Nombre: ..... Fecha: / 11 / 2009

Grupo: 1 ☐ 2 ☐ 3 ☐ 4 ☐

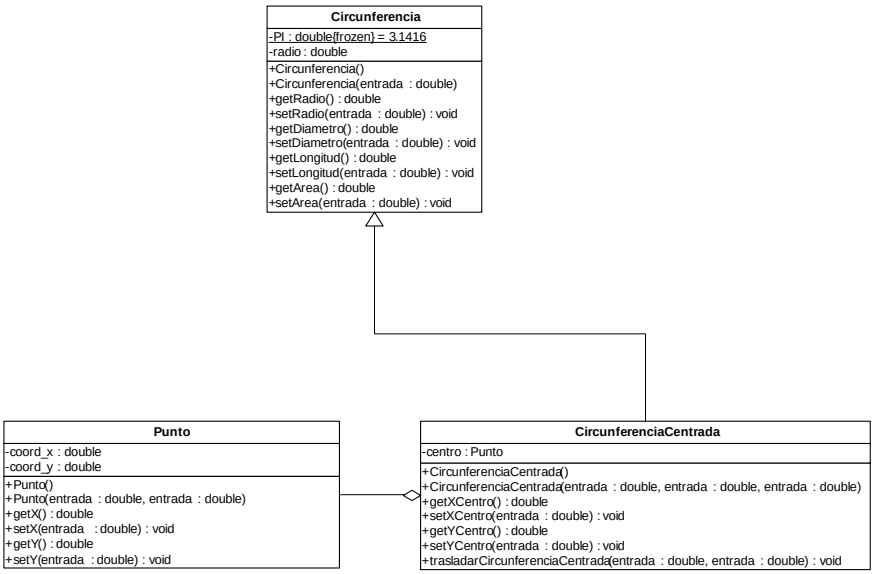
PRÁCTICA 7
POLIMORFISMO

En esta práctica introduciremos la noción de polimorfismo de métodos, la forma de conseguir comportamiento polimorfo en C++ y Java, y las consecuencias de su uso por medio de ejemplos. Para ello insistiremos primero en la importancia de diferenciar entre **declaración** y **construcción** de objetos. Después trataremos de ilustrar las consecuencias del uso de *enlazado estático* en C++, para finalmente mostrar los mecanismos que permiten conseguir *enlazado dinámico* en C++ y compararlo con los resultados antes obtenidos. También veremos cómo en Java el enlazado es siempre dinámico y el obtener polimorfismo de métodos no requiere ninguna modificación adicional en nuestros programas.

El primer requisito que debe cumplir un programa para que se pueda dar polimorfismo es que haya **relaciones de herencia** en el mismo, así como **métodos redefinidos** (o "hijos malos") que se comporten de forma distinta dependiendo de la clase en la que nos encontremos.

Parte 1. Diferencia entre declaración y construcción.

Recupera el siguiente caso de herencia que introdujimos en la Práctica 5 en Java y C++:



1. Crea un programa cliente, tanto en C++ como en Java, para el siguiente diagrama de clases (por medio de una función "main" en un fichero "principal_prac7_1") que realice las siguientes acciones:

a) Declara dos objetos de la clase **Circunferencia** de nombre "circ1" y "circ2" (Recuerda que en C++, la única forma de separar declaración de construcción es por medio de la declaración de punteros a objetos, que posteriormente sean construidos).

b) Declara otros dos objetos "circCentr1" y "circCentr2" de la clase **CircunferenciaCentrada**.

c) Construye "circ1" por medio del constructor "CircunferenciaCentrada ()" y "circ2" por medio del constructor "CircunferenciaCentrada (double, double, double)" (con valores iniciales 5.0, 6.5, 3.5).

d) Construye "circCentr1" por medio del constructor "CircunferenciaCentrada()" y "circCentr2" por medio del constructor "CircunferenciaCentrada (double, double, double)" con valores iniciales 3.0, 6.7, 8.9

2. Pasamos ahora a ver el comportamiento de los anteriores objetos en Java y C++:

a) Invoca al método "trasladarCircunferenciaCentrada(double, double): void" con valores 3.5 y 5.7 sobre los objetos "circ1", "circ2", "circCentr1" y "circCentr2". Observa el resultado y deja los errores encontrados entre comentarios.

b) Realiza las siguientes invocaciones en C++:

```
cout << "El tipo de circ1 " << typeid (circ1).name() << endl;
cout << "El tipo de *circ1 " << typeid (*circ1).name() << endl;

cout << "El tipo de circCentr1 " << typeid (circCentr1).name() << endl;
cout << "El tipo de *circ1Centr1 " << typeid (*circCentr1).name() << endl;
```

Relaciona los resultados con la noción de *enlazado estático*. ¿Qué tipo se ha asignado a los objetos y a los punteros a objetos? ¿El de su declaración o el de su construcción?

Realiza la siguiente operación en Java:

System.out.println ("El tipo de circ1 es " + circ1.getClass().getName());
System.out.println ("El tipo de circCentr1 es " + circCentr1.getClass().getName());

Comprueba en primer lugar el funcionamiento de los métodos "getClass(): Class" y "getName (): String" en la API de Java (http://java.sun.com/javase/6/docs/api/).

¿A qué clase pertenecen los objetos "circ1" y "circCentr1"? ¿A la clase de la que se han declarado o a la clase en que se han construido? Comprueba el resultado con el obtenido en C++ y relaciónalo con la noción de *enlazado dinámico*, opción propia de Java.

3. Veamos también las consecuencias de la diferencia entre declaración y definición a la hora de trabajar con estructuras genéricas. Comprueba lo siguiente en Java y C++:

a) Declara un "array" de la clase **Circunferencia** de nombre "array_circ" y de 4 componentes (recuerda que en Java se exige la inicialización de objetos de la clase **Array**).

b) Introduce los cuatro objetos anteriormente creados ("circ1" en "array_circ[0]", "circ2" en "array_circ[1]", "circCentr1" en "array_circ[2]" y "circCentr2" en "array_circ[3]") en "array_circ".

c) Compara el comportamiento ahora de las siguientes invocaciones:

circCentr1.getXCentro(); //utiliza el operador "->" para C++
array_circ[2].getXCentro();

circCentr2.trasladarCircunferenciaCentrada(2.3, 4.5); //utiliza el operador "->" para C++
array_circ[3].trasladarCircunferenciaCentrada(2.6, 7.8);

Deja entre comentarios los posibles errores obtenidos.

4. Realiza la siguiente comprobación. Vuelve a construir "circ1" llamando al constructor "Circunferencia()". Como puedes observar, el objeto "circ1" pertenece a la clase **Circunferencia**, y los métodos que podremos usar sobre el objeto "circ1" son los propios de la misma (y no aquellos que sean exclusivos de las clases derivadas, como **CircunferenciaCentrada**).

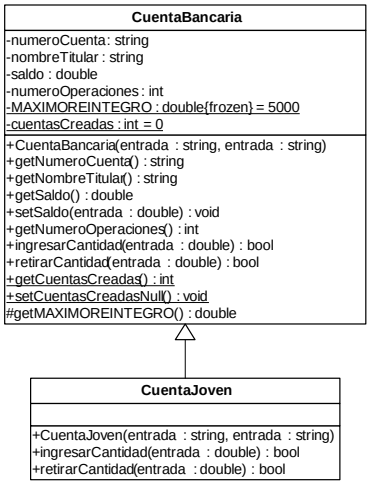
Parte 2. Polimorfismo.

Para que podamos hablar de polimorfismo es necesario tener relaciones de herencia y, además, **redefinición de métodos**. Los métodos redefinidos (Sección 2.6 de los apuntes) son aquéllos que cambian su comportamiento entre las clases bases y las derivadas. Lo que nos debe preocupar, por lo tanto, cuando invoquemos a un método redefinido desde un objeto de uno de sus subtipos es, ¿qué versión del mismo será invocada? ¿La de la clase base, la de la clase derivada, ...?

Tanto en Java como en C++, para que pueda haber polimorfismo debe haber **enlazado dinámico** de los métodos. Esto quiere decir que en tiempo de ejecución se debe comprobar qué definición de un método debe invocarse (la de la clase base, la de la clase derivada, ...).

Por defecto, C++ ofrece **enlazado estático**, mientras que Java ofrece **enlazado dinámico**. Comprobaremos esta afirmación en el siguiente ejercicio.

Recuperamos el tercer ejemplo de la Práctica 5 sobre las cuentas bancarias en C++ y Java.



5. Realiza las siguientes acciones en un cliente del anterior conjunto de clases en la forma de una función "main" (en un fichero de nombre "principal_pract7_2"):

a) Declara dos objetos de la clase **CuentaBancaria** de nombres "cuenta1" y "cuenta2" (recuerda que en C++ deberás declararlos como punteros).

b) Constrúyelos, el primero de ellos con el constructor "CuentaBancaria(string, string)" con valores "44445522229988776655" y "Antonio Montes", y el segundo con el constructor "CuentaJoven (string, string)" con valores "33336677778883334445" y "Miguel Delgado".

c) Recuerda que, tal y como definimos el comportamiento de la clase **CuentaJoven**, el método "ingresarCantidad (double): bool" sólo permitía ingresar cantidades hasta el 25% de "MAXIMOREINTEGRO". Comprueba si lo anterior es cierto en Java y en C++, haciendo un ingreso en "cuenta2" de 6000 euros.

¿Qué ha pasado en Java?

¿Qué ha pasado en C++? Relaciona tu respuesta con el hecho de que hayas declarado "cuenta2" como **CuentaBancaria**.

Comprueba en C++ el resultado de aplicar la función "typeid()" sobre "cuenta2" y sobre "*cuenta2". Comprueba el resultado de aplicar en Java el método "getClass(): Class" sobre "cuenta2".

d) Realiza ahora, en C++, un reintegro en "cuenta2" de 4000 euros. Recuerda que "retirarCantidad(double): bool", según se ha definido en **CuentaJoven**, no debía permitir reintegros superiores al 25% de "MAXIMOREINTEGRO" ni al saldo de la cuenta ¿Qué ha sucedido?

e) Realiza la siguiente modificación en C++. Cambia la declaración de "cuenta2" y declárala como **CuentaJoven**. Reconstruye tu proyecto para que se vuelva a alojar memoria. Ejecútalo.

¿Qué resultados has obtenido ahora?

f) Restablece la declaración de "cuenta2" a **CuentaBancaria**.

Los resultados anteriores en C++ son consecuencia del **enlazado estático**. Cuando hemos declarado "cuenta2" como **CuentaBancaria**, se le ha asignado ese tipo, y, a la vez, todos los métodos de esa clase. Cuando lo hemos construido como **CuentaJoven**, el tipado estático no ha sido capaz de reconocer que algún método (tanto "retirarCantidad(double): bool" como "ingresarCantidad(double): bool") habían sido **redefinidos** y por tanto se debía "buscar" su definición en otra clase (**CuentaJoven**).

El comportamiento de un objeto no debería depender de cómo éste ha sido declarado (de la clase base o de la clase derivada). Lo natural es que el comportamiento de un objeto dependa de cómo este ha sido construido. Por lo tanto, lo normal es que en nuestros programas (en C++) siempre busquemos conseguir **enlazado dinámico**.

La forma de conseguir enlazado dinámico en C++ es por medio de utilizar el modificador "**virtual**" en la declaración de los métodos (es decir, en el archivo de cabeceras).

6. En el archivo "CuentaBancaria.h" añade el modificador "virtual" a los métodos que van a ser redefinidos, es decir, "retirarCantidad(double): bool" e "ingresarCantidad(double): bool". Reconstruye el proyecto (para que los cambios en los ficheros de cabeceras surtan efecto) y vuelve a ejecutarlo (asegúrate de que "cuenta2" está declarado como **CuentaBancaria** para que observes que realmente hemos obtenido polimorfismo).

¿Qué resultado has obtenido ahora? ¿Qué métodos "retirarCantidad(double): bool" e "ingresarCantidad(double): bool" han sido ahora utilizados?

Comprueba en C++ el resultado de aplicar la función "typeid()" sobre "cuenta2" y sobre "*cuenta2". Comprueba el resultado de aplicar en Java el método "getClass(): Class" sobre "cuenta2". ¿Obtienes ahora el resultado esperado?

Veamos ahora cómo el uso del polimorfismo se puede ampliar tanto a estructuras genéricas (que introdujimos en la Sección 2.5.2) como a funciones auxiliares.

7. Define, tanto en Java como en C++, un "array" con nombre "array_cuentas" de objetos de la clase **CuentaBancaria** de dos componentes (recuerda que debes inicializarlo en Java y que en C++ debes declarar y definir un constructor sin parámetros "CuentaBancaria()").

En "array_cuentas[0]" introduce el objeto "cuenta1", y en "array_cuentas[1]" introduce el objeto "cuenta2". Ejecuta el siguiente bucle:

```
for (int i = 0; i < 2; i++){
        "Mostrar por pantalla" array_cuentas[i].ingresarCantidad (5500);
        "Mostrar por pantalla" array_cuentas[i].getSaldo ();
}
```

¿Qué ha sucedido? ¿Cómo se ha comportado "array_cuentas[1]" en C++? ¿Como una **CuentaBancaria** o como una **CuentaJoven**? ¿Cómo debería haberse comportado?

8. Define un "array" de punteros a objetos de nombre "array_punteros_cuentas" de la clase **CuentaBancaria** en C++. Repite el ejercicio 7. Observa el resultado obtenido.

Como conclusión a los ejercicios 7 y 8 podemos señalar que, en C++, para obtener polimorfismo no sólo debemos declarar los métodos como "virtual" sino que también debemos utilizar punteros (o referencias) para gestionar los objetos sobre los que queremos obtener polimorfismo.

Pasemos ahora a ver lo mismo con **funciones auxiliares**.

9. Declara una función auxiliar (junto a tu programa principal, no en la clase **CuentaBancaria**) "ingresarCantidadAux(CuentaBancaria, double): void" que tome como parámetros un objeto de la clase **CuentaBancaria** (o de cualquiera de sus subtipos) y un "double".

El comportamiento del mismo será el siguiente. Si es posible realizar el ingreso especificado, lo realizará y, además, mostrará por pantalla un mensaje advirtiendo del nuevo saldo de la cuenta. Si no es posible, mostrará un mensaje advirtiendo de tal extremo y también mostrará el saldo que tengamos en la cuenta.

Desde el cliente "main" invoca a la función "ingresarCantidadAux(CuentaBancaria, double): void" con las cuentas "cuenta1" y "cuenta2" y las cantidades "5000". ¿Qué ha pasado con el objeto "cuenta2" en C++? De nuevo hemos perdido el polimorfismo.

10. Corrijamos la situación anterior. Para obtener polimorfismo en funciones auxiliares debemos utilizar "virtual" en los métodos que van a ser redefinidos y utilizar **referencias** o **punteros** en el paso de parámetros (igual que para los arrays).

Cambia la declaración y definición de la función auxiliar a "ingresarCantidadAux(CuentaBancaria &, double): void" y "ingresarCantidadAux(CuentaBancaria *, double): void". Repite las invocaciones a la función y comprueba los resultados.

ENTREGAR los archivos *CircunferenciaCentrada.cpp*, *CircunferenciaCentrada.h*, *Punto.cpp*, *Punto.h*, *Circunferencia.cpp*, *Circunferencia.h*, *principal_prac7_1.cpp*, *CircunferenciaCentrada.java*, *Punto.java*, *Circunferencia.java*, *principal_prac7_1.java*, *CuentaBancaria.cpp*, *CuentaBancaria.h*, *CuentaJoven.cpp*, *CuentaJoven.h*, *principal_prac7_2.cpp*, *CuentaBancaria.java*, *CuentaJoven.java*, *principal_prac7_2.java*. Los archivos deben contener **todos los cambios** realizados durante los ejercicios de la práctica. La entrega es a través de Belenus en el repositorio de Julio Rubio. También deben contener unas cabeceras que permitan identificarte:

```
/* Nombre: ___
   Grupo: _____
   Nombre del fichero: ____
   Descripción: _______*/
```

Guarda una copia de los ficheros para las prácticas sucesivas