaxis diferente, la cual dependerá del lenguaje de programación empleado. Por ejemplo, en Java, a través de la seudovariable *super*.

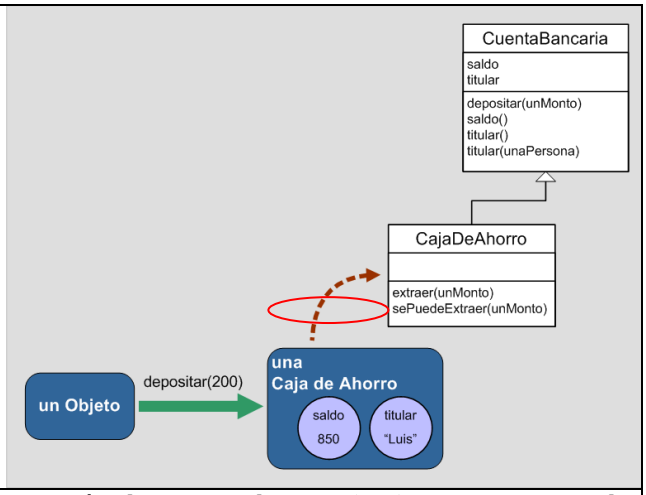
```
public String toString() {
        String s = super.toString();
        s = s + " Presupuesto: " + this.getPresupuesto();
        return s;
}
```

Ejemplo 1: Implementar el mensaje *extraer(unMonto)* en una caja de ahorro.

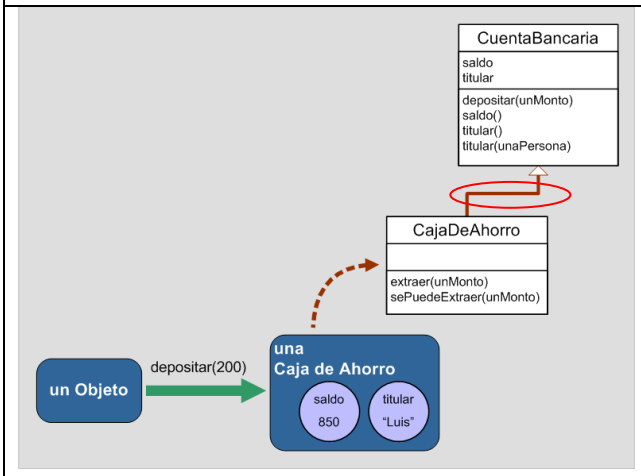
<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>Se dispone de un objeto unaCajadeAhorro, instancia de la clase CajaDeAhorro, que deriva de la clase CuentaBancaria, en una jerarquía de herencia de clases.</td><td>El objeto unObjeto envía el mensaje extraer(350) al objeto unaCajadeAhorro.</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>Al recibir el mensaje extraer(350), la instancia de CajaDeAhorro delega la búsqueda del método correspondiente a su clase.</td><td>La clase CajaDeAhorro define un método extraer(), por lo cual no se sigue buscando más arriba en la jerarquía y se procede a ejecutar este método.</td></tr>
</table>

Ejemplo 2: Implementar el mensaje *depositar(unMonto)* en una caja de ahorro.

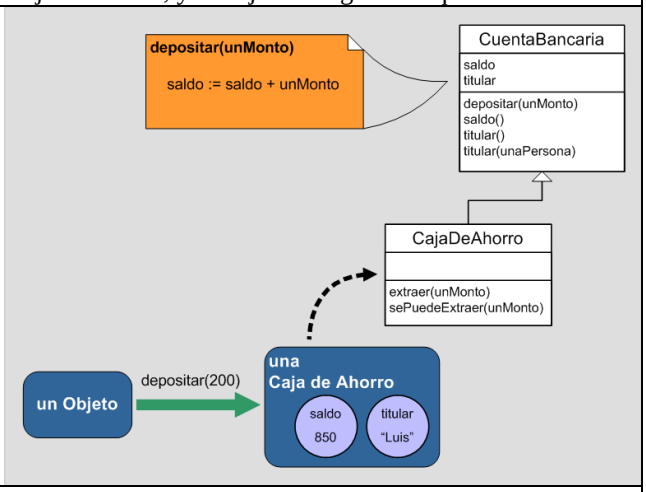




Se envía el mensaje *depositar*(200) a una instancia de CajaDeAhorro, y el objeto delega la búsqueda a su clase.



Pero en este caso, la clase CajaDeAhorro no tiene un método asociado al mensaje *depositar()*, por lo que vuelve a delegar la búsqueda, esta vez a su superclase, la clase CuentaBancaria.



La clase CuentaBancaria tiene definido el método *depositar*(), con lo cual la búsqueda termina y se ejecuta este método **en el contexto del objeto receptor** del mensaje, la instancia unaCajadeAhorro.

Herencia y ocultación de información

En una jerarquía de herencia, el diseñador puede controlar qué miembros de las superclases son visibles en las subclases. En el caso de Java y C++ los especificadores de acceso (private, protected, public) de los miembros de la superclase afectan también a la herencia:

- Private: ningún miembro privado de la superclase es visible en la subclase.
- Protected: los miembros protegidos de la superclase son visibles en la subclase, pero no visibles para el exterior.
- Public: los miembros públicos de la superclase siguen siendo públicos en la subclase.

El calificador *protected* permite la visibilidad de paquete, además de ser visible para las clases descendientes.

Como se ha visto anteriormente, el encapsulamiento es una característica de los lenguajes orientados a objetos muy útil para evitar la propagación de los cambios al resto del sistema. Al trabajar con herencia y definir atributos o métodos protegidos, surge un punto interesante: esta visibilidad permite violar el encapsulamiento. Una subclase puede acceder a las variables de instancia protegidas en la superclase, sin necesidad de definir ningún método con este propósito (get/set). En forma similar, una subclase puede acceder a todos los métodos definidos protegidos en su superclase.

A muchos programadores les parece natural la existencia de atributos y métodos protegidos. Sin embargo, se supone que una clase pública puede ser heredada por cualquier otra creada en el futuro, por lo que al declarar un atributo o método como protegido se lo está poniendo a disposición de cualquier clase que simplemente quiera extender la primera clase, y esto puede comprometer las modificaciones futuras.

Por eso, en general hay que respetar la norma de declarar los atributos privados y los métodos públicos. De este modo, incluso la subclase respeta el encapsulamiento de su superclase (podrá acceder a los atributos sólo mediante los métodos públicos definidos a tal efecto). El calificador *protected* debe dejarse solamente para aquellos casos particulares que lo ameriten, que son bastante poco frecuentes.

Ejemplo, si en la clase Empleado se define:

```
class Empleado {
    protected double sueldo;
    . . .
}
```

desde la clase Ejecutivo se puede acceder al dato miembro sueldo. Si se declara private no podrá acceder, y en ese caso la clase debe proveer los accessors (get/set).