Herencia

Salvador López Mendoza

Noviembre 2022

Introducción

Las clases no existen aisladas, en general se relacionan unas con otras.

Ejemplos:

- Una línea esta definida por dos puntos.
- Un triángulo está definido en términos de tres puntos.
- ¿Un auto?

Son ejemplos de una relación de uso entre un cliente y un servidor.

Evolución del software

En ocasiones es preciso modificar o adaptar clases existentes a nuevas necesidades.

Restricción: sin violar el principio de encapsulación.

Situación real:

- Se necesita agregar nuevos métodos a una clase ya existente.
- No se cuenta con el código fuente.

Objetivo: Mostrar la forma de crear nuevas clases por combinación, extensión y/o especialización de clases existentes.

La clase punto en tres dimensiones

Se ha definido la clase `Punto`, que nos permite representar puntos de un espacio cartesiano en dos dimensiones.

Ahora necesitamos manejar puntos en un espacio de tres dimensiones.

Vamos a suponer que no tenemos forma de acceder al código fuente.

¿Y si tuviéramos el código fuente?

Modificar el código para incorporar lo que se necesita.

- ¿Qué pasa con los programas que ya usaban la clase `Punto`?
- Si hay dos (o más) versiones nuevas de la clase `Punto`, ¿cuál es la que debo usar?

¿Y si programo todo de nuevo?

La clase Punto

Comportamiento de los objetos:

- Crear puntos.
- Desplazar un punto.
- Calcular la distancia con respecto a otro punto.
- Determinar si un punto está alineado con otros dos puntos.
- Determinar si un punto es igual a otro punto.
- Imprimir un punto en cierto formato.

La clase Punto3D

Comportamiento de los objetos:

- Crear puntos.
- Desplazar un punto.
- Calcular la distancia con respecto a otro punto.
- Determinar si un punto está alineado con otros dos puntos.
- Determinar si un punto es igual a otro punto.
- Imprimir un punto en cierto formato.

¿Cuál es la diferencia?

. . . los atributos, y los métodos.

Cuentas bancarias

Se nos contrata para resolver el siguiente problema:

Banco

Se requiere hacer un programa para el mantenimiento de cuentas bancarias teniendo cuentas de débito, cuentas con pago automático de servicios y cuentas de crédito.

Especificaciones adicionales:

- Con todas las cuentas se permite retirar dinero, depositar dinero y conocer el dinero disponible de la misma.
- Las cuentas con pago de servicio además permiten el pago automático del teléfono.
- Las cuentas de crédito tienen un límite de crédito para poder realizar compras y en cualquier momento se puede consultar el crédito disponible y el monto de la deuda.

Encontrar los objetos principales

- Cuenta bancaria.
- Cuenta de débito.
- Cuenta con pago de servicios.
- Cuenta de crédito.

Las cuentas de débito no tienen ningún comportamiento adicional al de cualquier cuenta bancaria así que pueden considerarse sinónimos.

Comportamiento de los objetos

Cuenta bancaria
Crear una cuenta
Retirar dinero
Depositar dinero
Consultar saldo

Cuenta de pago
Crear una cuenta
Retirar dinero
Depositar dinero
Consultar saldo
Pagar servicio

Cuenta de crédito
Crear una cuenta
Retirar dinero
Depositar dinero
Consultar saldo
Consultar crédito
Comprar a crédito

Escenarios de uso

Programa principal:

1. El programa da la bienvenida al usuario.
2. El programa presenta al usuario un menú con las diferentes opciones para usar las cuentas.
3. El usuario elige una opción.
4. El programa valida la opción.
5. De acuerdo a la opción elegida el programa solicita o muestra la información requerida. Si la opción es inválida regresar a (2).
6. El programa repite los pasos anteriores hasta que el usuario elige la opción de terminar.

Escenarios de uso (II)

Escenario: Retiro de dinero de una cuenta bancaria.

1. El programa solicita al usuario la cantidad de dinero que desea retirar.
2. El usuario indica la cantidad de dinero que desea retirar.
3. El programa valida la cantidad indicada, verificando que sea positiva y menor o igual que el disponible de la cuenta.
4. Si la cantidad es válida el programa *proporciona* al usuario la cantidad solicitada, en caso contrario envía un mensaje de error.

La clase Cuenta

Existe una clase Cuenta con métodos para realizar operaciones con cuentas bancarias, como son: crear cuentas, retirar dinero, depositar dinero y conocer el disponible de una cuenta.

```
public class Cuenta {
  private double disponible;
  private final long numCta;

  public Cuenta(double montoInicial) { ... }
  public void retirar(double monto) { ... }
  public void depositar(double monto) { ... }
  public double obtenerDisponible() { ... }
  public long obtenerNumCta() { ... }
}
```

La clase CuentaConServicios

Es muy parecida a la clase `Cuenta`.

Tiene un método para pagar los servicios.

Propuesta de solución: Ampliar la funcionalidad de la clase `Cuenta`, añadiendo los métodos necesarios para instrumentar el pago de servicios.

1. Modificar la clase `Cuenta` agregando los métodos requeridos.
2. Copiar el código de `Cuenta` en el programa para el pago de servicios y hacer el cambio necesario.
3. Usar el concepto de herencia de clases.

Definición

La **herencia** permite definir una nueva clase C_n a partir de una clase existente C, muy parecida a C_n, evitando la duplicidad de código.

En la clase C_n se definen sólo los atributos y los métodos que difieren de los existentes en la clase C y automáticamente se incluyen los métodos y atributos de C.

Los atributos y métodos de C_n son todos los de C más los especificados en C_n.

Los métodos en C_n pueden ser nuevos métodos o bien la redefinición o cancelación de métodos ya existentes en C.

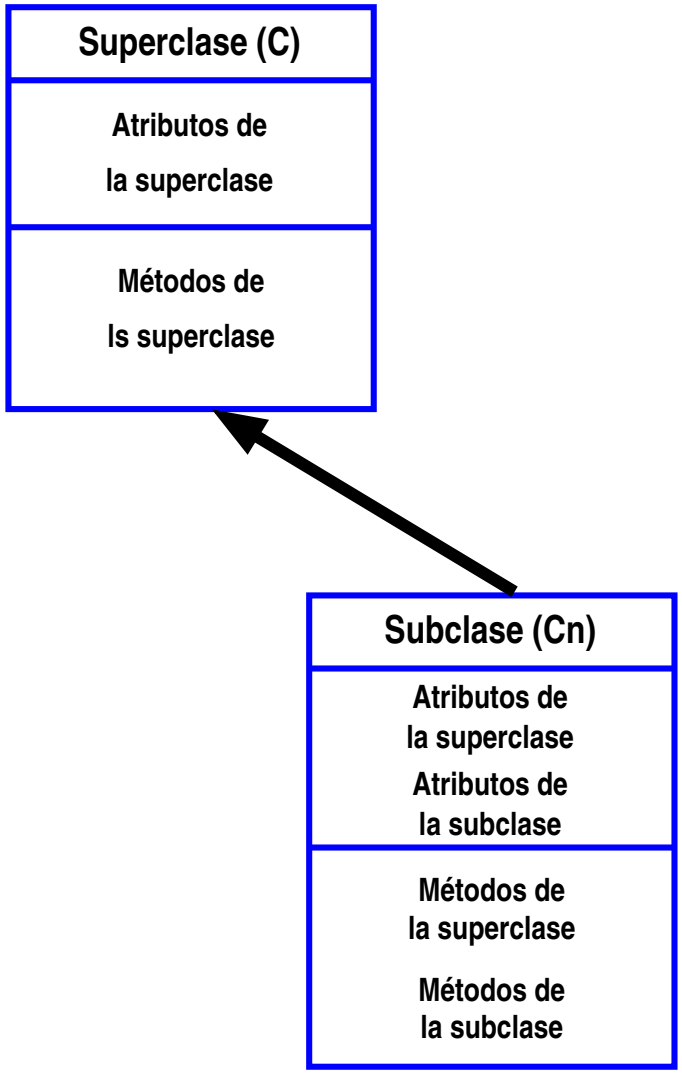
La herencia facilita desarrollar programas de manera incremental.

Definición (II)

La clase C_n se denomina **subclase** o clase derivada.

La clase C se conoce como **superclase**, clase base o clase padre.

Relación de herencia



Ejemplo: clase CuentaConServicios

Se agrega funcionalidad a la clase Cuenta: se puede pagar la renta de un teléfono determinado, con el beneficio de abonar $10.00 a la cuenta bancaria por hacer uso de este tipo de servicios.

```
public class CuentaConServicios extends Cuenta {
  /**
   * Método para pagar teléfono tomando dinero de la cuenta
   * con la que se llama a este método.
   * @param numTel - Número telefónico a donde se paga
   * @param monto - Cantidad que debe pagarse
   */
  public void pagarTelefono(String numTel, double monto) {
    retirar(monto);
    ...              // Código para pagar el teléfono
    disponible += 10.00;
  }
}
```

Atributos y métodos

Un objeto de la subclase tiene los atributos y métodos de la clase de la cual hereda, más los definidos en esta misma clase, como se muestra en la tabla:

Clase:	`Cuenta`	`CuentaConServicios`
Estructura:	disponible numCuenta	disponible numCuenta
Comportamiento:	retirar depositar obtenerDisponible	retirar depositar obtenerDisponible pagarTelefono

Control de acceso

Los objetos de una clase tienen acceso a los atributos y métodos definidos en su clase.

También tienen acceso a los atributos y métodos públicos de las clases que usa.

Si una clase se deriva de otra clase (herencia), ¿a qué tienen acceso los objetos de la clase hija?

Tienen acceso a los elementos públicos de su superclase.

NO tienen acceso a los elementos privados de su superclase.

Parar acceder a los atributos privados de la superclase se deben usar los métodos de esa superclase.

Elementos protegidos

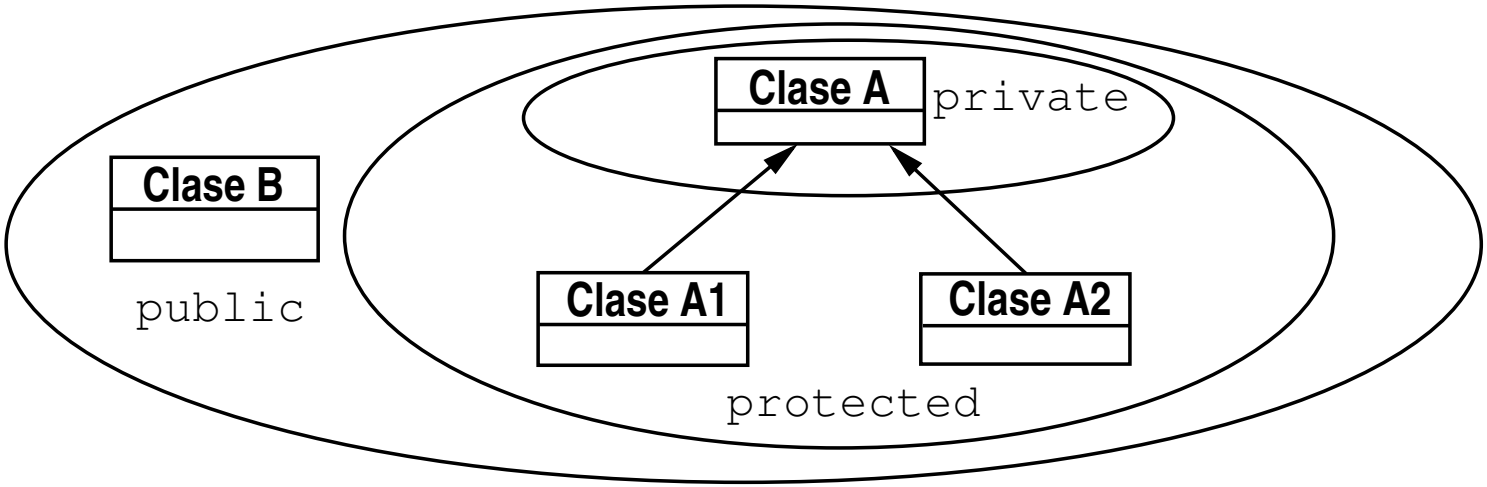
En algunos casos se desea definir atributos y métodos que son privados para las clases ajenas a la clase que se define, pero que están accesibles a los objetos de las subclases derivadas de esa clase.

En su declaración se preceden de la palabra `protected` en vez de la palabra `public`.

En la programación de un método se pueden usar los atributos públicos de cualquier clase, los atributos definidos en su clase y los atributos protegidos de sus superclases.

Para manipular el resto de los atributos se debe llamar a los métodos apropiados para trabajar con esos atributos.

Niveles de acceso



La clase CuentaConServicios

Para que funcionen correctamente los métodos de la clase `CuentaConServicios` la definición de la estructura de la clase `Cuenta` debería ser como sigue:

```
public class Cuenta {
  protected double disponible;
...
}
```

Si no se tiene acceso a la superclase para modificar la visibilidad de los atributos, o no se desea hacerlo, se debe acceder a los atributos usando los métodos creados para ese propósito.

En lugar de usar la instrucción `disponible += 10.00` usar la instrucción `depositar(10.00)` con lo cual se ejecuta el método `depositar` de la clase `Cuenta` que permite incrementar el disponible.

Constructores

Al crear un objeto se llama a un constructor de su clase para asegurar que se cree con un estado inicial válido.

Cuando se crea un objeto de una clase derivada implícitamente se crea un objeto de la clase base.

Al programar una subclase es necesario programar algún método constructor, el cual, en general, incluye la llamada a algún constructor de la superclase.

Se hace mediante la instrucción `super` con los argumentos adecuados. La instrucción `super` en una subclase, sirve para acceder a elementos de su superclase, en este caso al constructor.

Constructores (II)

La llamada al constructor de la superclase debe ser la primera instrucción del constructor de la subclase.

Se empieza por asignar un estado inicial a la parte heredada y luego se inicializa la parte propia de la clase.

Si no se incluye la llamada explícita al constructor de la superclase, Java realiza una llamada implícita al constructor por omisión de la superclase.

Constructor de la clase CuentaConServicios

```
/**
 * Crea una cuenta con disponible mínimo de $2500
 * @param montoInicial cantidad con la que se crea la cuenta
     .
 */
  public CuentaConServicios (double montoInicial) {
    super(montoInicial);
  }
```

En este caso el constructor de la subclase no añade nada al constructor de la superclase, pero puede hacerlo.

La creación de un objeto de una clase derivada puede causar una serie de llamadas a constructores en la jerarquía de clases.

Se ejecuta primero el constructor de la clase superior en la jerarquía y se va descendiendo hasta llegar a la clase que llamó originalmente al constructor.

La clase CuentaDeCredito

Estas cuentas permiten efectuar compras hasta un cierto límite de crédito.

Para implementar esa funcionalidad, las cuentas de crédito requieren, además de los atributos de una cuenta común, otro atributo para definir el importe de la deuda.

También se requieren métodos específicos para este tipo de cuentas: `comprar` y `obtenerValorDeuda`.

Una cuenta de crédito tiene los siguientes atributos y métodos ya mencionados.

Estructura para las cuentas de crédito

```
/**
 * Clase para trabajar cuentas de crédito
 * @author  Amparo López Gaona
 * @version Noviembre 2010
 */
public class CuentaDeCredito extends Cuenta {
  private double deuda;          // Importe de la deuda
```

De esta forma las cuentas de crédito tienen tres atributos: dos heredados (`numCta` y `disponible`) y otro definido en ella (`deuda`).

Constructor para cuentas de crédito

```
/**
 * Constructor de una cuenta de crédito
 * @param crédito - límite de crédito otorgado
 */
public CuentaDeCredito (double credito) {
  super (credito);
  deuda = 0;
}
```

El constructor de cuentas de crédito inicializa la parte correspondiente a `Cuenta`, mediante el uso de la instrucción `super`.

Luego inicializa las variables particulares de las cuentas de crédito.

Método obtenerValorDeuda

```java
/**
 * Método para conocer el importe de la deuda.
 * @return double - importe de la deuda.
 */
public double obtenerValorDeuda() {
  return deuda;
}
```

Método comprar

```java
/**
 * Método para  comprar con la cuenta de crédito
 * @param monto - importe de la compra
 */
public void comprar(double monto) {
  if (monto > 0.0 && monto < disponible ) {
    // disponible -= monto; // No recomendado
    retirar(monto) ; // Recomendado
    deuda += monto;
  } else {
    System.out.println("No se autoriza la compra " +
                       "por ese monto");
  }
}
```

El método retirar

¿Podemos usar el método `retirar` de la clase `Cuenta`?

En principio sí.

Pero, en las cuentas de crédito se cobra una comisión por retiro de dinero.

Es necesario **redefinir** este método para las cuentas de crédito.

```
/**
 * Método para retirar dinero de una cuenta de crédito
 * @param monto - importe del retiro
 */
public void retirar (double monto) {
  if (monto >0.0 && monto <= disponible ) {
    double comision = monto *0.02;
    super.retirar(monto+comision);
    deuda += (monto + comision);
  }
}
```

Especialización mediante herencia

En la redefinición de `retirar` se llama al método `retirar` de la clase `Cuenta` mediante la instrucción `super.retirar()`.

Es una situación en la que la nueva clase amplía la semántica del método equivalente definido en la superclase, pero aprovechando la definición existente en ésta.

En este caso, se dice que la clase `CuentaDeCredito` es más **especializada** que la clase `Cuenta`.

> *La especialización de clases se obtiene agregando atributos y métodos con los cuales se cambia la forma en que la subclase ofrece sus servicios con respecto a la superclase.*

Sobreescritura

El modificar la forma de trabajar de un método en una subclase sin modificar la firma se conoce como **sobreescritura de métodos**.

Es distinta a la sobrecarga de métodos.

Sobrecarga Métodos en la misma clase.
Tienen en común el nombre del método.
Los parámetros son distintos.

Sobreescritura Métodos en clases distintas, pero ligadas por la herencia.
La firma es exactamente la misma.

Sobreescritura (II)

Un método sobreescrito en una subclase oculta el método del ancestro.

Polimorfismo

La habilidad de decidir cuál método aplicar a un objeto en una jerarquía de herencia se denomina **polimorfismo**.

La palabra polimorfismo significa varias formas, esto significa que se permite usar el mismo nombre para referirse a distintos métodos.

La misma llamada puede, en diferentes momentos, referirse a diferentes métodos dependiendo de la clase del objeto que la hace.

El método depositar

Es necesario verificar que el pago sea positivo y en ese caso se reduce de la deuda y se aumenta el disponible. Si se paga más de lo que se debe ese excedente pasa a formar parte del disponible.

```
/**
 * Deposita una cantidad de dinero en la cuenta
 * @param monto cantidad que se desea depositar
 */
  public void depositar(double monto) {
    if (monto > 0) {
      deuda -= monto;
      if (deuda < 0) {
        super.depositar((-1)*deuda);
        deuda = 0;
      } else super.depositar(monto);
    }
  }
```

El método obtenerLimiteDeCredito

Para saber a cuánto asciende el límite de crédito disponible se puede definir el siguiente método.

```
/**
 * Método para conocer el límite de crédito de la tarjeta
 * @return double - límite de crédito
 */
public double obtenerLimiteCredito() {
  super.obtenerDisponible();
}
```

En este caso se podía usar directamente el método `obtenerDisponible`.

Jerarquía de clases

La relación de herencia entre clases forma una estructura jerárquica similar a un árbol.

Cada clase se representa como un nodo en el árbol.

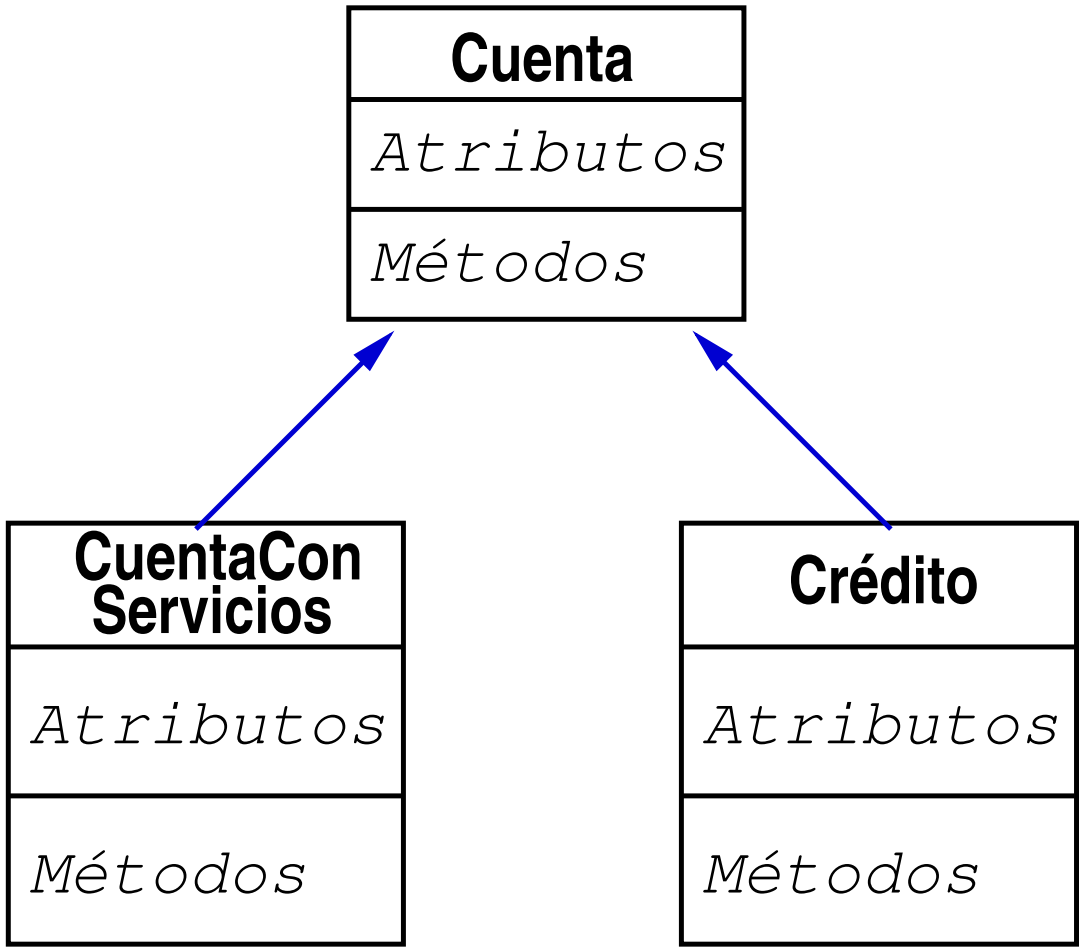
Una superclase tiene una relación jerárquica con sus subclases.

Una clase mientras más arriba está en la jerarquía, menor nivel de detalle requiere, es decir se trata de una clase más general.

Cuando una clase se crea con el mecanismo de herencia se convierte ya sea en una superclase que proporciona elementos a otras clases, o en una subclase que hereda elementos de otras.

La relación de herencia es transitiva, por tanto una superclase puede ser subclase de otra clase superior y una subclase puede ser superclase de otra clase.

Jerarquía de clases (II)



Clases sin herencia

En caso que no se desee permitir que se deriven clases de una clase se debe preceder su definición de la palabra reservada `final`.

Esta situación va en contra de la filosofía de programación orientada a objetos.

En caso de requerir nuevas funcionalidades se tiene que programar toda la clase.

Si se desea que un método de una clase no se oculte en una jerarquía de clases, su definición debe empezar con la palabra reservada `final`.

La palabra reservada `final` se utiliza para definir datos constantes, para definir clases que no se pueden extender y para definir métodos que no se pueden sobreescribir.

Ventajas de la herencia

- Se reduce el tiempo que toma construir nuevos programas, debido a que el software no tiene que re-inventarse (ni volver a escribirse y depurarse). Una aplicación se construye tomando clases ya escritas y extendiéndolas.
- Cuando se crean nuevas clases utilizando herencia, éstas son pequeñas debido a que sólo contienen las diferencias con respecto a sus superclases.
- Es posible extender clases sin necesidad de tener el código de ellas.
- Se facilita el trabajo de mantenimiento debido a que cualquier cambio en un atributo o método compartido sólo se hace en un lugar, evitando posibles inconsistencias.

Compatibilidad

Un objeto de una clase puede usarse en cualquier lugar en que se usaría un objeto de su superclase.

Se debe a que el objeto también pertenece a su superclase, posiblemente con más atributos y métodos.

Al definir una referencia a un objeto de cierta clase, se está en posibilidad de almacenar también una referencia a algún objeto de cualquiera de sus subclases.

No es posible almacenar una referencia a un objeto de su superclase.

Compatibilidad. Ejemplo

En una aplicación se han definido las siguientes referencias:

```
Cuenta cuenta;
CuentaConServicios ctaServicios = new CuentaConServicios();
CuentaDeCredito ctaCredito = new CuentaDeCredito();
```

¿Cuál de las siguientes asignaciones es válida?

```
cuenta = ctaCredito;
ctaCredito = cuenta;
```

Siempre es posible asignar un objeto especializado a uno general.

Compatibilidad. Conversión de clases

Para asignar un objeto general a uno especializado se requiere hacer una conversión explícita.

En el caso de las cuentas bancarias:

- Todas las cuentas de crédito son cuentas bancarias.
- Todas las cuentas con pago de servicios son cuentas bancarias.
- No todas las cuentas bancarias son cuentas de crédito.
  Una cuenta bancaria podría ser una cuenta con pago de servicios.

Si se insiste en hacer esta última asignación, se debe usar una conversión explícita de tipo para evitar el error de sintaxis.

```
ctaCredito = (CuentaDeCredito) cuenta;
```

Cuidado: En este caso sólo se pueden usar los métodos de la superclase.

Uso de las cuentas bancarias

Antes de asignar una referencia a un objeto es conveniente utilizar el operador `instanceof`. El resultado es verdadero si el objeto pertenece a la clase que se indica.

Ejemplo que usa diferentes clases de cuentas.

En vez de definir una referencia a objetos de `CuentaConSerevicios` se define como miembro de la clase `Cuenta`.

```
public class PruebaCuentas {
    ...
    Cuenta cuenta = null;
```

¿Qué se puede asignar a `cuenta`?

En el sistema bancario se permite crear cuentas, pero no se sabe de antemano de qué tipo es cada una.

Uso de las cuentas bancarias (II)

No todas las operaciones son aplicables a todas las cuentas.

Las cuentas de débito no tienen permitida la operación de pago del teléfono.

```
case 5:                    // Pago de teléfono
  if (cuenta instanceof CuentaConServicios) {
    CuentaConServicios cs = (CuentaConServicios)cuenta;
    capital = cantidadValida("Indica la cantidad a pagar " +
                             "a tu cuenta telefónica");
    cs.pagarTelefono(capital);
    cuenta = cs;
  } else
    System.out.println("Tu cuenta no tiene habilitado " +
                       "este servicio");
  break;
```

Generalización mediante herencia

La herencia se ha presentado como especialización de alguna clase existente para crear nuevas clases agregando los atributos o métodos necesarios para lograr la especialización.

Se desarrolla un árbol que va creciendo de la raíz a las hojas.

En ocasiones, durante el diseño de la solución a un problema se obtienen diversas clases relacionadas y no existen clases que extender.

Lo natural es que esas clases sean parte de una jerarquía de clases, pero no existe un ancestro común.

Se puede crear una jerarquía de herencia tomando las clases que tienen elementos en común, *factorizando* éstos en una superclase y creando cada subclase con las diferencias de ésta con respecto a la superclase.

En este caso se construye un árbol de las hojas a la raíz.

Problema

Un investigador necesita de un sistema que le permita llevar control de los libros, tesis y artículos que tiene para realizar su trabajo.

La información que requiere es:

Libros Nombre del autor, título del libro, editorial y año de publicación.

Tesis Nombre del autor, título de la tesis, director de la misma, año de graduación y grado obtenido.

Artículos Nombre del autor del mismo, título del artículo, nombre de la revista en que se publicó, año de publicación, volumen y número de la revista.

Resultados de la metodología de diseño

Los objetos principales son:

- Investigador.
- Libros.
- Tesis.
- Artículo.

Investigador es el usuario del sistema, no se programa como clase.

Estructura y comportamiento de los objetos encontrados

Libro	**Tesis**	**Artículo**
autor	autor	autor
título	título	título
tema	tema	tema
editor	director	nombre de revista
año	año de graduación	año de publicación
	grado	volumen
		número

Los métodos para asignar y consultar los atributos correspondientes.

Diseño

Al hacer un análisis de las clases se encuentra que tienen elementos en común (autor, título, tema y año).

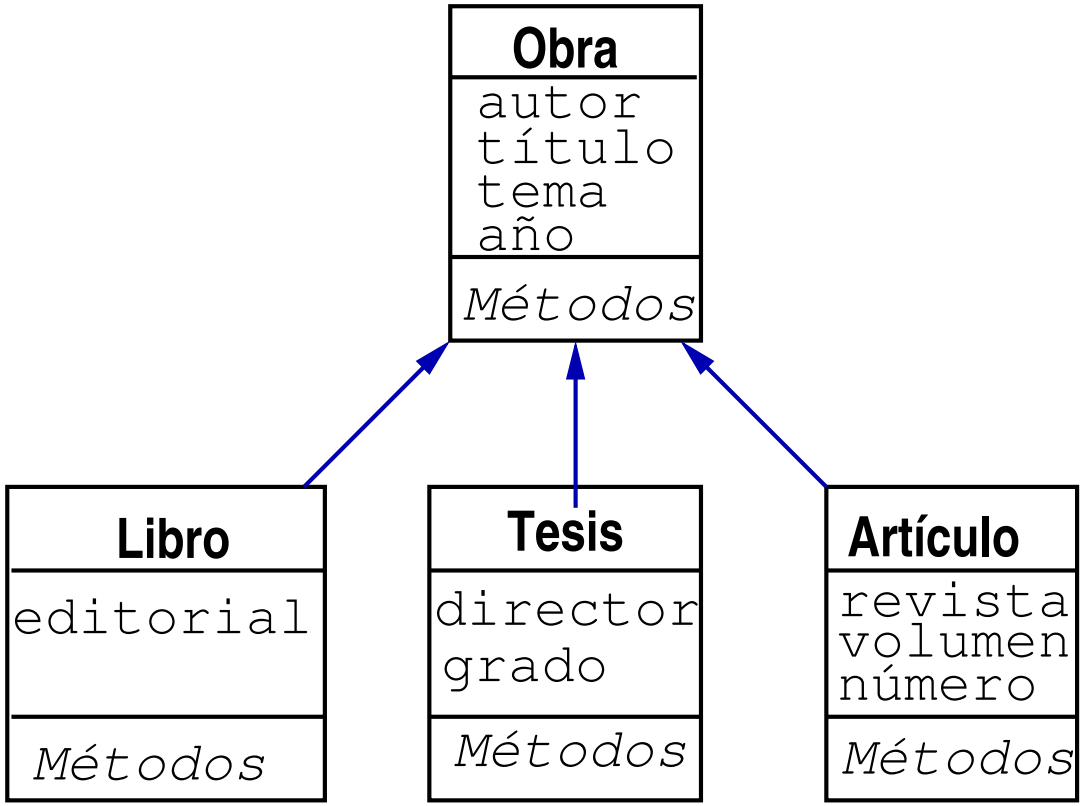
En vez de crear una clase para cada uno de los tipos de objetos encontrados se crea una jerarquía de clases en la que las clases encontradas son las hojas y se les crea una raíz con los elementos en común.

A esta clase se le llama `Obra`.

Ahora ya se pueden crear las clases que surgieron del diseño:

- `Libro`. Hay que agregar el atributo `editorial`
- `Tesis`. Se necesitan los atributos `director` y `grado`.
- `Artículo`. Se agregan `revista`, `volumen` y `número`.

Jerarquía de clases de obras impresas



La clase Obra

```java
/**
 * Clase raíz de la jerarquía de libros, artículos y tesis
 * Objetivo: ilustrar el concepto de herencia.
 * @author  Amparo López Gaona
 * @version Noviembre de 2010
 */
public class Obra {
  private String autor;
  private String titulo;
  private String tema;
  private int anio;
  public Obra(String a, String t, String tem, int an){
    autor = a;
    titulo = t;
    tema = tem;
    anio = an;
  }
```

La clase Obra (II)

```
// Aquí métodos para obtener y asignar valor a cada atributo

  public String toString() {
    return autor + "\t"+ titulo + "\t"+ tema + "\t" + anio;
  }
}
```

La clase Libro

```java
/**
 * Clase para almacenar y trabajar con información de libros
 * Objetivo: ilustrar el concepto de herencia.
 * @see Obra
 * @author  Amparo López Gaona
 * @version Noviembre de 2010
 */
public class Libro extends Obra {
  private String editorial;
  public Libro(String autor, String titulo, String tema,
               int anio, String ed){
    super(autor, titulo, tema, anio);
    editorial = ed;
  }
  public String toString() {
    return super.toString() + "\t" + editorial;
  }
}
```

La clase Tesis

```java
/**
 * Clase para almacenar y trabajar con información de tesis
 * Objetivo: ilustrar el concepto de herencia.
 * @see Obra
 * @author  Amparo López Gaona
 */
public class Tesis extends Obra {
  private String director;  private String grado;
  public Tesis(String autor, String titulo, String tema,
               int anio, String d, String g){
    super(autor, titulo, tema, anio);
    director = d;
    grado = g;
  }
  public String toString() {
    return super.toString() + "\t" + director + "\t"+grado;
  }
}
```

La clase Articulo

```
/**
 * Clase Articulo. Almacena y trabaja con artículos
 * @see Obra
 * @author  Amparo López Gaona
 * @version Noviembre de 2010
 */
public class Articulo extends Obra {
  private String revista;
  private int volumen;
  private int número;
  public Articulo(String autor, String titulo, String tema,
                  int anio, String r, int v, int n){
    super(autor, titulo, tema, anio);
    revista = r;
    volumen = v;
    número = n;
  }
```

La clase Articulo (II)

```
  // Aquí los métodos para obtener y asignar valor
  // a cada atributo.

  public String toString() {
    return super.toString() + "\t" + revista + "\t" +
                                volumen + "\t" + número ;
  }
}
```

La clase Object

En Java se tiene definida una clase llamada `Object` que es superclase de toda clase definida por los programadores.

Se encuentra definida en el paquete `java.lang`.

En la clase `Object` se define la estructura y comportamiento de todos los objetos.

Estructura y comportamiento de todos los objetos

Estructura:

No hay estructura en común.

Comportamiento:

Cosas que se pueden hacer con cualquier objeto:

- Comparar con otro objeto (`equals`).
- Convertir a `String` (`toString`).
- Operaciones para control de concurrencia.

Estructura y comportamiento de todos los objetos (II)

Método `equals` para determinar la igualdad de dos objetos.

Para poder decir que dos objetos son iguales si su estado es el mismo es necesario sobreescribir este método cuya firma es:

```
public boolean equals(Object o)
```

El parámetro es de tipo `Object`. Si fuera de otro tipo se estaría sobrecargando el método en vez de sobreescribirlo.

Si un método espera como parámetro un objeto de una clase particular puede recibir un objeto de cualquiera de sus subclases.

Como toda clase es hija de la clase `Object`, se puede llamar a este método con objetos de cualquier clase.

Comparación de objetos de la clase Punto

```java
/**
  * Determina si dos puntos son iguales
  * @param o - punto contra el cual se va a comparar
  * @return boolean - true si son iguales
  */
public boolean equals (Object o) {
  if (o == this) return true;
  if (o == null) return false;
  if (o instanceof Punto) {
    Punto p = (Punto) o;
    return obtenerX() == p.obtenerX() &&
           obtenerY() == p.obtenerY();
  }
  return false;
}
```

Conversión a cadenas para objetos de la clase Object

El método `toString` se usa para convertir a cadena la representación de un objeto.

Firma del método:

```
public String toString()
```

Este método no se llama explícitamente, se utiliza implícitamente.

Es otro ejemplo de sobreescritura.

Conversión a cadena para objetos de la clase Punto

```
/**
  * Convierte un objeto Punto a una cadena de caracteres
  * @return String - la cadena que representa al objeto
  */
public String toString () {

    return "(" + obtenerX() + ", " +
                 obtenerY() + ")" ;
}
```