Parte II

Programación Orientada a Objetos

Capítulo 5

Introducción a la programación orientada a objetos [1, 2]

5.1. Programación no estructurada

Comúnmente, las personas empiezan a aprender a programar escribiendo programas pequeños y sencillos consistentes en un solo programa principal.

Aquí ”programa principal”se refiere a una secuencia de comandos o instrucciones que modifican datos que son a su vez globales en el transcurso de todo el programa.

5.2. Programación procedural

Con la programación procedural se pueden combinar las secuencias de instrucciones repetitivas en un solo lugar.

Una llamada de procedimiento se utiliza para invocar al procedimiento.

Después de que la secuencia es procesada, el flujo de control procede exactamente después de la posición donde la llamada fue hecha.

Al introducir parámetros, así como procedimientos de procedimientos (subprocedimientos) los programas ahora pueden ser escritos en forma más estructurada y con menos errores.

Por ejemplo, si un procedimiento ya es correcto, cada vez que es usado produce resultados correctos. Por consecuencia, en caso de errores, se puede reducir la búsqueda a aquellos lugares que todavía no han sido revisados.

De este modo, un programa puede ser visto como una secuencia de llamadas a procedimientos. El programa principal es responsable de pasar los datos a las llamadas individuales, los datos son procesados por los procedimientos y, una vez que el programa ha terminado, los datos resultantes son presentados.

Así, el flujo de datos puede ser ilustrado como una gráfica jerárquica, un árbol, como se muestra en la figura para un programa sin sub-procedimientos.

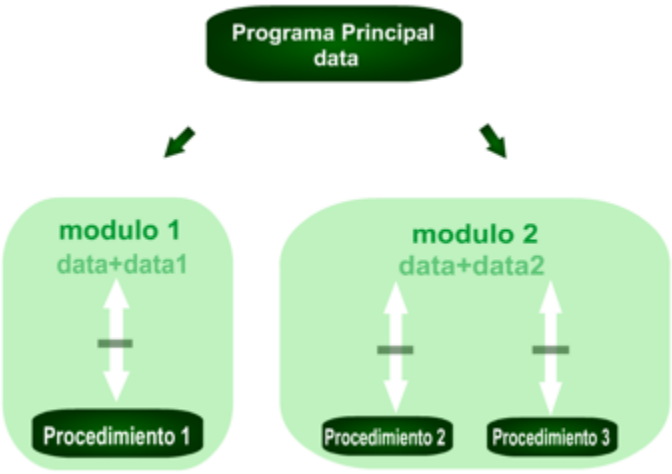
5.3. Programación modular

En la programación modular, los procedimientos con una funcionalidad común son agrupados en módulos separados.

Un programa por consiguiente, ya no consiste solamente de una sección. Ahora está dividido en varias secciones más pequeñas que interactúan a través de llamadas a procedimientos y que integran el programa en su totalidad.

Cada módulo puede contener sus propios datos. Esto permite que cada módulo maneje un estado interno que es modificado por las llamadas a procedimientos de ese módulo.

Sin embargo, solamente hay un estado por módulo y cada módulo existe cuando más una vez en todo el programa.



5.4. Datos y Operaciones separados

La separación de datos y operaciones conduce usualmente a una estructura basada en las operaciones en lugar de en los datos: **los Módulos agrupan las operaciones comunes en forma conjunta.**

Al programar entonces se usan estas operaciones proveyéndoles explícitamente los datos sobre los cuáles deben operar.

La estructura de módulo resultante está por lo tanto orientada a las operaciones más que sobre los datos. Se podría decir que las operaciones definidas especifican los datos que serán usados.

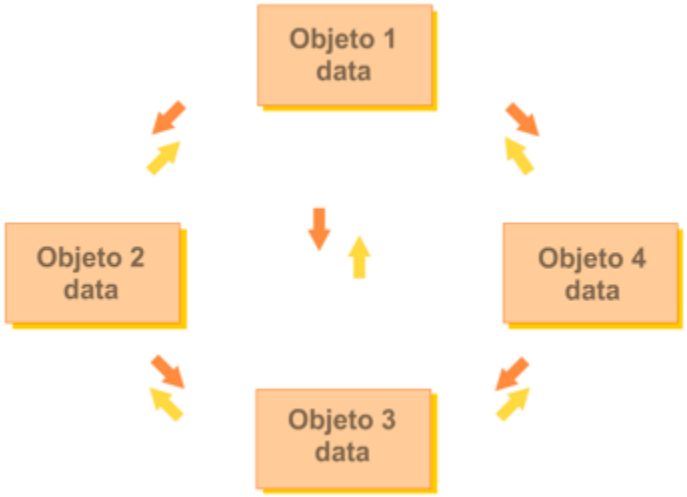
En la programación orientada a objetos, la estructura se organiza por los datos. Se escogen las representaciones de datos que mejor se ajusten a tus requerimientos. Por consecuencia, los programas se estructuran por los datos más que por las operaciones.

Los datos especifican las operaciones válidas. Ahora, los módulos agrupan representaciones de datos en forma conjunta.

5.5. Programación orientada a objetos

La programación orientada a objetos resuelve algunos de los problemas que se acaban de mencionar. De alguna forma se podría decir que obliga a prestar atención a los datos.

En contraste con las otras técnicas, ahora tenemos una telaraña de objetos interactuantes, cada uno de los cuáles manteniendo su propio estado.



Por ejemplo, en la programación orientada a objetos deberíamos tener tantos objetos de pila como sea necesario. En lugar de llamar un procedimiento al que le debemos proveer el manejador de la pila correcto, mandaríamos un mensaje directamente al objeto pila en cuestión.

En términos generales, cada objeto implementa su propio módulo, permitiendo por ejemplo que coexistan muchas pilas. Cada objeto es responsable de inicializarse y destruirse en forma correcta.

¿No es ésta solamente una manera más elegante de técnica de programación modular?

Podría ser, si esto fuera todo acerca de la orientación a objetos. De hecho se puede tratar de programar de esta forma sin POO. Pero eso no es todo lo que es la POO.

5.6. Tipos de Datos Abstractos

Algunos autores describen la programación orientada a objetos como programación de tipos de datos abstractos y sus relaciones. Los tipos de datos abstractos son como un concepto básico de orientación a objetos.

5.6.1. Los problemas

La primera cosa con la que uno se enfrenta cuando se escriben programas es el problema.

Típicamente, uno se enfrenta a problemas ”de la vida real” y nos queremos facilitar la existencia por medio de un programa para manejar dichos problemas.

Sin embargo, los problemas de la vida real son nebulosos y la primera cosa que se tiene que hacer es tratar de entender el problema para separar los detalles esenciales de los no esenciales: tratando de obtener tu propia perspectiva abstracta, o modelo, del problema. Este proceso de modelado se llama abstracción y se ilustra en la Figura:

El modelo define una perspectiva abstracta del problema. Esto implica que el modelo se enfoca solamente en aspectos relacionados con éste y que uno trata de definir propiedades del mismo. Estas propiedades incluyen

- los datos que son afectados
- las operaciones que son identificadas

por el problema.

Para resumir, la abstracción es la estructuración de un problema nebuloso en entidades bien definidas por medio de la definición de sus datos y operaciones. Consecuentemente, estas entidades combinan datos y operaciones. No están desacoplados unos de otras.

5.6.2. Tipos de Datos Abstractos y Orientación a Objetos

Los TDAs permiten la creación de instancias con propiedades bien definidas y comportamiento bien definido. En orientación a objetos, nos referimos a los TDAs como clases. Por lo tanto, una clase define las propiedades de objetos en un ambiente orientado a objetos.

Los TDAs definen la funcionalidad al poner especial énfasis en los datos involucrados, su estructura, operaciones, así como en axiomas y precondiciones. Consecuentemente, la

programación orientada a objetos es ”programación con TDAs”: al combinar la funcionalidad de distintos TDAs para resolver un problema. Por lo tanto, instancias (objetos) de TDAs (clases) son creadas dinámicamente, usadas y destruidas.

5.7. Conceptos de básicos de objetos

La programación tradicional separa los datos de las funciones, mientras que la programación orientada a objetos define un conjunto de objetos donde se combina de forma modular los datos con las funciones.

Aspectos principales:

1. Objetos.
   - El objeto es la **entidad básica** del modelo orientado a objetos.
   - El objeto integra una **estructura de datos** (atributos) y un **comportamiento** (operaciones).
   - Se distinguen entre sí por medio de su propia identidad, aunque internamente los valores de sus atributos sean iguales.
2. Clasificación.
   - Las clases **describen** posibles objetos, con una estructura y comportamiento común.
   - Los objetos que contienen los mismos atributos y operaciones pertenecen a la misma clase.
   - La estructura de clases **integra** las **operaciones** con los

   atributos a los cuales se aplican.
3. Instanciación
   - El proceso de **crear** objetos que pertenecen a una clase se denomina instanciación. (El objeto es la instancia de una clase).
   - Pueden ser instanciados un número indefinido de objetos de cierta clase.
4. Generalización.
   - En una jerarquía de clases, se **comparten** atributos y operaciones entre clases basados en la generalización de clases.
   - La jerarquía de generalización se construye mediante la herencia.
   - Las clases más generales se conocen como superclases. (clase padre)
   - Las clases más especializadas se conocen como subclases. (clases hijas)

- La herencia puede ser simple o múltiple.

5. Abstracción.
   - La abstracción se concentra en lo primordial de una entidad y no en sus propiedades secundarias.
   - Además en lo que el objeto hace y no en cómo lo hace.
   - Se da énfasis a cuales son los objetos y no cómo son usados.
6. Encapsulación.
   - Encapsulación o encapsulamiento es la **separación** de las **propiedades externas** de un objeto de los **detalles de implementación** internos del objeto.
   - Al separar la interfaz del objeto de su implementación, se limita la complejidad al mostrarse sólo la información relevante.
   - Disminuye el impacto a cambios en la implementación, ya que los cambios a las propiedades internas del objeto no afectan su interacción externa.
7. Modularidad
   - El encapsulamiento de los objetos trae como consecuencia una gran modularidad.
   - Cada módulo se concentra en una sola clase de objetos.
   - Los módulos tienden a ser pequeños y concisos.
   - La modularidad facilita encontrar y corregir problemas.
   - La complejidad del sistema se reduce facilitando su mantenimiento.
8. Extensibilidad.
   - La extensibilidad permite hacer cambios en el sistema sin afectar lo que ya existe.
   - Nuevas clases pueden ser definidas sin tener que cambiar la interfaz del resto del sistema.
   - La definición de los objetos existentes puede ser extendida sin necesidad de cambios más allá del propio objeto.
9. Polimorfismo (de subtipos).
   - El polimorfismo es la característica de definir las **mismas operaciones** con **diferente comportamiento** en diferentes clases.
   - Se permite llamar una operación sin preocuparse de cuál implementación es requerida en que clase, siendo responsabilidad de la jerarquía de clases y no del programador.

10. Reusabilidad de código.

    - La orientación a objetos apoya el reuso de código en el sistema.
    - Los componentes orientados a objetos se pueden utilizar para estructurar bibliotecas resuables.
    - El reuso reduce el tamaño del sistema durante la creación y ejecución.
    - Al corresponder varios objetos a una misma clase, se guardan los atributos y operaciones una sola vez por clase, y no por cada objeto.
    - La herencia es uno de los factores más importantes contribuyendo al incremento en el reuso de código dentro de un proyecto.

Actividad: lectura de artículo científico
Wegner, Peter. "Classification in object-oriented systems." ACM Sigplan Notices 21.10 (1986): 173-182.

5.8. Lenguajes de programación orientada a objetos

Simula I (1967) fue originalmente diseñado para problemas de simulación y fue el primer lenguaje en el cual los datos y procedimientos estaban unificados en una sola entidad. Su sucesor **Simula** (1973) , derivó definiciones formales a los conceptos de objetos y clase.

Simula sirvió de base a una generación de lenguajes de programación orientados a objetos. Es el caso de **C++** (1985), **Eiffel** (1986) y **Beta**. (1987)

Ada (1983), se derivan de conceptos similares, e incorporan el concepto de jerarquía de herencia. **CLU** -*clusters*- (1986) también incorpora herencia.

Smalltalk es descendiente directo de **Simula**, generaliza el concepto de objeto como única entidad manipulada en los programas. Existen tres versiones principales: **Smalltalk-72**, introdujo el paso de mensajes para permitir la comunicación entre objetos. **Smalltalk-76** que introdujo herencia. **Smalltalk-80** se inspira en **Lisp**.

Lisp contribuyó de forma importante a la evolución de la programación orientada a objetos.

Flavors (1986) maneja herencia múltiple apoyada con facilidades para la combinación de métodos heredados.

CLOS (1989), es el estándar del sistema de objetos de **Common Lisp**.

Los programas de programación orientada a objetos pierden eficiencia ante los lenguajes imperativos, pues al ser interpretado estos en la arquitectura *von Neumann* resulta en un **excesivo** manejo dinámico de la memoria por la constante creación de objetos, así como una fuerte carga por la división en múltiples operaciones (métodos) y su ocupación. Sin embargo se gana mucho en **comprensión** de código y **modelado** de los problemas.

Cuadro 5.1. Características de LPOO

	Ada 95	Eiffel	Smalltalk	C++	Java
Paquetes	Sí	No	No	No	Sí
Herencia	Simple	Múltiple	Simple	Múltiple	Simple
Control de tipos	Fuerte	Fuerte	Sin tipos	Fuerte	Fuerte
Enlace	Dinámico	Dinámico	Dinámico	Dinámico	Dinámico
Concurrencia	Sí	No	No	No	Sí
Recolección de basura	No	Sí	Sí	No	Sí
Afirmaciones	No	Sí	No	No	Sí*
Persistencia	No	No	No	No	No
Generecidad	Sí	Sí	Sí	Sí	Sí*

*Afirmaciones y persistencia en Java fueron incluidos a partir de la versión 5 (1.5) del lenguaje.

5.8.1. Características de los algunos LPOO[1]

En el Cuadro 5.1 podemos ver algunas características de lenguajes de programación orientados a objetos.

[1]Lenguajes de Programación Orientada a Objetos

Capítulo 6

Abstracción de datos: Clases y objetos

6.1. Clases

Se mencionaba anteriormente que la base de la programación orientada a objetos es la abstracción de los datos o los TDAs. La abstracción de los datos se da realmente a través de las clases y objetos.

Concepto
Def. Clase. Se puede decir que una clase es la implementación real de un TDA, proporcionando entonces la estructura de datos necesaria y sus operaciones. Los datos son llamados **atributos** y las operaciones se conocen como **métodos**[2].

La unión de los atributos y los métodos dan forma al **comportamiento** (comportamiento común) de un grupo de objetos. La clase es entonces como la definición de un esquema dentro del cual encajan un conjunto de objetos.

El comportamiento debe ser descrito en términos de **responsabilidades** [7]. Resolviendo el problema bajo esos términos permite una mayor independencia entre los objetos, al elevar el nivel de abstracción.

En Programación Estructurada el programa opera **sobre** estructuras de datos. En contraste en Programación Orientada a Objetos, el programa solicita a las estructuras de datos que ejecuten un servicio.

Ejemplos de clases:

- automóvil,
- persona,
- libro,
- revista,

- reloj,
- silla,
- ...

6.2. Objetos e instancias

Una de las características más importantes de los lenguajes orientados a objetos es la **instanciación**. Esta es la capacidad que tienen los nuevos tipos de datos, para nuestro caso en particular las clases de ser **instanciadas** en cualquier momento.

El instanciar una clase produce un objeto o instancia de la clase requerida. Todos los objetos son **instancia** de una clase[7].

Concepto
Def. Objeto. Un objeto es una instancia de una clase. Puede ser identificado en forma única por su nombre y define un estado, el cuál es representado por los valores de sus atributos en un momento en particular [2].

El estado de un objeto cambia de acuerdo a los métodos que le son aplicados. Nos referimos a esta posible secuencia de cambios de estado como el comportamiento del objeto:

Concepto
Def. Comportamiento. El comportamiento de un objeto es definido por un conjunto de métodos que le pueden ser aplicados [2].

6.2.1. Instanciación

Los objetos pueden ser creados de la misma forma que una estructura de datos:

1. **Estáticamente**. En **tiempo de compilación** se le asigna un área de memoria.
2. **Dinámicamente**. Se le asigna un área de memoria en **tiempo de ejecución** y su existencia es temporal. Es necesario liberar espacio cuando el objeto ya no es útil; para esto puede ser que el lenguaje proporcione mecanismos de recolección de basura.

En Java, los objetos sólo existen de manera dinámica, además de que incluye un recolector de basura para no dejar como responsabilidad del usuario la eliminación de los objetos de la memoria.

6.3. Clases en C++

Una clase entonces, permite encapsular la información a través de atributos y métodos que la utilizan, ocultando la misma y la implementación del comportamiento de las clases.

La definición de una clase define nuevos TDAs y la definición en C++ consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre llaves y finalizando con ";". Notar la similitud de las clases con las estructuras de C.

El cuerpo de la clase contiene la declaración de los atributos de la clase (variables) y la declaración de los métodos (funciones). Tanto los atributos como los métodos pertenecen exclusivamente a la clase y sólo pueden ser usados a través de un objeto de esa clase.

Sintaxis

```
class <nombre_clase> {
        <cuerpo de la clase>
};
```

Ejemplo:

```
1 class Ejemplo1 {
2         int x;
3          float y;
4          void fun(int a, float b) {
5                 x=a;
6                 y=b;
7          }
8 };
```

6.4. Miembros de una clase en C++

Una clase está formada por un conjunto de miembros que pueden ser datos, funciones, clases anidadas, enumeraciones, tipos de dato, etc. (amigos)Por el momento nos vamos a centrar en los datos y las funciones (atributos y métodos).

Es importante señalar que un miembro no puede ser declarado más de una vez[1]. Tampoco es posible añadir miembros después de la declaración de la clase.

Ejemplo:

[1]Aunque existe el concepto de sobrecarga que se verá más adelante

```
class Ejemplo2{
        int i;
        int i;   //error
        int j;
        int func(int, int);
};
```

6.4.1. Atributos miembro

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

6.4.2. Métodos miembro

Los métodos al igual que los atributos, deber ser definidos en la clase, pero el cuerpo de la función puede ir dentro o fuera de la clase. Si un método se declara completo dentro de la clase, se considera como inline.

La declaración dentro de la clase no cambia con respecto a la declaración de una función, salvo que se hace dentro de la clase. Veamos un ejemplo parecido al inicial de esta sección, pero ahora con el cuerpo de un método fuera del cuerpo de la clase.

Ejemplo:

```
//código en ejemplo3.h
class Ejemplo3 {
        public:
        int x;
         float y;
         int funX(int a) {
                x=a;
                return x;
        }
        float funY(float);
};
```

Podemos ver que en la definición de la clase se incluye un método en línea y un prototipo de otro método.

Para definir un método miembro de una clase fuera de la misma, se debe escribir antes del nombre del método, el nombre de la clase con la que el método esta asociado. Para esto se ocupa el operador de resolución de alcance (o de ámbito) ::.

Continuación del ejemplo:

```
float Ejemplo3::funY(float b){
        y=b;
        return y;
}
```

Reiteramos que al declarar los métodos fuera de la clase no puede mencionarse la declaración de un método que no esté contemplado dentro de la clase. Si esto fuera válido, cualquier método podría ganar acceso a la clase con sólo declarar una función adicional.

Ejemplo:

```
1  //error en declaración de un método
2  class x{
3          public:
4          int a;
5          f();
6  };
7
8  int x::g() {        //error, el metodo debe ser f()
9          return a*=3.1234;
10 }
```

La declaración de una función miembro es considerada dentro del ámbito de su clase. Lo cual significa que puede usar nombres de miembros de la clase directamente sin usar el operador de acceso de miembro de la clase.

Recordar que por convención en la programación orientada a objetos las funciones son llamadas métodos y la invocación o llamada se conoce como mensaje.

6.4.3. Un vistazo al acceso a miembros

Otra de las ventajas de la POO es la posibilidad de encapsular datos, ocultándolos de otros objetos si es necesario. Para esto existen principalmente dos calificadores que definen a los datos como **públicos** o **privados**.

Miembros públicos

Se utiliza cuando queremos dar a usuarios de una clase (e.g., otras clases) el acceso a miembros de esa clase, los miembros deben ser declarados públicos.

Sintaxis

```
public:
        <definición de miembros>
```

Miembros privados

Si queremos ocultar ciertos miembros de una clase de los usuarios de la misma, debemos declarar a los miembros como privados. De esta forma nadie más que los miembros de la clase pueden usar a los miembros privados. Con excepción de las funciones amigas. Por omisión los miembros se consideran privados. En una estructura se consideran públicos por omisión.

Sintaxis

```
private:
        <definición de miembros>
```

Normalmente, los atributos de la clase deben ser privados; así como los métodos que no sean necesarios externamente o que puedan conducir a un estado inconsistente del objeto[2].

En el caso de los atributos, estos al ser privados deberían de contar con métodos de modificación y de consulta pudiendo incluir alguna validación.

Es una buena costumbre de programación accesar a los atributos solamente a través de las funciones de modificación, sobre todo si es necesario algún tipo de verificación sobre el valor del atributo.

Ejemplo:

```
1  //código en ejemplo3.h
2  class Fecha {
3             private:
4             int dia;
5             int mes;
6             int an;
7
8           public:
9             bool setDia(int);    //poner día
10            int getDia();         //devuelve día
11            bool setMes(int);
12            int getMes();
13            bool setAn(int);
14            int getAn();
15 };
```

[2]Un estado inconsistente sería ocasionado por una modificación indebida de los datos, por ejemplo una modificación sin validación.

6.5. Objetos de clase en C++

Ya se ha visto como definir una clase, declarando sus atributos y sus operaciones, mismas que pueden ir dentro de la definición de la clase (*inline*) o fuera. Ahora vamos a ver como es posible crear objetos o instancias de esa clase.

Hay que recordar que una de las características de los objetos es que cada uno guarda un estado particular de acuerdo al valor de sus atributos[3].

Lo más importante de los lenguajes orientados a objetos es precisamente el objeto, el cual es una identidad lógica que contiene datos y código que manipula esos datos. En C++, un objeto es una variable de un tipo definido por el usuario[8]. Un ejemplo completo:

```
1  #include <iostream>
2  using namespace std;
3
4  class Ejemplo3 {
5          public:
6                  int i;
7                  int j;
8  };
9
10 int main() {
11         Ejemplo3 e1;
12         Ejemplo3 e2;
13         e1.i=10;
14         e1.j=20;
15
16         e2.i=100;
17         e2.j=20;
18         cout<<e1.i<<endl;
19         cout<<e2.i<<endl;
20
21         return 0;
22 }
```

Listing 34. Ejemplo de clases en C++.

Otro ejemplo, una cola:

```
1  class Cola{
2          private:
3            int q[10];
```

[3] A diferencia de la programación modular, donde cada módulo tiene un solo estado.

```
4           int sloc, rloc;
5
6         public:
7           void ini() {  //funci¢n en l¡nea
8                 sloc=rloc=-1;
9           }
10          bool set(int);
11          int get();
12  };
13
14  #include <iostream>
15  #include "Cola.h"
16
17  using namespace std;
18
19  bool Cola::set(int val){
20          if(sloc>=10){
21                  cout<<"la cola esta llena";
22                  return false;
23          }
24          sloc++;
25          q[sloc]=val;
26          return true;
27  }
28  int Cola::get(){
29          if(rloc==sloc)
30                  cout<<"la cola esta vacia";
31          else {
32                  rloc++;
33                  return q[rloc];
34          }
35  }
36
37
38  //cola definida en un arreglo
39  #include <iostream>
40  #include "Cola.h"
41
42  using namespace std;
43
44  int main(){
45          Cola a,b, *pCola= new Cola;  // *pCola=NULL y después asignarle
```

```
46
47          a.ini();
48          b.ini();
49          pCola->ini();
50          a.set(1);
51          b.set(2);
52          pCola->set(3);
53          a.set(11);
54          b.set(22);
55          pCola->set(33);
56          cout<<a.get()<<endl;
57          cout<<a.get()<<endl;
58          cout<<b.get()<<endl;
59          cout<<b.get()<<endl;
60          cout<<pCola->get()<<endl;
61          cout<<pCola->get()<<endl;
62
63             delete pCola;
64             return 0;
65  }
```

Listing 35. Ejemplo 2 de clases en C++, una estructura de cola simple.

Nota
Tomar en cuenta las instrucciones siguientes para el precompilador en el manejo de múltiples archivos.

```
1  #ifndef CCOLA_H
2  #define CCOLA_H
3  <definición de la clase>
4  #endif
```

6.6. Clases en Java

La definición en **Java**, de manera similar a C++, consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre llaves.

Sintaxis

```
class <nombre_clase> {
        <cuerpo de la clase>
}
```

Ejemplo[4]:

```
1 public class Ejemplo1 {
2         int x;
3         float y;
4         void fun(int a, float b) {
5                 x=a;
6                 y=b;
7         }
8 }
```

6.7. Miembros de una clase en Java

Los miembros en Java son esencialmente los atributos y los métodos de la clase.
Ejemplo:

```
1 class Ejemplo2{
2         int i;
3         int i;   //error
4         int j;
5         int func(int, int){}
6 }
```

[4] Algunos ejemplos como este no son programas completos, sino simples ejemplos de clases. Podrán ser compilados pero no ejecutados directamente. Para que un programa corra debe contener o ser una clase derivada de *applet*, o tener un método *main*.

6.7.1. Atributos miembro

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

La sintaxis mínima es la siguiente:

Sintaxis

```
tipo nombreAtributo;
```

Los atributos pueden ser inicializados desde su lugar de declaración:

```
tipo nombreAtributo = valor;
```

o, en el caso de variables de objetos:

```
tipo nombreAtributo = new Clase();
```

6.7.2. Métodos miembro

Un método es una operación que pertenece a una clase. No es posible declarar métodos fuera de la clase. Además, en Java no existe el concepto de método prototipo como en C++.

Sin embargo, igual que en C++, la declaración de una función ó método miembro es considerada dentro del ámbito de su clase.

La sintaxis básica para declarar a un método:

Sintaxis

```
tipoRetorno nombreMétodo ( [<parámetros>] ) {
        <instrucciones>
}
```

Un aspecto importante a considerar es que el paso de parámetros en Java es realizado exclusivamente por valor. Datos básicos y objetos son pasados por valor. Pero los objetos no son pasados realmente, se pasan las referencias a los objetos (i.e., una copia de la referencia al objeto).

6.7.3. Un vistazo al acceso a miembros

Si bien en Java existen también los miembros públicos y privados, estos tienen una sintaxis diferente a C++. En Java se define el acceso a cada miembro de manera unitaria, al contrario de la definición de acceso por grupos de miembros de C++.

Miembros públicos

Sintaxis
`public <definición de miembro>`

Miembros privados

Sintaxis
`private <definición de miembro>`

Recordatorio
Es una buena costumbre de programación acceder a los atributos solamente a través de las funciones de modificación, sobre todo si es necesario algún tipo de verificación sobre el valor del atributo. Estos métodos de acceso y modificación comúnmente tienen el prefijo *get* y *set*, respectivamente.

Ejemplo:

```
1  class Fecha {
2          private int dia;
3          private int mes, an;
4
5          public boolean setDia(int d){}   //poner día
6          public int getDia()        {} //devuelve día
7          public boolean setMes(int m){}
8          public int getMes(){}
9          public boolean setAn(int a) {}
10         public int getAn() {}
11 }
```

6.8. Objetos de clase en Java

En Java todos los objetos son creados dinámicamente, por lo que se necesita reservar la memoria de estos en el momento en que se van a ocupar. El operador de Java está basado

también en el de C++ y es *new*[5].

6.8.1. Asignación de memoria al objeto

El operador *new* crea automáticamente un área de memoria del tamaño adecuado, y regresa la referencia del área de memoria. Esta referencia debe de recibirla un identificador de la misma clase de la que se haya reservado la memoria.

Sintaxis

```
<identificador> = new Clase();
```

o en el momento de declarar a la variable de objeto:

```
Clase <identificador> = new Clase();
```

El concepto de *new* va asociado de la noción de constructor, pero esta se verá más adelante, por el momento basta con adoptar esta sintaxis para poder completar ejemplos de instanciación.

Un ejemplo completo:

```
1  public class Ejemplo3 {
2          public int i, j;
3
4          public static void main(String argv[]) {
5                  Ejemplo3 e3= new Ejemplo3();
6                  Ejemplo3 e1= new Ejemplo3();
7
8                  e1.i=10;
9                  e1.j=20;
10                 e3.i=100;
11                 e3.j=20;
12                 System.out.println(e1.i);
13                 System.out.println(e3.i);
14         }
15  }
```

[5]La instrucción *new*, ya había sido usada para reservar memoria a un arreglo, ya que estos son considerados objetos.

Listing 36. Ejemplo de clase en Java.

Otro ejemplo, una estructura de cola:

```
class Cola{
        private int q[];
        private int sloc, rloc;

        public void ini() {
                sloc=rloc=-1;
        q=new int[10];
        }

    public boolean set(int val){
                if(sloc>=10){
                        System.out.println("la cola esta llena");
                        return false;
                }
                sloc++;
                q[sloc]=val;
                return true;
        }

        public int get(){
        if(rloc==sloc) {
            System.out.println("la cola esta vacia");
            return -1;
        }
                else {
                        rloc++;
                        return q[rloc];
                }
        }
}

public class PruebaCola {
        public static void main(String argv[]){

                Cola a= new Cola(); // new crea realmente el objeto
                Cola b= new Cola(); // reservando la memoria
                Cola pCola= new Cola();

```

```
39          //Inicializacion de los objetos
40                  a.ini();
41                  b.ini();
42                  pCola.ini();
43
44                  a.set(1);
45                  b.set(2);
46                  pCola.set(3);
47                  a.set(11);
48                  b.set(22);
49                  pCola.set(33);
50
51                  System.out.println(a.get());
52                  System.out.println(a.get());
53                  System.out.println(b.get());
54                  System.out.println(b.get());
55                  System.out.println(pCola.get());
56                  System.out.println(pCola.get());
57          }
58  }
```

Listing 37. Ejemplo de clase con una estructura de Cola simple en Java.

6.9. Clases en Python

Una clase en **Python** consiste de la palabra reservada *class*, seguida del nombre de la clase y finalmente el cuerpo de la clase. Recordar que la identación es usada en Python para definir bloques de código.

Sintaxis

```
class <nombre_clase> :
                <cuerpo de la clase>
```

Ejemplo[6]:

```
1 class Ejemplo01:
2         def fun(a,b):
3                 x=a
4                 y=b
```

6.10. Miembros de una clase en Python

6.10.1. Métodos miembro

Un método es una operación que pertenece a una clase. La sintaxis básica para declarar a un método:

Sintaxis

```
def nombreMétodo( <parámetros> ) :
        <instrucciones>
```

Un aspecto importante a considerar es que la lista de parámetros, al igual que el uso de variables, **no** requiere definir el tipo de dato, pues éste se determina en el momento de la llamada al método.

[6] Algunos ejemplos como este no son programas completos, sino simples ejemplos de clases. Python es un lenguaje interpretado. No existe un método principal que inicie la ejecución. El intérprete recibe una lista de instrucciones y éste comienza ejecutando de la línea inicial hasta la última línea.

6.10.2. Atributos miembro

Los atributos o variables de instancia en Python deben ser declarados dentro de un método de la clase mediante:

```
self.nombre= valor
```

Los atributos podrán ser accedidos usando la misma notación (*self.nombre*)
Ejemplo:

```
1 class InstTest:
2     def set_foo(self, n):
3         self.foo = n
4     def set_bar(self, n):
5         self.bar = n
```

Es importante señalar que los métodos deben recibir al objeto en el primer parámetro. Esto se hace nombrando al primer parámetro como *self*.

6.10.3. Un vistazo al acceso a miembros

No existen miembros privados en Python, todos los miembros son considerados públicos. Sin embargo, existe una convención que siguen los programadores en este lenguaje: un nombre precedido por un guión bajo _ . Entonces una variable "privada" deberá de ser nombrada, por ejemplo *_atributo*.
Ejemplo:

```
1 class A:
2
3   def _metodoPrivado():
4     # codigo
```

6.11. Objetos de clase en Python

En Python todos los números, arreglos, cadenas y demás entidades son objetos. Cada objeto tiene un valor, un identificador y un tipo. El identificador del objeto y el tipo pueden ser accedidos mediante las operaciones *id*() y *type*() respectivamente:

```
1 >>> n=10
2 >>> id(n)
3 3492368
4 >>> type(n)
5 <class 'int'>
6 >>> arr=[2,3,4]
```

```
>>> id(arr)
19201080
>>> type(arr)
<class 'list'>
```

6.11.1. Asignación de memoria al objeto

En Python, un objeto es instanciado mediante la ejecución objeto de la clase que se quiere instanciar, éste crea automáticamente un área de memoria del tamaño adecuado, y regresa la referencia del área de memoria. El objeto instanciado típicamente es asignado a una variable.

Sintaxis
`identificador = Clase()`

Un ejemplo completo:

```
class Ejemplo3:
    def unMetodo(self, x, y):
        self.i=x
        self.j=y
        print("El valor de i es " + str(self.i))
        print("El valor de j es " + str(self.j))
# script de ejecucion
obj1= Ejemplo3()
obj2= Ejemplo3()
obj1.unMetodo(10, 20)
obj2.unMetodo(100, 200)
```

Listing 38. Ejemplo de clases en Python.

Otro ejemplo, una estructura de cola:

```
class Cola:
    def ini(self):
        self.sloc= self.rloc=-1
        self.q=[]

    def setC(self, val):
        if self.sloc>1000:
            print("La cola esta llena")
            return False
```

```
10          self.sloc+=1
11          self.q.append(val)
12          return True
13
14      def getC(self):
15          if self.rloc == self.sloc:
16              print("La cola esta vacia")
17              return False
18          else:
19              self.rloc+=1
20              return self.q[self.rloc]
21
22  # script de ejecucion
23  a = Cola()
24  b = Cola()
25
26  a.ini()
27  b.ini()
28
29  a.setC(1)
30  b.setC(2)
31  a.setC(11)
32  b.setC(22)
33
34  print(a.getC())
35  print(a.getC())
36  print(b.getC())
37  print(b.getC())
38
39  a.getC()
```

Listing 39. Ejemplo.

6.12. Usando la palabra reservada this en C++, C#, D, Scala y Java

Cuando en algún punto dentro del código de algunos de los métodos se quiere hacer referencia al objeto ligado en ese momento con la ejecución del método, podemos hacerlo usando la palabra reservada *this*.

Una razón para usarlo es querer tener acceso a algún atributo posiblemente oculto por un parámetro del mismo nombre.

También puede ser usado para regresar el objeto a través del método, sin necesidad de realizar una copia en un objeto temporal.

La sintaxis es la misma en C++ y en Java, con la única diferencia del manejo del operador de indirección "*" si, por ejemplo, se quiere regresar una copia y no la referencia del objeto. **D**, **C#** y **Scala** también utilizan *this*.

Ejemplo en C++:

```
Fecha Fecha::getFecha(){
        return *this;
}
```

Ejemplo en Java:

```
class Fecha {
        private int dia;
        private int mes, an;
        ...
        public Fecha getFecha(){
                return this;
    }
        ...
}
```

6.13. Usando *self* en Python

En Python, el primer argumento de un método es llamado *self*. Esto no es más que una convención puesto que no se trata de una palabra reservada. Es utilizado para identificar al objeto que ejecuta al método y es necesario inclusive para el uso de sus atributos y métodos, como se ha visto en ejemplos anteriores.

Ejemplo:

```
1  class SelfEjemplo:
2      def ini(self):
3          self.x=0
4          self.y=0
5
6      def getSelfEjemplo(self):
7          return self
8
9
10 # script de ejecucion
11 obj1 = SelfEjemplo()
12 obj1.ini()
13
14 obj1.x=10
15 print(obj1.x)
16 obj1.y="YY"
17 print(obj1.y)
18
19 obj2=obj1.getSelfEjemplo()
20
21 print(obj2.x)
22
23 obj1.y=123
24 print(obj1.y) #despliega 123
25 print(obj2.y) #despliega 123
```

Listing 40. Ejemplo de uso de *self* en Python.

Capítulo 7

Polimorfismo AdHoc: Sobrecarga de operaciones

7.1. Introducción

Es posible tener el **mismo nombre** para una operación con la condición de que tenga parámetros diferentes. La diferencia debe de ser al menos en el tipo de datos. Al menos un parámetro debe ser diferente.

Concepto
Si se tienen dos o más operaciones con el mismo nombre y diferentes parámetros se dice que dichas operaciones están **sobrecargadas**.

El compilador sabe que operación ejecutar a través de la **firma** de la operación, que es una combinación del nombre de la operación y el número y tipo de los parámetros.

El tipo de regreso de la operación puede ser igual o diferente.

La sobrecarga[1] de operaciones sirve para hacer un código más legible y modular. La idea es utilizar el mismo nombre para operaciones relacionadas. Si no tienen nada que ver entonces es mejor utilizar un nombre distinto. A continuación ejemplos de sobrecarga en C++ y Java.

7.2. Ejemplos de sobrecarga en C++

```
1 class MiClase{
2         int x;
3
4         public:
```

[1]También conocida como homonimia.

```
5           void modifica() {
6                   x++;
7           }
8           void modifica(int y){
9                   x=y*y;
10          }
11  }
```

Ejemplo completo e C++:

```
1   //fuera de POO
2   #include <iostream>
3   using namespace std;
4
5   int cuadrado(int i){
6           return i*i;
7   }
8   double cuadrado(double d){
9           return d*d;
10  }
11
12  int main() {
13          cout<<"10 elevado al cuadrado:  "<<cuadrado(10)<<endl;
14          cout<<"10.5 elevado al cuadrado: "<<cuadrado(10.5)<<endl;
15          return 0;
16  }
```

Listing 41. Ejemplo de sobrecarga en C++.

7.3. Ejemplo de sobrecarga en Java

```
1   class MiClase{
2           int x;
3           public          void modifica() {
4                   x++;
5           }
6           public void modifica(int y){
7                   x=y*y;
8           }
9   }
```

7.4. En Python no hay sobrecarga

Como en Ruby, Python **no** soporta sobrecarga de operaciones. Si dos métodos son definidos con el mismo nombre, la última implementación definida es la que se espera.

```
1  class MiClase:
2      def modifica(self):
3          self.x+=1
4
5      def modifica(self, y):
6          self.x=y*y
7
8
9  #prueba
10 mc=MiClase()
11 mc.modifica(5)
12 print(mc.x)
13 #  mc.modifica()    -- Error pues se ha redefinido el metodo
14 mc.modifica(10)
15 print(mc.x)
```

Listing 42. Ejemplo redefiniendo un método en Python.

Capítulo 8

Constructores y destructores

8.1. Constructores y destructores en C++

Con el manejo de los tipos de datos primitivos, el compilador se encarga de reservar la memoria y de liberarla cuando estos datos salen de su ámbito.

En la programación orientada a objetos, se trata de proporcionar mecanismos similares, aunque con mayor funcionalidad. Cuando un objeto es creado es llamado un método conocido como **constructor**, y al salir se llama a otro conocido como **destructor**. Si no se proporcionan estos métodos se asume la acción más simple.

8.1.1. Constructor

Un **constructor** es un método con el mismo nombre de la clase. Este método no puede tener un tipo de dato y si puede permitir la homonimia o sobrecarga.

Ejemplo:

```
1  class Cola{
2          private:
3          int q[100];
4          int sloc, rloc;
5
6           public:
7          Cola( );         //constructor
8          void put(int);
9          int get( );
10 };
11
12 //implementación del constructor
13 Cola::Cola ( ) {
14         sloc=rloc=0;
15         cout<<"Cola inicializada \n";
16 }
```

Un constructor si puede ser llamado desde un método de la clase.

8.1.2. Constructor de Copia

Es útil agregar a todas las clases un constructor de copia que reciba como parámetro un objeto de la clase y copie sus datos al nuevo objeto.

C++ proporciona un constructor de copia por omisión, sin embargo es una **copia a nivel de miembro** y puede no realizar una copia exacta de lo que queremos. Por ejemplo en casos de apuntadores a memoria dinámica, se tendría una copia de la dirección y no de la información referenciada.

Sintaxis
<nombre clase>(const <nombre clase> &<objeto>);

Ejemplo:

```
1  //ejemplo de constructor de copia
2  #include <iostream>
3  #include <time.h>
4  #include <stdlib.h>
5
6  using namespace std;
7
8  class Arr{
9          private:
10         int a[10];
11         public:
12
13         Arr(int x=0) {
14                 for( int i=0; i<10; i++){
15                         if (x==0)
16                                 x=rand();
17                         a[i]=x;
18         }
19 }
20
21         Arr(const Arr &copia){   //constructor de copia
22                 for( int i=0; i<10; i++)
23                         a[i]=copia.a[i];
24         }
25
26         char set(int, int);
27         int get(int) const ;
28         int get(int);
29 };
30
31 char Arr::set(int pos, int val ){
32         if(pos>=0 && pos<10){
33                 a[pos]=val;
34                 return 1;
35         }
36         return 0;
37 }
```

```
38
39 int Arr::get(int pos) const {
40         if(pos>=0 && pos<10)
41                 return a[pos];
42         //   a[9]=0;  error en un metodo constante
43         return 0;
44 }
45
46 int Arr::get(int pos) {  //no es necesario sobrecargar
47         if(pos>=0 && pos<10)   // si el metodo no modifica
48                 return a[pos];
49         return 0;
50 }
51
52 int main(){
53         Arr a(5), b;
54         srand( time(NULL) );
55
56         a.set(0,1);
57         a.set(1,11);
58         cout<<a.get(0)<<endl;
59         cout<<a.get(1)<<endl;
60
61         b.set(0,2);
62         b.set(1,22);
63         cout<<b.get(0)<<endl;
64         cout<<b.get(1)<<endl;
65
66         Arr d(a);
67         cout<<d.get(0)<<endl;
68         cout<<d.get(1)<<endl;
69
70         return 0;
71 }
```

Listing 43. Ejemplo de constructor de copia en C++.

8.1.3. Destructor

La contraparte del constructor es el **destructor**. Este se ejecuta momentos antes de que el objeto sea destruido, ya sea porque salen de su ámbito o por medio de una instrucción *delete*. El uso más común para un destructor es liberar la memoria asignada dinámicamente,

aunque puede ser utilizado para otras operaciones de finalización, como cerrar archivos, una conexión a red, etc.

El destructor tiene al igual que el constructor el nombre de la clase pero con una tilde como prefijo (~).

El destructor tampoco regresa valores ni tiene parámetros.

Ejemplo:

```
class Cola{
         private:
        int q[100];
        int sloc, rloc;

         public:
        Cola( );        //constructor
        ~Cola();  //destructor
        void put(int);
        int get( );
};

Cola::~Cola( ){
        cout<<"cola destruida\n";
}
```

Ejemplo completo de Cola con constructor y destructor:

```
//cola definida en un arreglo
//incluye constructores y destructores de ejemplo
#include <iostream>
#include <string.h>
#include <stdio.h>

using namespace std;

class Cola{
        private:
        int q[10], sloc, rloc;
        char *nom;

        public:
        Cola(const char *cad=NULL) {  //funcion en linea
                if(cad){   //cadena!=NULL
                        nom=new char[strlen(cad)+1];
```

```
18                      strcpy(nom, cad);
19              }else
20                      nom=NULL;
21              sloc=rloc=-1;
22      }
23      ~Cola( ) {
24              if(nom){  //nom!=NULL
25                      cout<<"Cola : "<<nom<<" destruida\n";
26                      delete [] nom;
27              }
28      }
29
30      char set(int);
31      int get();
32  };
33
34  char Cola::set(int val){
35      if(sloc>=10){
36              cout<<"la cola esta llena";
37              return 0;
38      }
39      sloc++;
40      q[sloc]=val;
41      return 1;
42  }
43  int Cola::get(){
44      if(rloc==sloc)
45              cout<<"la cola esta vacia";
46      else {
47              rloc++;
48              return q[rloc];
49      }
50      return 0;
51  }
52
53  int main(){
54      Cola a("Cola a"),b("Cola b"),
55              *pCola= new Cola("Cola dinamica pCola");
56      a.set(1);
57      b.set(2);
58      pCola->set(3);
59      a.set(11);
```

```
60          b.set(22);
61          pCola->set(33);
62          cout<<a.get()<<endl;
63          cout<<a.get()<<endl;
64          cout<<b.get()<<endl;
65          cout<<b.get()<<endl;
66          cout<<pCola->get()<<endl;
67          cout<<pCola->get()<<endl;
68
69          delete pCola;
70  }
```

Listing 44. Ejemplo completo de Cola con constructor y destructor en C++.

8.2. Constructores y finalizadores en Java

En Java, cuando un objeto es creado es llamada un método conocido como **constructor**, y al salir se llama a otro conocido como **finalizador**[1]. Si no se proporcionan estos métodos se asume la acción más simple.

8.2.1. Constructor

Un constructor es un método con el mismo nombre de la clase. Este método no puede tener un tipo de dato de retorno y si puede permitir la homonimia o sobrecarga, y la modificación de acceso al mismo.

Ejemplo:

```
1  public class Cola{
2          private int q[];
3          private int sloc, rloc;
4          public void put(int){ ... }
5          public int get( ){ ... }
6          //        implementación del constructor
7          public Cola ( ) {
8                  sloc=rloc=0;
9                  q= new int[100];
10                 System.out.println("Cola inicializada ");
11         }
12 }
```

El constructor se ejecuta en el momento de asignarle la memoria a un objeto, y es la razón de usar los paréntesis junto al nombre de la clase al usar la instrucción *new*:

```
Fecha f = new Fecha(10,4,2007);
```

Si no se especifica un constructor, Java incluye uno predeterminado, que asigna memoria para el objeto e inicializa las variables de instancia a valores predeterminados. Este constructor se omite si se especifica uno o más por parte del programador.

8.2.2. Finalizador

La contraparte del constructor en Java es el método *finalize* o finalizador. Este se ejecuta momentos antes de que el objeto sea destruido por el recolector de basura. El uso más común para un finalizador es liberar los recursos utilizados por el objeto, como una conexión de red o cerrar algún archivo abierto.

[1] En C++ no existe el concepto de finalizador, sino el de destructor, porque su tarea primordial es liberar la memoria ocupada por el objeto, cosa que no es necesario realizar en Java.

No es muy común utilizar un método finalizador, más que para asegurar situaciones como las mencionadas antes. El método iría en términos generales como se muestra a continuación[2]:

Sintaxis

```
protected void finalize() {
        <instrucciones>
}
```

El finalizador puede ser llamado como un método normal, inclusive puede ser sobrecargado, pero un finalizador con parámetros no puede ser ejecutado automáticamente por la máquina virtual de Java. Se recomienda evitar el definir un finalizador con parámetros.

[2] No se ha mencionado el modificador *protected*. Este concepto se explicará una vez que se haya visto el manejo de herencia.

8.3. Inicializando y eliminando en Python

Clases en Python no tienen constructores ni destructores explícitos. Lo más parecido a un constructor es el método __init__ . El concepto es similar al de inicialización de objetos en Ruby.

Sintaxis

```
class NombreClase:
    def __init__(self, [<parametros>]):
        <código>
        ...

obj = NombreClase([<parámetros>])
```

Ejemplo:

```
class Cola:

        def __init__(self)
                self.sloc= self.rloc=-1
                self.q=[]
        end
        ...
end
```

Por otro lado, el método *__del__()* es lo más parecido que se tiene a un destructor, más parecido realmente al finalizador de java, ya que es llamado cuando todas las referencias a un objeto se han eliminado y éste es recogido por el recolector de basura.

```
#Ejemplo de inicializador y eliminador en Python
class Empleado:
    # Inicializador
    def __init__(self):
        print('Objeto empleado creado.')

    # Eliminando (Llamando al 'destructor')
    def __del__(self):
        print('Destructor llamado, Empleado borrado.')

```

```
11 obj = Empleado()
12 del obj
```

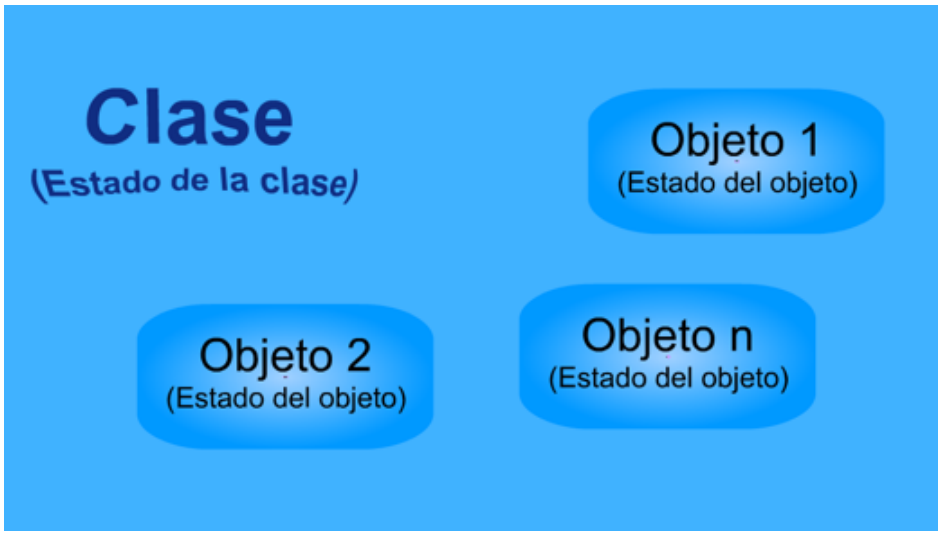
Listing 45. Ejemplo mostrando el uso de los métodos __init__() y __del__() en Python.

Capítulo 9

Miembros de clase o estáticos

Cada objeto tiene su propio estado, pero a veces es necesario tener valores por clase y no por objeto. En esos casos se requiere tener atributos **estáticos** que sean compartidos por todos los objetos de la clase.

Concepto
Existe solo una copia de un miembro estático y no forma parte de los objetos de la clase. Este tipo de miembro son también conocidos como miembros de clase.



9.1. Miembros estáticos en C++

Un miembro estático es accesible desde cualquier objeto de la clase o mediante el operador de resolución de alcance binario (::) y el nombre de la clase, dado que el miembro estático **existe** aunque no haya instancias de la clase.

Sin embargo, el acceso sigue restringido bajo las reglas de acceso a miembros:

- Si se quiere acceder a un miembro estático que es privado deberá hacerse mediante un método público.
- Si no existe ninguna instancia de la clase entonces deberá ser por medio de un método público y estático.

Además, un método estático solo puede tener acceso a miembros estáticos.

Ejemplo:

```
1  class Objeto{
2          private:
3          char nombre[10];
4          static int numObjetos;
5           public:
6          Objeto(char *cadena=NULL);
7          ~Objeto();
8  };
9
10 Objeto::Objeto(const char *cadena){
11         if(cadena!=NULL)
12                 strcpy(nombre, cadena);
13         else
14                 nombre=NULL;
15         numObjetos++;
16 }
17
18 Objeto::~Objeto(){
19         numObjetos--;
20 }
```

Los atributos estáticos deben de ser inicializados al igual que los atributos constantes, fuera de la declaración de la clase. Por ejemplo:

```
int Clase::atributo=0;                int const Clase::ATRCONST=50;
```

Ejemplo:

```
1  //prueba de miembros estáticos
2  #include <iostream>
3  #include <stdio.h>
4  #include <string.h>
5
6  using namespace std;
7
8  class Persona{
9          private:
10         static int nPersonas;
11         static const int MAX;
12         char *nombre;
13
14         public:
15         Persona(const char *c=NULL){
16                 if(c!=NULL){
17                         nombre= new char[strlen(c)+1];
18                         strcpy(nombre, c);
19                         cout<<"Persona: "<<nombre<<endl;
20                 }else{
21                         nombre=NULL;
22                         cout<<"Persona: "<<endl;
23                 }
24                 nPersonas++;
25         }
26
27         ~Persona(){
28                 cout<<"eliminando persona : "<<nombre<<endl;
29                 if(nombre)
30                         delete []nombre;
31                 nPersonas--;
32         }
33
34         static int getMax(){
35                 return MAX;
36         }
37
38         static int getnPersonas(){
39                 return nPersonas;
40         }
41 };
```

```
42 int Persona::nPersonas=0;
43 const int Persona::MAX=10;
44
45 int main() {
46
47         cout<<"Máximo de personas: "<<Persona::getMax()<<endl;
48         cout<<"Número de personas: "<<Persona::getnPersonas()<<endl;
49
50         Persona per1;
51         cout<<"Máximo de personas: "<<Persona::getMax()<<endl;
52         cout<<"Número de personas: " <<Persona::getnPersonas()<<endl;
53
54         Persona per2("persona 2");
55         cout<<"Máximo de personas: "<<per2.getMax()<<endl;
56         cout<<"Número de personas: "<<per2.getnPersonas()<<endl;
57         return 0;
58 }
```

Listing 46. Ejemplo miembros estáticos en C++.

9.2. Miembros estáticos en Java

Un miembro estático en Java se maneja de la misma forma que en C++. Cada uno de los objetos tiene su propio estado independiente del resto de los objetos, compartiendo al mismo tiempo un estado común al tener todos los objetos acceso al estado de la clase, el cual es único y existe de forma independiente.

Ejemplo:

```
1  public class Objeto{
2      private String nombre;
3          private static int numObjetos;
4          public Objeto(String cadena){
5                  if(cadena.length()!=0)
6                          nombre=cadena;
7                  else
8                          nombre="cadena por omision";
9                  numObjetos++;
10         }
11
12         public static int getNumObjetos(){
13                 return numObjetos;
14         }
15
16         public static void main(String argv[]) {
17                 System.out.println("Objetos: " + getNumObjetos());
18                 System.out.println("Objetos: " + Objeto.getNumObjetos());
19                 Objeto uno,dos;
20                 uno= new Objeto("");
21                 dos= new Objeto("Objeto dos");
22                 System.out.println("Objetos: " + uno.getNumObjetos());
23                 System.out.println("Objetos: " + dos.getNumObjetos());
24         }
25 }
```

Listing 47. Ejemplo de miembros estáticos en Java.

Una diferencia que se puede apreciar en este ejemplo con respecto a C++, es que no estamos obligados a inicializar el elemento estático, pues éste es inicializado automáticamente.

9.3. Miembros estáticos / de clase en Python

En Python existe el concepto de atributos de clase y métodos de clase, pero estos últimos se diferencian de los métodos estáticos como se explica a continuación.

9.3.1. Atributos de clase

En Python los **atributos de clase** en realidad son los atributos que se definen **dentro de la clase**. Los atributos de instancia (*data attributes* en Python) son definidos dentro del método *__init__*.

9.3.2. Método de clase y Método estático

En Python existe una ligera diferencia entre un método de clase y un método estático. Son definidos usando *decorators*: *@staticmethod* y *@classmethod*. Un método de clase recibe a la clase como primer argumento, de la misma forma que un método de instancia recibe a la instancia como primer argumento, por lo que tiene el ámbito de la clase. Como en Java, el método puede ser llamado mediante el nombre de la clase o el de un objeto de la clase. El método estático por su parte, no recibe un argumento implícito[1], por lo que no pueden modificar ni el estado de un objeto ni el de la clase. Un ejemplo marcando la diferencia:

```
1  class A:
2      def metodo(self,x):
3          print ("ejecutando metodo(%s,%s)"%(self,x))
4
5      @classmethod
6      def metodo_de_clase(cls,x):
7          print ("ejecutando metodo de clase(%s,%s)"%(cls,x))
8
9      @staticmethod
10     def metodo_estatico(x):
11         print ("ejecutando metodo estatico(%s)"%x)
12
13 #Script de ejecución
14 a=A()
15
16 #La llamada normal a un método de instancia.
17 #El objeto es pasado de manea implícita como primer argumento
18 a.metodo(1)
```

[1] Class method vs static method 2015, http://www.bogotobogo.com/python/python_differences_between_static_method_and_class_method_instance_method.php

```
# salida: ejecutando metodo(<__main__.A object at 0xb7dbef0c>,1)

# En un metodo de clase, la clase del objeto es pasada
# implicitamente como primer argumento
a.metodo_de_clase(1)
# salida: ejecutando metodo de clase(<class '__main__.A'>,1)

# Tambien puede ejecutarse un metodo de clase usando
# el nombre de la clase
A.metodo_de_clase(1)
# salida: ejecutando metodo de clase(<class '__main__.A'>,1)

# Un metodo estatico no pasa implicitamente ni al objeto
# ni la clase como primer argumento
a.metodo_estatico(1)
# salida: ejecutando metodo estatico(1)

#metodo esta confinado al objeto a
print(a.metodo)

#metodo_de_clase esta confinado a la clase A, no a al objeto a
print(a.metodo_de_clase)

# metodo_estatico no esta confinado ni al objeto
# ni a la clase, no va dicha informacion como argumento
print(a.metodo_estatico)
print(A.metodo_estatico)
```

Listing 48. Ejemplo de métodos estáticos y de clase en Python.

Ejemplo:

```
class Persona:
    nPersonas=0
    MAX=100     # no es una constante
    def __init__(self, nom):
        self.nombre=nom
        print("Persona: " + self.nombre)
        Persona.nPersonas+=1

    @classmethod
    def getMax(cls):
```

```
11          return cls.MAX
12
13      @classmethod
14      def getnPersonas(cls):
15          return cls.nPersonas
16
17      def getnPersonas2(self):
18          return self.nPersonas
19
20  #código principal
21
22  print("Numero maximo de personas:", Persona.getMax() )
23  print("Numero de personas:", Persona.getnPersonas() )
24
25  per1 = Persona("Persona 1")
26  print("Numero maximo de personas: ", Persona.getMax() )
27  print("Numero de personas: ", Persona.getnPersonas() )
28
29  per2 = Persona("Persona 2")
30  print(Persona.getMax() )
31  print(per2.getnPersonas() )
32  print(per2.getnPersonas2() )
```

Listing 49. Ejemplo de métodos de clase en Python.

Capítulo 10

Objetos constantes

Algunas ocasiones puede ser útil tener objetos constantes, los cuales no puedan ser modificados. Sin embargo, cada lenguaje interpreta de manera ligeramente distinta, como se verá a continuación.

10.1. Objetos constantes en C++

En C++, es posible tener objetos de tipo constante, los cuales no podrán ser modificados en ningún momento[1] Tratar de modificar un objeto constante se detecta como un error en tiempo de compilación.

Sintaxis
`const <clase> <lista de objetos>;`

Ejemplo:

```
const Hora h1(9,30,20);
```

Para estos objetos, algunos compiladores llegan a ser tan rígidos en el cumplimiento de la instrucción, que no permiten que se hagan llamadas a métodos sobre esos objetos. La compilación estándar permite la ejecución de métodos, siempre y cuando no modifiquen el estado del objeto.

Si se quiere consultar al objeto mediante llamadas a métodos get, lo correcto es declarar métodos con la palabra reservada *const*, para permitirles actuar libremente sobre los objetos sin modificarlo. La sintaxis requiere añadir después de la lista de parámetros la palabra reservada *const* en la declaración y en su definición.

[1] Ayuda a cumplir el principio del mínimo privilegio, donde se debe restringir al máximo el acceso a los datos cuando este acceso estaría de sobra[9].

Sintaxis

Declaración.

```
<tipo> <nombre> (<parámetros>) const;
```

Definición del método fuera de la declaración de la clase.

```
<tipo> <clase> :: <nombre> (<parámetros>) const {
        <código>
}
```

Definición del método dentro de la declaración de la clase.

```
<tipo> <nombre> (<parámetros>) const {
        <código>
}
```

Los compiladores generalmente restringen el uso de métodos constantes a objetos constantes. Para solucionarlo es posible sobrecargar el método con la única diferencia de la palabra *const*, aunque el resto de la firma del método sea la misma.

Un método puede ser declarado dos veces tan sólo con que la firma del método difiera por el uso de *const*. Objetos constantes ejecutarán al método definido con *const*, y objetos variables ejecutarán al método sin esta restricción. De hecho, un objeto variable puede ejecutar el método no definido con *const* por lo que si el objetivo del método es el mismo, y este no modifica al objeto (e.g., métodos tipo *get*) bastaría con definir al método una vez[2].

Los constructores no necesitan la declaración *const*, puesto que deben poder modificar al objeto.

Ejemplo:

```
1  #include <iostream>
2  #include <time.h>
3  #include <stdlib.h>
4
5  using namespace std;
6
7  class Arr{
8          private:
```

[2]Además, declarar a los métodos *get* y otros métodos que no modifican al objeto con el calificador *const* es una buena práctica de programación.

```
9         int a[10];
10        public:
11        Arr(int x=0) {
12                srand( time(NULL) );
13                for( int i=0; i<10; i++){
14                        if (x==0)
15                                x=rand()%100;
16                        a[i]=x;
17                }
18        }
19        char set(int, int);
20        int get(int) const ;
21        int get(int);
22};
23
24char Arr::set(int pos, int val ){
25        if(pos>=0 && pos<10){
26                a[pos]=val;
27                return 1;
28        }
29        return 0;
30}
31
32int Arr::get(int pos) const {
33        if(pos>=0 && pos<10)
34                return a[pos];
35//   a[9]=0;  error en un método constante
36        return 0;
37}
38
39int Arr::get(int pos) {  //no es necesario sobrecargar
40        if(pos>=0 && pos<10)   // si el método no modifica
41                return a[pos];
42        return 0;
43}
44
45int main(){
46        const Arr a(5),b;
47        Arr c;
48
49//        a.set(0,1);  //error llamar a un método no const
50//        b.set(0,2);         // para un objeto constante (comentar estas lineas)
```

```
51          c.set(0,3);
52  //        a.set(1,11); //error llamar a un método no const
53  //        b.set(1,22); // para un objeto constante (comentar estas lineas)
54          c.set(1,33);
55          cout<<a.get(0)<<endl; // ejecuta int get(int) const ;
56          cout<<a.get(1)<<endl;
57          cout<<b.get(0)<<endl;
58          cout<<b.get(1)<<endl;
59          cout<<c.get(0)<<endl; // ejecuta int get(int);
60          cout<<c.get(1)<<endl;
61          return 0;
62  }
```

Listing 50. Ejemplo de objetos constantes en C++.

10.2. Objetos finales en Java

Ya se mencionó en la sección de fundamentos de Java el uso de la palabra reservada *final*, la cual permite a una variable ser inicializada sólo una vez. En el caso de los objetos o referencias a los objetos el comportamiento es el mismo. Si se agrega la palabra *final* a la declaración de una referencia a un objeto, significa que la variable podrá ser inicializada una sola vez, en el momento que sea necesario.

Sintaxis
`final <clase> <lista de identificadores de objetos>;`

Por ejemplo:

```
final Hora h1= new Hora(9,30,20);
```

Concepto
Es importante remarcar que no es el mismo sentido de *const* en C++. Aquí lo único que se limita es la posibilidad de una variable de referencia a ser inicializada de nuevo, pero no inhibe la modificación de miembros.

Por ejemplo:

```
1  final Light aLight = new Light();  // variable local final
2  aLight.noOfWhatts = 100;          //Ok. Cambio en el edo. del objeto
3
4  aLight = new Light();          // Inválido. No se puede modificar la referencia
```

Ejemplo:

```
class Fecha {
        private int dia;
        private int mes, año;

        public Fecha(){
                dia=mes=1;
                año=1900;
        }
        public boolean setDia(int d){
                if (d >=1 && d<=31){
                        dia= d;
                        return true;
                }
                return false;

        }   //poner día
        public int getDia()        {
                return dia;
        } //devuelve día
        public boolean setMes(int m){
                if (m>=1 && m<=12){
                        mes=m;
                        return true;
                }
                return false;
        }
        public int getMes(){
                return mes;
        }

        public boolean setAño(int a) {
                if (a>=1900){
                        año=a;
                        return true;
                }
                return false;
        }
        public int getAño() {
                return año;
        }
```

```
41 }
42
43 public class MainF {
44
45         public static void main(String[] args) {
46                 final Fecha f;
47
48                 f= new Fecha();
49
50                 f.setDia(10);
51                 f.setMes(3);
52                 f.setAño(2001);
53                 System.out.println(f.getDia()+"/"+f.getMes()+"/"+f.getAño());
54                 f= new Fecha(); //Error: la variable f es final y no puede ser reasignada
55         }
56 }
```

Listing 51. Ejemplo de objetos finales en Java.

10.3. Objetos constantes en Python

Python no cuenta con objetos ni variables constantes. Se acostumbra definir variables de referencia a objetos que no se deben cambiar con mayúsculas, pero no existe ningún mecanismo que el lenguaje proporciona para evitar la modificación de los objetos.

Capítulo 11

Amistad en C++

En C++ existe el concepto de **amistad**. Aunque puede ser considerado por algunos como una intrusión a la encapsulación o a la privacidad de los datos:

Concepto
"... la amistad corrompe el ocultamiento de información y debilita el valor del enfoque de diseño orientado a objetos"[10]

Un amigo de una clase es una función u otra clase que no es miembro de la clase, pero que tiene permiso de usar los miembros públicos y privados de la clase[1].

Es importante señalar que el ámbito de una función amiga no es el de la clase, y por lo tanto los amigos no son llamados con los operadores de acceso de miembros.

[1]También tiene acceso a los miembros protegidos que se verán más adelante.

Sintaxis

Sintaxis para una función amiga:

```
class <nombreClase> {
        friend <tipo> <metodo>();
...
public:
        ...
};
```

Sintaxis para una clase amiga:

```
class <nombreClase> {
        friend <nombreClaseAmiga>;
...
public:
        ...
};
```

Las funciones o clases amigas no son privadas ni públicas (o protegidas), pueden ser colocadas en cualquier parte de la definición de la clase, pero se acostumbra que sea al principio.[Deitel, 1995]

Como la amistad entre personas, esta es **concedida** y no tomada. Si la clase B quiere ser amigo de la clase A, la clase A debe declarar que la clase B es su amiga.

La amistad **no es simétrica ni transitiva**: si la clase A es un amigo de la clase B, y la clase B es un amigo de la clase C, no implica:

- Que la clase B sea un amigo de la clase A.
- Que la clase C sea un amigo de la clase B.
- Que la clase A sea un amigo de la clase C.

El concepto de amistad no esta implementado en otros lenguajes aunque el nivel protegido permite un cierto nivel de acceso miembros de clases del mismo módulo an algunos lenguajes.

Ejemplo:

```
1 //Ejemplo de funcion amiga con acceso a miembros privados
2 #include <iostream>
3
```

```
4  using namespace std;
5
6  class ClaseX{
7          friend void setX(ClaseX &, int);  //declaración friend
8          public:
9          ClaseX(){
10                 x=0;
11         }
12         void print() const {
13                 cout<<x<<endl;
14         }
15         private:
16         int x;
17 };
18
19 void setX(ClaseX &c, int val){
20         c.x=val;        //es legal el acceso a miebros privados por amistad.
21 }
22
23 int main(){
24         ClaseX pr;
25
26         cout<<"pr.x después de instanciación : ";
27         pr.print();
28         cout<<"pr.x después de la llamada a la función amiga setX : ";
29         setX(pr, 10);
30         pr.print();
31 }
```

Listing 52. Ejemplo de funciones amigas en C++.

```
1  //ejemplo 2 de funciones amigas
2  #include <iostream>
3  using namespace std;
4
5  class Linea;
6
7  class Recuadro {
8          friend int mismoColor(Linea, Recuadro);
9
10         private:
```

```
11          int color; //color del recuadro
12          int xsup, ysup; //esquina superior izquierda
13          int xinf, yinf; //esquina inferior derecha
14
15          public:
16          void ponColor(int);
17          void definirRecuadro(int, int, int, int);
18  };
19
20  class Linea{
21          friend int mismoColor(Linea, Recuadro);
22
23          private:
24          int color;
25          int xInicial, yInicial;
26          int lon;
27
28          public:
29          void ponColor(int);
30          void definirLinea(int, int, int);
31  };
32
33  int mismoColor(Linea l, Recuadro r){
34          if(l.color==r.color)
35                  return 1;
36          return 0;
37  }
38
39  //métodos de la clase Recuadro
40  void Recuadro::ponColor(int c) {
41          color=c;
42  }
43
44  void Recuadro::definirRecuadro(int x1, int y1, int x2, int y2) {
45          xsup=x1;
46          ysup=y1;
47          xinf=x2;
48          yinf=y2;
49  }
50
51  //métodos de la clase Linea
52  void Linea::ponColor(int c) {
```

```
53          color=c;
54  }
55
56  void Linea::definirLinea(int x, int y, int l) {
57          xInicial=x;
58          yInicial=y;
59          lon=l;
60  }
61
62  int main(){
63          Recuadro r;
64          Linea l;
65
66          r.definirRecuadro(10, 10, 15, 15);
67          r.ponColor(3);
68          l.definirLinea(2, 2, 10);
69          l.ponColor(4);
70          if(!mismoColor(l, r))
71                  cout<<"No tienen el mismo color"<<endl;
72          //se ponen en el mismo color
73          l.ponColor(3);
74          if(mismoColor(l, r))
75                  cout<<"Tienen el mismo color";
76          return 0;
77  }
```

Listing 53. Ejemplo 2 de funciones amigas en C++.

Capítulo 12

Polimorfismo AdHoc: Sobrecarga de operadores

La sobrecarga de operadores es la capacidad de definir nuevo comportamiento para operadores existentes en un lenguaje con tipos de datos definidos por el usuario. De esta forma, en programación orientada a objetos, se les da a los operadores un nuevo significado de acuerdo al objeto sobre el cual se aplique.

C++, Ruby, Scala[1], C#[2] y D[3] permiten la sobrecarga de de operadores. Java no permite la sobrecarga de operadores. Python cuenta con una aproximación a la sobrecarga de operadores.

Concepto
Para sobrecargar un operador, se define un método que es invocado cuando el operador es aplicado sobre ciertos tipos de datos.

[1]Scala: Overloading Operators, https://j2eethoughts.wordpress.com/2010/10/13/scala-overloading-operators/
[2]Operator Overloading Tutorial, https://msdn.microsoft.com/en-us/library/aa288467(v=vs.71).aspx
[3]Operator Overloading, http://dlang.org/operatoroverloading.html

12.1. Sobrecarga de operadores en C++

La sintaxis para definir un método con un operador difiere de la definición normal de un método, pues el nombre del método esta definido por la palabra reservada *operator* y el operador que se va a sobrecargar:

Sintaxis

```
<tipo> operator <operador> (<argumentos>) ;

o

<tipo> operator <operador> (<argumentos>) {
<cuerpo del método>
}
```

Para la definición fuera de la clase:

```
<tipo> <clase>::operator <operador> (<argumentos>) {
<cuerpo del método>
}
```

Para utilizar un operador con objetos, es necesario que el operador este sobrecargado, aunque existen dos excepciones:

- El **operador de asignación =**, puede ser utilizado sin sobrecargarse explícitamente, pues el comportamiento por omisión es una **copia a nivel de miembro** de los miembros de la clase. Sin embargo no debe de usarse si la clase cuenta con miembros a los que se les asigne memoria de manera dinámica.
- El **operador de dirección** , esta sobrecargado por omisión para devolver la dirección de un objeto de cualquier clase.

Algunas restricciones:

1. Operadores que no pueden ser sobrecargados: $..* ::? : sizeof$
2. La precedencia de un operador no puede ser modificada. Deben usarse los paréntesis para obligar un nuevo orden de evaluación.
3. La asociatividad de un operador no puede ser modificada.

4. No se puede modificar el número de operandos de un operador. Los operadores siguen siendo unarios o binarios.

5. No es posible crear nuevos operadores.

6. No puede modificarse el comportamiento de un operador sobre tipos de datos definidos por el lenguaje.

Ejemplo:

```
1  //programa de ejemplo de sobrecarga de operadores.
2  class Punto {
3          float x, y;
4  public:
5          Punto(float xx=0, float yy=0){
6                  x=xx;
7                  y=yy;
8          }
9          float magnitud();
10         Punto operator =(Punto);
11         Punto operator +(Punto);
12         Punto operator -();
13         Punto operator *(float);
14         Punto operator *=(float);
15         Punto operator ++();  //prefijo
16         Punto operator ++(int);  //posfijo
17         int operator >(Punto);
18         int operator <=(Punto);
19 };
20
21 Punto Punto::operator =(Punto a){ //copia o asignación
22         x=a.x;
23         y=a.y;
24         return *this;
25 }
26
27 Punto Punto::operator +(Punto p){
28         return Punto(x+p.x, y+p.y);
29 }
30
31 Punto Punto::operator -(){
32         return Punto(-x, -y);
33 }
```

```
34
35  Punto Punto::operator *(float f){
36          Punto temp;
37          temp=Punto(x*f, y*f);
38          return temp;
39  }
40
41  // incremento prefijo
42  Punto Punto::operator ++(){
43          x++;
44          y++;
45          return *this;
46  }
47
48  // incremento posfijo
49  Punto Punto::operator++(int)
50  {
51      Punto temp= *this;
52      x++;
53      y++;
54      return temp;
55  }
56
57  int Punto::operator >(Punto p){
58          return (x>p.x && y>p.y) ? 1 : 0;
59  }
60
61  int Punto::operator <=(Punto p){
62          return (x<=p.x && y<=p.y) ? 1 : 0;
63  }
64
65  int main(){
66          Punto a(1,1);
67          Punto b;
68          Punto c;
69
70          b++;
71          ++b;
72          c=b;
73          c=a+b;
74          c=-a;
75          c=a*.5;
```

```
76
77          return 0;
78  }
```

Listing 54. Ejemplo de sobrecarga de operadores con C++.

Ejercicio

Crear una clase *String* para el manejo de cadenas. Tendrá dos atributos: apuntador a carácter y un entero *tam*, para almacenar el tamaño de la cadena. Sobrecargar operadores $=$ (asignación), $(==)$ igualdad, $!=, <, >, <=, >=, +, +=, []$. Generar un programa de prueba para objetos de la clase. La estructura inicial será la siguiente:

```
class String{
   char *s;
   int tam;
public:
   String(char * =NULL);
   String(const String &copia);  //constructor de copia
   ~String();
//sobrecarga de operador de asignación
   const String &operator =(const String &);
   //igualdad
   int operator ==(const String &) const ;
};
```

Ejemplo: código de *String*

Véase que es posible asignar una cadena sin sobrecargar el operador de asignación, o comparar un objeto *String* con una cadena. Esto se logra gracias a que se provee de un constructor que convierte una cadena a un objeto *String*. De esta manera, este **constructor de conversión** es llamado automáticamente, creando un objeto temporal para ser comparado con el otro objeto.

No es posible que la cadena (o apuntador a char) vaya del lado izquierdo, pues se estaría llamando a la funcionalidad del operador para un apuntador a char. Si se requiere que el operando izquierdo sea de un tipo distinto al de la clase, la sobrecarga del operador debe hacerse usando una función externa a la clase. Si fuera necesario, esta función podría declararse como amiga de la clase.

```
1  //Sobrecarga de operadores. Implementación de una clase String
2  #include <iostream>
3  #include <stdio.h>
4  #include <string.h>
5
6  using namespace std;
7
8  class String{
9          //operadores de salida de flujo
10         friend ostream &operator << (ostream &, const String &);
11 private:
12         char *s;
13         int tam;
14 public:
15         String(const char * =NULL);
16
17         String(const String &copia){
18                 s=NULL;
19                 tam=0;
20                 *this=copia;//¿se vale o no?
21         }
22         ~String(){
23                 if(s!=NULL){
24                         delete []s;
25                         s=NULL;
26                         tam=0;
27                 }
28         }
29
30         //sobrecarga de constructor de asignación
```

```
31          const String &operator =(const String &);
32
33          //igualdad
34          int operator ==(const String &) const ;
35
36          //concatenación
37          String operator +(const String &) const;
38
39          //concatenación y asignación
40          const String &operator +=(const String &);
41
42          String &copia (const String &);
43
44          //sobrecarga de los corchetes
45          char &operator[] (int) const;
46  };
47
48  //operadores de inserción y extracción de flujo
49  ostream& operator<< (ostream &salida, const String &cad){
50          salida<<cad.s;
51          return salida;  //permite concatenación
52  }
53
54  istream &operator >> (istream &entrada, String &cad){
55          char tmp[100];
56          entrada >> tmp;
57          cad=tmp; //usa operador de asignación de String y const. de conversión
58          return entrada;         //permite concatenación
59  }
60
61  String::String(const char *c){
62          if(c==NULL){
63                  s=NULL;
64                  tam=0;
65          } else {
66                  tam=strlen(c);
67                  s= new char[tam+1];
68                  strcpy(s, c);
69          }
70  }
71
72  const String &String::operator =(const String &c){
```

```
73          if(this!= &c) {      //verifica no asignarse a si mismo
74                  if(s!=NULL)
75                          delete []s;
76                  tam=c.tam;
77                  s= new char[tam+1];
78                  strcpy(s, c.s);
79          }
80          return *this; //permite concatenación de asignaciones
81  }
82
83  int String::operator ==(const String &c)const {
84                  return strcmp(s, c.s)==0;
85  }
86
87  //operador de suma regresa una copia de la suma obtenida
88  //en un objeto local.
89  String String::operator +(const String &c) const {
90          String tmp(*this);
91          tmp+=c;
92          return tmp;
93  }
94
95  const String &String::operator +=(const String &c){
96          char *str=s, *ctmp= new char [c.tam+1];
97          strcpy(ctmp, c.s);
98          tam+=c.tam;
99          s= new char[tam+1];
100         strcpy(s, str);
101         strcat(s, ctmp);
102         delete []str;
103         delete []ctmp;
104         return *this;
105 }
106
107 String &String::copia (const String &c) {
108         if(this!= &c) {      //verifica no asignarse a si mismo
109                 if(s!=NULL)
110                         delete []s;
111                 tam=c.tam;
112                 s= new char[tam+1];
113                 strcpy(s, c.s);
114         }
```

```
115         return *this; //permite concatenación de asignaciones
116 }
117
118 char &String::operator[] (int i) const {
119         if(i>=0 && i<tam)
120                 return s[i];
121         return s[0];
122 }
123
124 int main(){
125         String a("AAA");
126         String b("Prueba de cadena");
127         String c(b);
128     // Es un error hacer una asignación sin liberar memoria.
129     // ese es el principal peligro de usar el operador sobrecargado
130     // por default de asignación
131         a=b;
132         b.copia("H o l a");
133         b=c+c;
134         b="nueva";
135         c+=c;
136         String d("nueva cadena");
137         d+="Hola";
138         String e;
139         e=d+"Adios";
140         d="coche";
141         int x=0;
142         x=d=="coche"; //Lo contrario no es válido "coche"==d
143         char ch;
144         ch=d[7];
145         d[2]='X';
146         cout<<d<<endl;
147         cout<<"Introduce dos cadenas:";
148         cin>>e>>d;
149         cout<<"Cadenas:\n";
150         cout<<e<<endl<<d;
151         return 0;
152 }
```

Listing 55. Ejemplo de String con sobrecarga de operadores en C++.

12.2. Sobrecarga de operadores en Python

Python permite la sobrecarga de operadores mediante la definición de métodos especiales[4]. Dichos métodos inician y terminan con doble '_'. Por ejemplo, si se quiere sobrecargar la igualdad (==) se tiene que crear una definición del método *__eq__*.

Ejemplo:

```
#programa de ejemplo de sobrecarga de operadores.
class Punto:

  def __init__(self,x=10, y=10 ):
    self.x=x
    self.y=y

  def __add__(self, op):
    return Punto(self.x+op.x, self.y+op.y)

  def __sub__(self, op):
    return Punto(self.x-op.x, self.y-op.y)

  def __mul__(self, f):
    temp=Punto(self.x*f, self.y*f)
    return temp

  def __gt__(self, op):  # >
    return (self.x>op.x and self.y>op.y)

  def __le__ (self, op): # <=
    return (self.x<=op.x and self.y<=op.y)

# principal
a= Punto(1,1)
b= Punto()
c= Punto()
print( a.x)
print(b.x)
print(c.x)

c=a+b
print(c.x)
```

[4]Ver más sobre métodos especiales en: https://docs.python.org/3/reference/datamodel.html#special-method-names

```
34 print(c.y)
35 c+=a
36 print(c.x)
37 print(c.y)
38
39 p=a-b
40 print(p.x)
41 print(p.y)
42
43 c=a*0.5
44 print(c.x)
45 print(c.y)
46
47 print( a>b)
48 print( a<=b)
```

Listing 56. Ejemplo de sobrecarga de operadores en Python.

Una lista de las funciones para la sobrecarga de operadores se muestra a continuación[5].

Sintaxis

`__add__(self,other) → value: self + other`
`__sub__(self,other) → value: self-  other`
`__mul__(self,other) → value: self * other`
`__div__(self,other)→value: self/other`
`__mod__(self,other) → value: self % other`
`__pow__(self,other) → value: self ** other`
`__neg__(self) → value:- self`
`__pos__(self) → value: +self`
`__abs__(self) → value: abs(self)`
`__iadd__(self,other)  self += other`
`__isub__(self,other)  self- = other`
`__imul__(self,other)  self *= other`
`__idiv__(self,other) self/=other`
`__imod__(self,other)  self %= other`
`__ipow__(self,other)  self **= other`

[5]Fuente: Pointal, Lauren; Python 2.4 Quick Reference Card, 2006.

Ejercicio
Crear un programa con una clase *Pila* y los métodos correspondientes (*push, pop, getTope, getElemTope, estaVacia, estaLlena, ...*) ocupando los temas OO antes vistos de acuerdo a lo que cada lenguaje permita (constructores, miembros estáticos, sobrecarga de operaciones, sobrecarga de operadores, funciones amigas)

Capítulo 13

Herencia

13.1. Introducción

La **herencia** es un mecanismo potente de abstracción que permite compartir similitudes entre clases manteniendo al mismo tiempo sus diferencias.

Es una forma de reutilización de código, tomando clases previamente creadas y formando a partir de ellas nuevas clases, heredándoles sus atributos y métodos. Las nuevas clases pueden ser modificadas agregándoles nuevas características.

Los términos para distinguir los tipos de clases pueden variar. Por ejemplo, en C++ la clase de la cual se toman sus características se conoce como **clase base**; mientras que la clase que ha sido creada a partir de la clase base se conoce como **clase derivada**. Existen otros términos para estas clases:

En Java es más común usar el término de superclase y subclase.

Una clase derivada es potencialmente una clase base, en caso de ser necesario.

Cada objeto de una clase derivada también es un objeto de la clase base. En cambio, un objeto de la clase base no es un objeto de la clase derivada.

La implementación de herencia a varios niveles forma un árbol jerárquico similar al de un árbol genealógico. Esta es conocida como **jerarquía de herencia**.

Generalización. Una clase base o superclase se dice que es más general que la clase derivada o subclase.

Especialización. Una clase derivada es por naturaleza una clase más especializada que su clase base.

Clase base	Clase derivada
Superclase	Subclase
Clase padre	Clase hija

13.2. Herencia: Implementación en C++

La herencia en C++ es implementada permitiendo a una clase incorporar a otra clase dentro de su declaración.

Sintaxis

```
class <claseDerivada>: <acceso>  <claseBase> {
        //cuerpo clase derivada
};
```

Ejemplo: *Una clase vehículo que describe a todos aquellos objetos vehículos que viajan en carreteras. Puede describirse a partir del número de ruedas y de pasajeros. De la definición de vehículos podemos definir objetos más específicos (especializados). Por ejemplo la clase camión.*

```
1  #include <iostream>
2  using namespace std;
3
4  class Vehiculo{
5          int ruedas;
6          int pasajeros;
7  public:
8          void setRuedas(int);
9          int  getRuedas();
10         void setPasajeros(int);
11         int  getPasajeros();
12 };
13
14 void Vehiculo::setRuedas(int num){
15         ruedas=num;
16 }
17 int Vehiculo::getRuedas(){
18         return ruedas;
19 }
20 void Vehiculo::setPasajeros(int num){
21         pasajeros=num;
22 }
23
```

```
24 int Vehiculo::getPasajeros(){
25         return pasajeros;
26 }
27
28 //clase Camion con herencia  de Vehículo
29 class Camion: public Vehiculo {
30         int carga;
31         public:
32         void setCarga(int);
33         int getCarga();
34         void muestra();
35 };
36
37 void Camion::setCarga(int num){
38         carga=num;
39 }
40 int Camion::getCarga(){
41         return carga;
42 }
43 void Camion::muestra(){
44         cout<<"Ruedas: "<< getRuedas()<<endl;
45         cout<<"Pasajeros: "<< getPasajeros()<<endl;
46         cout<<"Capacidad de carga: "<<getCarga()<<endl;
47 }
48
49 int main(){
50         Camion ford;
51         ford.setRuedas(6);
52         ford.setPasajeros(3);
53         ford.setCarga(3200);
54         ford.muestra();
55         return 0;
56 }
```

Listing 57. Ejemplo de herencia en C++, jerarquía de Vehículo.

13.2.1. Control de Acceso a miembros en C++

Existen tres palabras reservadas para el control de acceso: *public*, *private* y *protected*. Estas sirven para proteger los miembros de la clase en diferentes formas.

El control de acceso, como ya se vio anteriormente, se aplica a los métodos, atributos, constantes y tipos anidados que son miembros de la clase.

Resumen de tipos de acceso:

- **private**. Un miembro privado únicamente puede ser utilizado por los métodos miembro y funciones amigas de la clase donde fue declarado.
- **protected**. Un miembro protegido puede ser utilizado únicamente por los métodos miembro y funciones amigas de la clase donde fue declarado o por los métodos miembro y funciones amigas de las clases derivadas. El acceso protegido es como un nivel intermedio entre el acceso privado y público.
- **public**. Un miembro público puede ser utilizado por cualquier método. Una estructura es considerada por C++ como una clase que tiene todos sus miembros públicos.

```
1  //ejemplo de control de acceso
2
3  class S{
4          char *f1();
5          int a;
6           protected:
7          int b;
8          int f2();
9           private:
10         int c;
11         int f3();
12         public:
13         int d, f;
14         char *f4(int);
15 };
16
17 int main(){
18         S obj;
19         obj.f1(); //error
20         obj.a=1;        //error
21         obj.f2();//error
22         obj.b=2; //error
23         obj.c=3; //error
24         obj.f3(); //error
25         obj.d=5;
26         obj.f4(obj.f);
27         return 0;
28 }
```

Listing 58. Ejemplo de herencia y control de acceso a miembros en C++.

13.2.2. Control de acceso en herencia en C++

Hasta ahora se ha usado la herencia con un solo tipo de acceso, utilizando el especificador *public*. Los miembros públicos de la clase base son miembros públicos de la clase derivada, los miembros protegidos permanecen protegidos para la clase derivada.

Ejemplo:

```
1  //ejemplo de control de acceso en herencia
2
3  class Base{
4          int a;
5          protected:
6          int b;
7           public:
8          int c;
9  };
10
11 class Derivada: public Base {
12         void g();
13 };
14
15 void Derivada::g(){
16         a=0; //error, es privado
17         b=1; //correcto, es protegido
18         c=2; //correcto, es público
19 }
```

Listing 59. Ejemplo control de acceso en herencia en C++.

Para accesar a los miembros de una clase base desde una clase derivada, se pueden ajustar los permisos por medio de un calificador *public*, *private* o *protected*.

Si una clase base es declarada como **pública** de una clase derivada, los miembros públicos y protegidos son accesibles desde la clase derivada, no así los miembros privados.

Si una clase base es declarada como **privada** de otra clase derivada, los miembros públicos y protegidos de la clase base serán miembros privados de la clase derivada. Los miembros privados de la clase base permanecen inaccesibles.

Si se omite el calificador de acceso de una clase base, se asume **por omisión** que el calificador es *public* en el caso de una estructura y *private* en el caso de una clase.

Ejemplo de sintaxis:

```
1 class base {
2  ...
3 };
```

```
4
5  class d1: private base {
6  ...
7  };
8
9  class d2: base {
10 ...
11 };
12
13 class d3: public base {
14 ...
15 };
```

Es recomendable declarar explícitamente la palabra reservada *private* al tomar una clase base como privada para evitar confusiones:

```
1  class x{
2            public:
3      f();
4  };
5
6  class y: x {   //privado por omisión
7           ...
8  };
9
10 void g( y *p){
11          p->f();  //error
12 }
```

Finalmente, si una clase base es declarada como **protegida** de una clase derivada, los miembros públicos de la clase base se convierten en miembros protegidos de la clase derivada.

Ejemplo:

```
1  //acceso por herencia
2  #include <iostream>
3  using namespace std;
4
5  class X{
6          protected:
7          int i;
8          int j;
9
```

```
10          public:
11          void preg_ij();
12          void pon_ij();
13  };
14
15  void X::preg_ij() {
16          cout<< "Escriba dos números: ";
17          cin>>i>>j;
18  }
19
20  void X::pon_ij() {
21          cout<<i<<' '<<j<<endl;
22  }
23
24  //en Y, i y j de X siguen siendo miembros protegidos
25  //Si se llegara a cambiar este acceso a private i y j se heredarian como
26  // miembros privados de Y, además de los métodos públicos
27  class Y: public X{
28          int k;
29  public:
30          int preg_k();
31          void hacer_k();
32  };
33
34  int Y:: preg_k(){
35          return k;
36  }
37
38  void Y::hacer_k() {
39          k=i*j;
40  }
41
42  // Z tiene acceso a i y j de X, pero no a k de Y
43  // porque es private por omisión
44  // Si Y heredara de x como private, i y j serían privados en Y,
45  // por lo que no podrían ser accesados desde Z
46  class Z: public Y {
47  public:
48          void f();
49  };
50
51  // Si Y heredara a X con private, este método ya no funcionaría
```

```
52 // no se podría acceder a i ni a j.
53 void Z::f() {
54         i=2;
55         j=3;
56 }
57
58 // si Y hereda de x como private, no es posible accesar a los métodos
59 //públicos desde objetos de Y ni de Z.
60 int main() {
61         Y var;
62         Z var2;
63
64         var.preg_ij();
65         var.pon_ij();
66         var.hacer_k();
67         cout<<var.preg_k()<<endl;
68         var2.f();
69         var2.pon_ij();
70         return 0;
71 }
```

Listing 60. Ejemplo control de acceso en herencia con C++.

13.2.3. Manejo de objetos de la clase base como objetos de una clase derivada y viceversa en C++

Un objeto de una clase derivada pública, puede ser manejado como un objeto de su clase base. Sin embargo, un objeto de la clase base no es posible tratarlo de forma automática como un objeto de clase derivada.

La opción que se puede utilizar, es enmascarar un objeto de una clase base a un apuntador de clase derivada. El problema es que no debe ser desreferenciado (accedido) así, primero se tiene que hacer que el objeto sea referenciado por un apuntador de su propia clase.

Si se realiza la conversión explícita de un apuntador de clase base - que apunta a un objeto de clase base - a un apuntador de clase derivada y, posteriormente, se hace referencia a miembros de la clase derivada, es un error pues esos miembros no existen en el objeto de la clase base.

Ejemplo:

```
1 // POINT.H
2 // clase Point
3 #ifndef POINT_H_
```

```
4  #define POINT_H_
5  using namespace std;
6
7  class Point {
8     friend ostream &operator<<(ostream &, const Point &);
9          public:
10    Point(float = 0, float = 0);
11    void setPoint(float, float);
12    float getX() const { return x; }
13    float getY() const { return y; }
14         protected:
15    float x, y;
16 };
17
18 #endif /*POINT_H_*/
19
20
21 // POINT.CPP
22 #include <iostream>
23 #include "point.h"
24
25 Point::Point(float a, float b){
26    x = a;
27    y = b;
28 }
29
30 void Point::setPoint(float a, float b){
31    x = a;
32    y = b;
33 }
34
35 ostream &operator<<(ostream &output, const Point &p){
36    output << '[' << p.x << ", " << p.y << ']';
37
38    return output;
39 }
40
41
42 // CIRCLE.H
43 // clase Circle
44 #ifndef CIRCLE_H
45 #define CIRCLE_H
```

```
46
47  #include <iostream>
48  #include <iomanip>
49  #include "point.h"
50
51  class Circle : public Point {  // Circle hereda de Point
52            friend ostream &operator<<(ostream &, const Circle &);
53          public:
54             Circle(float r = 0.0, float x = 0, float y = 0);
55
56             void setRadius(float);
57             float getRadius() const;
58             float area() const;
59          protected:
60             float radius;
61  };
62  #endif /*CIRCLE_H_*/
63
64
65  // CIRCLE.CPP
66  #include "circle.h"
67
68  Circle::Circle(float r, float a, float b)
69     : Point(a, b)   // llama al constructor de la clase base
70  { radius = r; }
71
72  void Circle::setRadius(float r) { radius = r; }
73
74  float Circle::getRadius() const { return radius; }
75
76  float Circle::area() const
77  { return 3.14159 * radius * radius; }
78
79  // salida en el formato:
80  // Center = [x, y]; Radius = #.##
81  ostream &operator<<(ostream &output, const Circle &c){
82     output << "Center = [" << c.x << ", " << c.y
83            << "]; Radius = " << setprecision(2) << c.radius;
84
85     return output;
86  }
87
```

```
88  //Prueba.cpp
89  // Probando apuntadores a clase base a apuntadores a clase derivada
90  #include <iostream>
91  #include <iomanip.h>
92  #include "point.h"
93  #include "circle.h"
94
95  int main(){
96          Point *pointPtr, p(3.5, 5.3);
97          Circle *circlePtr, c(2.7, 1.2, 8.9);
98
99          cout << "Point p: " << p << "\nCircle c: " << c << endl;
100
101         // Maneja a un Circle como un Circle
102         pointPtr = &c;   // asigna la direccion de Circle a pointPtr
103         circlePtr = (Circle *) pointPtr;  // mascara de base a derivada
104         cout << "\nArea de c (via circlePtr): "
105         << circlePtr->area() << endl;
106
107         // Es riesgoso manejar un Point como un Circle
108         // getRadius() regresa basura
109         pointPtr = &p;   // asigna direccion de Point a pointPtr
110         circlePtr = (Circle *) pointPtr;  // mascara de base a derivada
111         cout << "\nRadio de objeto apuntado por circlePtr: "
112         << circlePtr->getRadius() << endl;
113
114         return 0;
115 }
```

Listing 61. Ejemplo de objetos de clase derivada como clase base en C++.

13.2.4. Constructores de clase base en C++

El constructor de la clase base puede ser llamado desde la clase derivada, para inicializar los atributos heredados. La sintaxis es igual que el inicializador de objetos componentes.

Los constructores y operadores de asignación de la clase base no son heredados por las clases derivadas. Pero pueden ser llamados por los de la clase derivada.

Un constructor de la clase derivada llama primero al constructor de la clase base. Si se omite el constructor de la clase derivada, el constructor por omisión de la clase derivada llamará al constructor de la clase base.

Los destructores son llamados en orden inverso a las llamadas del constructor: un destructor de una clase derivada será llamado antes que el de su clase base.

Sintaxis

```
<clase>::<constructor>(<lista de argumentos>) :
  <contructor de clase base>(<lista de argumentos sin el tipo>)
```

Ejemplo:

```
1  // POINT.H
2  #ifndef POINT_H
3  #define POINT_H
4
5  class Point {
6  public:
7     Point(float = 0.0, float = 0.0);
8     ~Point();
9  protected:
10    float x, y;
11 };
12
13 #endif /*POINT_H_*/
14
15 // POINT.CPP
16 #include <iostream>
17 #include "point.h"
18 using namespace std;
19 Point::Point(float a, float b){
20    x = a;
21    y = b;
22
23    cout << "Constructor Point: "
24         << '[' << x << ", " << y << ']' << endl;
25 }
26
27 Point::~Point(){
28    cout << "Destructor Point:  "
29         << '[' << x << ", " << y << ']' << endl;
30 }
31
32 // CIRCLE.H
```

```
33 #ifndef CIRCLE_H
34 #define CIRCLE_H
35
36 #include "point.h"
37
38 class Circle : public Point {
39 public:
40    Circle(float r = 0.0, float x = 0, float y = 0);
41    ~Circle();
42 private:
43    float radius;
44 };
45
46 #endif /*CIRCLE_H_*/
47
48 // CIRCLE.CPP
49 #include "circle.h"
50
51 using namespace std;
52
53 Circle::Circle(float r, float a, float b)
54    : Point(a, b)   // llamada al constructor de clase base
55 {
56    radius = r;
57
58    cout << "Constructor Circle: radio es "
59         << radius << " [" << a << ", " << b << ']' << endl;
60 }
61
62 Circle::~Circle(){
63    cout << "Destructor Circle:   radio es "
64         << radius << " [" << x << ", " << y << ']' << endl;
65 }
66
67 // Main.CPP
68 #include <iostream.h>
69 #include "point.h"
70 #include "circle.h"
71
72 int main(){
73    // Muestra llamada a constructor y destructor de Point
74    {
```

```
75        Point p(1.1, 2.2);
76    }
77
78    cout << endl;
79    Circle circle1(4.5, 7.2, 2.9);
80    cout << endl;
81    Circle circle2(10, 5, 5);
82    cout << endl;
83    return 0;
84 }
```

Listing 62. Ejemplo constructor de clase base en C++.

13.2.5. Redefinición de métodos en C++

Algunas veces, los métodos heredados no cumplen completamente la función que quisiéramos que realicen en las clases derivadas. Es posible en C++ **redefinir** un método de la clase base en la clase derivada. Cuando se hace referencia al nombre del método, se ejecuta la versión de la clase en donde fue redefinida.

Es posible sin embargo, utilizar el método original de la clase base por medio del operador de resolución de alcance.

Se sugiere redefinir métodos que no vayan a ser empleados en la clase derivada, inclusive sin código para inhibir cualquier acción que no nos interese[1].

Concepto
La redefinición de métodos **no** es una sobrecarga porque se definen exactamente con la misma firma.

Ejemplo:

```
1  // EMPLOY.H
2  #ifndef EMPLOY_H
3  #define EMPLOY_H
4
5  using namespace std;
6
7  class Employee {
```

[1]n teoría esto no debería ser necesario anular operaciones si nos apegamos a la regla del 100 % (**de conformidad con la definición**)

```
8         public:
9            Employee(const char*, const char*);
10           void print() const;
11           ~Employee();
12        private:
13           char *firstName;
14           char *lastName;
15 };
16 #endif /*EMPLOY_H_*/
17
18 // EMPLOY.CPP
19 #include <string.h>
20 #include <iostream>
21 #include <assert.h>
22 #include "employ.h"
23
24 Employee::Employee(const char *first, const char *last){
25    firstName = new char[ strlen(first) + 1 ];
26    assert(firstName != 0);     strcpy(firstName, first);
27
28    lastName = new char[ strlen(last) + 1 ];
29    assert(lastName != 0);
30    strcpy(lastName, last);
31 }
32
33 void Employee::print() const
34 { cout << firstName << ' ' << lastName; }
35
36 Employee::~Employee(){
37    delete [] firstName;
38    delete [] lastName;
39 }
40
41 // HOURLY.H
42 #ifndef HOURLY_H
43 #define HOURLY_H
44
45 #include "employ.h"
46
47 class HourlyWorker : public Employee {
48        public:
49           HourlyWorker(const char*, const char*, float, float);
```

```
50            float getPay() const;
51          void print() const;
52        private:
53            float wage;
54            float hours;
55  };
56
57  #endif /*HOURLY_H_*/
58
59  // HOURLY_B.CPP
60  #include <iostream>
61  #include <iomanip.h>
62  #include "hourly.h"
63
64  HourlyWorker::HourlyWorker(const char *first, const char *last, float initHours,
65   float initWage) : Employee(first, last) {
66     hours = initHours;
67     wage = initWage;
68  }
69
70  float HourlyWorker::getPay() const { return wage * hours; }
71
72  void HourlyWorker::print() const {
73     cout << "HourlyWorker::print()\n\n";
74
75     Employee::print();   // llama a función de clase base
76
77     cout << " es un trabajador por hora con sueldo de"
78          << " $" << setprecision(2) << getPay() << endl;
79  }
80
81  // main.CPP
82  #include <iostream>
83  #include "hourly.h"
84
85  int main(){
86     HourlyWorker h("Bob", "Smith", 40.0, 7.50);
87     h.print();
88     return 0;
89  }
```

Listing 63. Ejemplo de refefinición de métodos en C++.

13.2.6. Herencia múltiple en C++

Es posible que una clase requiere recibir miembros de más de una clase. Esto es posible haciendo uso de **herencia múltiple**.

Concepto
Herencia múltiple es la capacidad de una clase derivada de heredar miembros de varias clases base.

Sintaxis

`class <nombre clase derivada> : <clase base 1> , <clase base 2>,`
`  ...<clase base n> {`
`  ...`
`};`

Ejemplo:

```
1  class A{
2          public:
3          int i;
4          void a(){}
5  };
6
7  class B{
8          public:
9          int j;
10         void b(){}
11 };
12
13 class C{
14         public:
15         int k;
16         void c(){}
17 };
18
19 class D: public A, public B, public C {
20         public:
```

```
21          int l;
22          void d(){}
23  };
24
25  int main() {
26          D var1;
27
28          var1.a();
29          var1.b();
30          var1.c();
31          var1.d();
32          return 0;
33  }
```

Listing 64. Ejemplo de herencia múltiple con C++.

Ejemplo:

```
1   // BASE1.H
2   #ifndef BASE1_H
3   #define BASE1_H
4
5   class Base1 {
6           public:
7              Base1(int x) { value = x; }
8              int getData() const { return value; }
9           protected:
10          int value;
11  };
12
13  #endif /*BASE1_H_*/
14
15  // BASE2.H
16  #ifndef BASE2_H
17  #define BASE2_H
18
19  class Base2 {
20          public:
21             Base2(char c) { letter = c; }
22             char getData() const { return letter; }
23          protected:
24             char letter;
```

```
};

#endif /*BASE2_H_*/

// DERIVED.H
#ifndef DERIVED_H
#define DERIVED_H

// herencia múltiple
class Derived : public Base1, public Base2 {
           friend ostream &operator<<(ostream &, const Derived &);
        public:
           Derived(int, char, float);
           float getReal() const;
        private:
           float real;
};

#endif /*DERIVED_H_*/

// DERIVED.CPP
#include <iostream.h>
#include "base1.h"
#include "base2.h"
#include "derived.h"

Derived::Derived(int i, char c, float f): Base1(i), Base2(c) {
        real = f;
}

float Derived::getReal() const {
        return real;
}

ostream &operator<<(ostream &output, const Derived &d) {
   output << "     Entero: " << d.value
          << "\n   Caracter: " << d.letter
          << "\nNúmero real: " << d.real;

   return output;
}

```

```
67  // main.CPP
68  #include <iostream.h>
69  #include "base1.h"
70  #include "base2.h"
71  #include "derived.h"
72
73  int main(){
74          Base1 b1(10), *base1Ptr;
75          Base2 b2('Z'), *base2Ptr;
76          Derived d(7, 'A', 3.5);
77
78             cout << "Objeto b1 contiene entero "
79                  << b1.getData()
80                  << "\nObjeto b2 contiene caracter "
81                  << b2.getData()
82                  << "\nObjeto d contiene:\n" << d;
83          cout << "\n\nmiembros de clase derivada pueden ser"
84                  << " accesados individualmente:"
85                  << "\n  Entero: " << d.Base1::getData()
86                  << "\n  Caracter: " << d.Base2::getData()
87                  << "\n  Número real: " << d.getReal() << "\n\n";
88          // Probar:  cout<<d.getData();  Es un error?
89             cout << "Miembros derivados pueden ser tratados como "
90                  << "objetos de su clase base:\n";
91
92             base1Ptr = &d;
93             cout << "base1Ptr->getData() "
94                  << base1Ptr->getData();
95
96             base2Ptr = &d;
97             cout << "\nbase2Ptr->getData() "
98          << base2Ptr->getData() << endl;
99             return 0;
100 }
```

Listing 65. Otro ejemplo de herencia múltiple en C++.

Aquí se tiene un problema de **ambigüedad** al heredar dos métodos con el mismo nombre de clases diferentes. Se resuelve poniendo antes del nombre del miembro el nombre de la clase: $objeto.<clase::>miembro$. El nombre del objeto es necesario, pues no se esta haciendo referencia a un miembro estático.

Ambigüedades

En el ejemplo anterior se vio un caso de ambigüedad al heredar de clases distintas un miembro con el mismo nombre. Normalmente se deben tratar de evitar esos casos, pues vuelven confusa nuestra jerarquía de herencia.

Existen otro casos donde es posible que se de la ambigüedad.

Ejemplo:

```
1  //ejemplo de ambigüedad en la herencia
2  class B{
3  public:
4          int b;
5  };
6
7  class D: public B, public B { //error
8  };
9
10 void f( D *p) {
11         p->b=0;  //ambiguo
12 }
13
14 int main(){
15         D obj;
16         f(&obj);
17         return 0;
18 }
```

Listing 66. Ejemplo de ambigüedad en herencia múltiple con C++.

El código anterior tiene un error en la definición de herencia múltiple, ya que no es posible heredar más de una vez una misma clase de manera directa. Sin embargo, si es posible heredar las características de una clase más de una vez indirectamente:

Ejemplo:

```
1  //ejemplo de ambigüedad en la herencia
2  #include <iostream>
3  using namespace std;
4
5  class A{
6          public:
7          int next;
8  };
9
```

```
10 class B: public A{
11 };
12
13 class C: public A{
14 };
15
16 class D: public B, public C {
17         int g();
18 };
19
20 int D::g(){
21         //next=0;  Error: asignación ambigua
22         return B::next == C::next;
23 }
24
25 class E: public D{
26          public:
27         int x;
28         int getx(){
29                 return x;
30         }
31 };
32
33 int main(){
34     D obj;
35         E obje;
36
37         obj.B::next=10;
38         obj.C::next=20;
39 //  obj.A::next=11;          Error: acceso ambiguo
40 //         obj.next=22;                 Error: acceso ambiguo
41         cout<<"next de B: "<<obj.B::next<<endl;
42         cout<<"next de C: "<<obj.C::next<<endl;
43
44         obje.x=0;
45         obje.B::next=11;
46         obje.C::next=22;
47         cout<<"obje next de B: "<<obje.B::next<<endl;
48         cout<<"obje next de C: "<<obje.C::next<<endl;
49         return 0;
50 }
```

Listing 67. Ejemplo 2 de ambigüedad en herencia múltiple con C++.

Este programa hace que las instancias de la clase D tengan objetos de clase base duplicados y provoca los accesos ambiguos. Este problema se resuelve con **herencia virtual**.

Concepto
Herencia de clase base virtual: Si se especifica a una clase base como virtual, solamente un objeto de la clase base existirá en la clase derivada.

Para el ejemplo anterior, las clases B y C deben declarar a la clase A como clase base virtual:

Sintaxis

```
class B: virtual public A {...}

class C: virtual public A {...}
```

El acceso entonces a los miembros puede hacerse usando una de las clases de las cuales heredo el miembro:

```
1 obj.B::next=10;
2 obj.C::next=20;
```

O simplemente accediéndolo como un miembro no ambiguo:ejercicio: probar los ejemplo y modificar la definición a clases base virtuales.

```
obj.next=22;
```

En cualquier caso se tiene solo una copia del miembro, por lo que cualquier modificación del atributo *next* es sobre una única copia del mismo.

Constructores en herencia múltiple

Si hay constructores con argumentos, es posible que sea necesario llamarlos desde el constructor de la clase derivada. Para ejecutar los constructores de las clases base, pasando los argumentos, es necesario especificarlos después de la declaración de la función de construcción de la clase derivada, separados por coma.

Sintaxis

```
<Constructor clase derivada>(<argumentos> : <base1> (<argumentos>),
   <base2> (<argumentos>), ... , <basen> (<argumentos>) {
        ...
}
```

Donde como en la herencia simple, el nombre base corresponde al nombre de la clase, o en este caso, clases base.

El orden de llamada a constructores de las clases base se puede alterar a conveniencia. Una **excepción** a considerar es cuando se resuelve ambigüedad de una clase base pues en ese caso el constructor de la clase base ambigua únicamente se ejecuta una vez. Si no es especificado por el programador, se ejecuta el constructor sin parámetros de la clase base ambigua. Si se requiere pasar parámetros, se debe especificar la llamada antes de las llamadas a constructores de clase base directas. Este es el único caso en que es posible llamar a un constructor de una clase que no es un ancestro directo[7].

13.3. Herencia: implementación en Java

La clase de la cual se toman sus características se conoce como **superclase**; mientras que la clase que ha sido creada a partir de la clase base se conoce como **subclase**.

La herencia en Java difiere ligeramente de la sintaxis de implementación de herencia en C++.

Sintaxis

```
class <subclase> extends  <superclase> {
        //cuerpo subclase
}
```

Ejemplo:

Una clase vehículo que describe a todos aquellos objetos vehículos que viajan en carreteras. Puede describirse a partir del número de ruedas y de pasajeros. De la definición de vehículos podemos definir objetos más específicos (especializados). Por ejemplo la clase camión.

```
1  //ejemplo de herencia
2  class Vehiculo{
3          private int ruedas;
4          private int pasajeros;
5
6          public void setRuedas(int num){
7                  ruedas=num;
8          }
9
10         public int getRuedas(){
11                 return ruedas;
12         }
13
14         public void setPasajeros(int num){
15                 pasajeros=num;
16         }
17
18         public int getPasajeros(){
19                 return pasajeros;
20         }
```

```
21
22  }
23
24  //clase Camion con herencia  de Vehiculo
25  public class Camion extends Vehiculo {
26          private int carga;
27
28          public void setCarga(int num){
29                  carga=num;
30          }
31
32          public int getCarga(){
33                  return carga;
34          }
35
36          public void muestra(){
37                  // uso de métodos heredados
38                  System.out.println("Ruedas: " + getRuedas());
39                  System.out.println("Pasajeros: " + getPasajeros());
40                  // método de la clase Camion
41                  System.out.println("Capacidad de carga: " + getCarga());
42          }
43
44          public static void main(String argvs[]){
45                  Camion ford= new Camion();
46                  ford.setRuedas(6);
47                  ford.setPasajeros(3);
48                  ford.setCarga(3200);
49                  ford.muestra();
50          }
51
52  }
```

Listing 68. Ejemplo de herencia en Java.

En el programa anterior se puede apreciar claramente como una clase Vehículo hereda sus características a la subclase *Camion*, pudiendo este último aprovechar recursos que no declara en su definición.

Tema sugerido
BlueJ en apéndice ¿A?.

13.3.1. Clase *Object*

En Java toda clase que se define tiene herencia implícita de una clase llamada object. En caso de que la clase que crea el programador defina una herencia explícita a una clase, hereda las características de la clase Object de manera indirecta[2].

Ver clase Object en: https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

13.3.2. Control de acceso a miembros en Java

Existen tres palabras reservadas para el control de acceso a los miembros de una clase: *public*, *private* y *protected*. Estas sirven para proteger los miembros de la clase en diferentes formas.

El control de acceso, como ya se vio anteriormente, se aplica a los métodos, atributos, constantes y tipos anidados que son miembros de la clase.

Resumen de tipos de acceso:

- *private*. Un miembro privado únicamente puede ser utilizado por los métodos miembro de la clase donde fue declarado. Un miembro privado no es posible que sea manejado ni siquiera en sus subclases.
- *protected*. Un miembro protegido puede ser utilizado únicamente por los métodos miembro de la clase donde fue declarado, por los métodos miembro de las clases derivadas ó clases que pertenecen al mismo paquete. El acceso protegido es como un nivel intermedio entre el acceso privado y público.
- *public*. Un miembro público puede ser utilizado por cualquier método. Este es visible en cualquier lugar que la clase sea visible

Ejemplo:

```
1  //ejemplo de control de acceso
2
3  class Acceso{
4          protected int b;
5          protected int f2() {
6                  return b;
7          }
8
9          private int c;
10         private int f3() {
11                 return c;
```

[2]En este caso hay que considerar que las características de la clase *Object* pudieron haber sido modificadas a través de la jerarquía de herencia.

```
12          }
13
14          public int d, f;
15          public int f4(){
16                  return  d;
17          }
18
19  }
20
21  public class EjemploAcceso {
22
23          public static void main(String argvs[]){
24                  Acceso obj= new Acceso();
25
26                  obj.f2(); //es válido, ya que por omisión
27                  obj.b=2; //las dos clases están en el mismo paquete
28
29                  obj.c=3; //error es un atributo privado
30                  obj.f3(); //error es un método privado
31
32                  obj.d=5;
33                  obj.f4();
34          }
35  }
```

Listing 69. Ejemplo de control de acceso en Java.

El ejemplo anterior genera errores de compilación al tratar de acceder desde otra clase a miembros privados. Sin embargo, los miembros protegidos si pueden ser accedidos porque están considerados implícitamente dentro del mismo paquete.

13.3.3. Control de acceso de clase public en Java

Este controlador de acceso *public*, opera a nivel de la clase para que esta se vuelva visible y accesible desde cualquier lugar, lo que permitiría a cualquier otra clase hacer uso de los miembros de la clase pública.

Sintaxis

```
public class TodosMeVen {
// definición de la clase
}
```

La omisión de este calificador limita el acceso a la clase para que solo sea utilizada por clases pertenecientes al mismo paquete.

Además, se ha mencionado que en un archivo fuente únicamente puede existir una clase pública, la cual debe coincidir con el nombre del archivo. La intención es que cada clase que tenga un objetivo importante debería ir en un archivo independiente, pudiendo contener otras clases no públicas que le ayuden a llevar a cabo su tarea.

13.3.4. Constructores de superclase

Los constructores no se heredan a las subclases. El constructor de la superclase puede ser llamado desde la clase derivada, para inicializar los atributos heredados y no tener que volver a introducir código de inicialización ya escrito en la superclase.

La llamada explícita al constructor de la superclase se realiza mediante la referencia **super** seguida de los argumentos –si los hubiera- del constructor de la clase base. La llamada a este constructor debe ser hecha en la primera línea del constructor de la subclase. Si no se introduce así, el constructor de la clase derivadada llamará automáticamente al constructor por omisión (sin parámetros) de la superclase.

En el ejemplo siguiente podrá apreciarse esta llamada al constructor de la superclase.

13.3.5. Manejo de objetos de subclase como objetos de superclase en Java

Un objeto de una clase derivada, puede ser manejado como un objeto de su superclase. Sin embargo, un objeto de la clase base no es posible tratarlo como un objeto de clase derivada.

Un objeto de una subclase puede ser asignado a una variable de referencia de su superclase sin necesidad de indicar una conversión explícita mediante enmascaramiento. Cuando si se necesita utilizar enmascaramiento es para asignar de vuelta un objeto que aunque sea de una clase derivada, este referenciado por una variable de clase base. Esta conversión explícita es verificada por la máquina virtual, y si no corresponde el tipo real del objeto, no se podrá hacer la asignación y se generará una excepción en tiempo de ejecución.

Ejemplo:

```
1 Superclase s= new superclase(), aptSuper;
2 Subclase sub= new subclase(), aptSub;
```

```
3
4  //válido
5  aptSuper = sub;
6
7  aptSub = (Subclase) aptSuper;
8
9  //inválido
10 aptSub= (Subclase) s;
```

Podemos ver en el ejemplo anterior, que un objeto puede "navegar" en la jerarquía de clases hacia sus superclases, pero no puede ir a una de sus subclases, ni utilizando el enmascaramiento. Esto se hace por seguridad, ya que la subclase seguramente contendrá un mayor número de elementos que una instancia de superclase y estos no podrían ser utilizados porque causarían una inconsistencia.

Por último, es importante señalar que mientras un objeto de clase derivada este referenciado como un objeto de superclase, deberá ser tratado como si el objeto fuera únicamente de la superclase; por lo que no podrá en ese momento tener referencias a atributos o métodos definidos en la clase derivada.

Un código completo se muestra a continuación. Ejemplo:

```
1  // definición de clase point
2  public class Point {
3     protected double x, y; // coordenadas del punto
4
5     // constructor
6     public Point( double a, double b ) {
7             setPoint( a, b );
8     }
9
10    // asigna a x,y las coordenadas del punto
11    public void setPoint( double a, double b ) {
12       x = a;
13       y = b;
14    }
15
16    // obtiene coordenada x
17    public double getX() {
18            return x;
19    }
20
21    // obtiene coordenada y
22    public double getY() {
```

```
23                  return y;
24          }
25
26      // convierte información a cadena
27          public String toString(){
28                  return "[" + x + ", " + y + "]";
29          }
30  }
31  // Definición de clase círculo
32  public class Circle extends Point {  // Hereda de Point
33     protected double radius;
34
35     // constructor sin argumentos
36     public Circle()   {
37        super( 0, 0 );  // llamada a constructor de clase base
38        setRadius( 0 );
39     }
40
41     // Constructor
42     public Circle( double r, double a, double b )   {
43        super( a, b );  // llamada a constructor de clase base
44        setRadius( r );
45     }
46
47     // Asigna radio del círculo
48     public void setRadius( double r )
49        { radius = ( r >= 0.0 ? r : 0.0 ); }
50
51     // Obtiene radio del círculo
52     public double getRadius() { return radius; }
53
54     // Cálculo área del círculo
55     public double area() { return 3.14159 * radius * radius; }
56
57     // Convierte información en cadena
58     public String toString() {
59        return "Centro = " + "[" + x + ", " + y + "]" +
60               "; Radio = " + radius;
61     }
62  }
63  // Clase de Prueba de las clases Point y Circle
64  public class Prueba {
```

```
65
66    public static void main( String argvs[] )  {
67       Point pointRef, p;
68       Circle circleRef, c;
69
70       p = new Point( 3.5, 5.3 );
71       c = new Circle( 2.7, 1.2, 8.9 );
72
73       System.out.println( "Punto p: " + p.toString() );
74       System.out.println( "Circulo c: " + c.toString());
75
76       // Tratamiento del círculo como instancia de punto
77       pointRef = c;   // asigna círculo c a pointRef
78       // en realidad Java lo reconoce dinámicamente como objeto Circle
79       System.out.println( "Circulo c (via pointRef): " + pointRef.toString());
80
81       // Manejar a un círculo como círculo (obteniéndolo de una referencia de punto)
82       pointRef = c;   // asigna círculo c a pointref. Se repite la operación por claridad
83       circleRef = (Circle) pointRef;  // enmascaramiento de superclase a subclase
84       System.out.println( "Circulo c (via circleRef): " + circleRef.toString());
85       System.out.println( "Area de c (via circleRef): " + circleRef.area());
86
87       // intento de referenciar a un objeto point
88       // desde una referencia de Circle (genera una excepcion)
89       circleRef = (Circle) p;
90    }
91 }
```

Listing 70. Ejemplo de objetos de sub clase como de superclase en Java.

13.3.6. Redefinición de métodos

Algunas veces, los métodos heredados no cumplen completamente la función que queremos que realicen en la subclase. Por esta razón, Java permite **redefinir** un método de la clase base en la clase derivada. Cuando se hace referencia al nombre del método, se ejecuta la versión de la clase en donde fue redefinida. Es posible sin embargo, utilizar el método de la clase base por medio del la referencia *super*.

De hecho, se sugiere redefinir métodos que no vayan a ser empleados en la clase derivada, inclusive sin código para inhibir cualquier acción que no nos interese en la clase derivada.

Ejemplo:

```
1  //clase empleado
2  public class Empleado {
3
4          private String firstName, lastName;
5
6          public Empleado(String first, String last){
7             firstName = new String(first);
8             lastName = new String(last);
9          }
10
11         public void print() {
12                 System.out.println( firstName + ' ' + lastName);
13         }
14
15 }
16
17 //clase TrabajadorporHora
18 public class TrabajadorPorHora extends Empleado {
19         private float wage, hours;
20         public TrabajadorPorHora(String first, String last, float initHours, float initWage) {
21                 super(first, last);
22                 hours = initHours;
23                 wage = initWage;
24         }
25
26         public float getPay() {
27                 return wage * hours;
28         }
29
30         public void print() {
31                 System.out.println("Metodo print de Trabajador por hora");
32
33                 super.print();   // llama a función de clase base
34
35                    System.out.println(" es un trabajador por hora con sueldo de" +
36                         " $" + getPay());
37         }
38 }
39
40 // clase de prueba EmpTest
41 public class EmpTest {
```

```
42
43          public static void main(String argvs[]) {
44                  Empleado e= new Empleado ("nombre", "apellido");
45                  TrabajadorPorHora h;
46
47                  h=new TrabajadorPorHora ("Juanito", "Perez", 40.0f, 7.50f);
48                  e.print();
49                  h.print();
50          }
51  }
```

Listing 71. Ejemplo de redefinición de métodos en Java.

13.3.7. Calificador *final*

Es posible que tengamos la necesidad de que cierta parte de una clase no pueda ser modificada en futuras extensiones de la jerarquía de herencia. Para esto es posible utilizar el calificador *final*.

Si un método se especifica en una clase X como *final*:

Sintaxis
`<acceso> final <tipo> nombreMétodo( <parámetros>)`

Se esta diciendo que el método no podrá ser redefinido en las subclases de X.

Aunque se omita este calificador, si se trata de un método de clase (estático) o privado, se considera final y no podrá ser redefinido.

Por otro lado, es posible que no queramos dejar la posibilidad de extender una clase, para lo que se utiliza el calificador *final* a nivel de clase:

Sintaxis

```
<acceso> final class nombreClase {

        //definición de la clase

}
```

De esta forma, la clase no permite generar subclases a partir de ella. De hecho, el API de Java incluye muchas clases *final*, por ejemplo la clase java.lang.String no puede ser especializada.

13.4. Interfaces en Java

Java únicamente cuenta con manejo de herencia simple, y la razón que se ofrece es que la herencia múltiple presenta algunos problemas de ambigüedad que complica el entendimiento del programa, sin que este tipo de herencia justifique las ventajas obtenidas de su uso.

Sin embargo, es posible que se necesiten recibir características de más de un origen. Java soluciona esto mediante el uso de interfaces, que son una forma para declarar tipos especiales de clase que, aunque con ciertas limitaciones, no ofrecen las complicaciones de la herencia múltiple.

Una interfaz tiene un formato muy similar a una clase, sus principales características:

- Una interfaz proporciona los nombres de los métodos, pero no sus implementaciones[3].
- Una clase puede implementar varias interfaces, aunque solo pueda heredar una clase.
- No es posible crear instancias de una interfaz.
- La clase que implementa la interfaz debe escribir el código de todos los métodos, de otra forma no se podrá generar instancias de esa clase.

El formato general para la declaración de una interfaz es el siguiente:

Sintaxis

```
[public] interface <nombreInterfaz> {
        //descripción de miembros
        //los métodos no incluyen código:
        <acceso> <tipo> <nombreMetodo> ( <parámetros> ) ;
}
```

El cuerpo de la interfaz generalmente es una lista de prototipos de métodos, pero puede contener atributos si se requiere[4].

Una clase implementa una interfaz a través de la palabra reservada *implements* después de la especificación de la herencia (si la hubiera) :

[3] En esta caso si se considera la declaración de prototipos.
[4] El parámetro debe incluir el nombre, el cual no es obligatorio que coincida en la implementación.

Sintaxis

```
class <SubClase> extends <Superclase> implements   <nombreInterfaz> {
        //definición de la clase
        //debe incluirse la definición de los métodos de la interfaz
        //con la implementación del código de dichos métodos.
}
```

Además, una interfaz puede ser extendida de la misma forma que una clase, aprovechando las interfaces previamente definidas, mediante el uso de la cláusula *extends*.

Sintaxis

```
[public] interface <nombreInterfaz> extends <InterfazBase> {

        //descripción de miembros

}
```

De forma distinta a la jerarquía de clases, donde se tiene una jerarquía lineal que parte siempre de una clase simple *Object*, una clase soporta herencia múltiple de interfaces, resultando en una jerarquía con múltiples raíces de diferentes interfaces.

Ejemplo:

```
1  //interfaz
2  interface IStack {
3      void push(Object item);
4      Object pop();
5  }
6
7  //clase implementa la interfaz
8  class StackImpl implements IStack {
9      protected Object[] stackArray;
10     protected int tos;
11
12     public StackImpl(int capacity) {
```

```
13          stackArray = new Object[capacity];
14          tos = -1;
15      }
16
17          //implementa el método definido en la interfaz
18      public void push(Object item)
19          { stackArray[++tos] = item; }
20
21          //implementa el método definido en la interfaz
22      public Object pop() {
23          Object objRef = stackArray[tos];
24          stackArray[tos] = null;
25          tos--;
26          return objRef;
27      }
28
29      public Object peek() { return stackArray[tos];}
30  }
31
32  // extendiendo una interfaz
33  interface ISafeStack extends IStack {
34      boolean isEmpty();
35      boolean isFull();
36  }
37
38
39  //esta clase hereda la implementación de la pila StackImpl
40  // e implementa la nueva interfaz extendida ISafeStack
41  class SafeStackImpl extends StackImpl implements ISafeStack {
42
43      public SafeStackImpl(int capacity) { super(capacity); }
44
45      //implementa los métodos de la interfaz
46      public boolean isEmpty() { return tos < 0; }
47      public boolean isFull() { return tos >= stackArray.length;
48           }
49  }
50
51  public class StackUser {
52
53      public static void main(String args[]) {
54          SafeStackImpl safeStackRef = new SafeStackImpl(10);
```

```
55          StackImpl stackRef = safeStackRef;
56          ISafeStack isafeStackRef = safeStackRef;
57          IStack istackRef = safeStackRef;
58          Object objRef = safeStackRef;
59
60          safeStackRef.push("Dolar");
61          stackRef.push("Peso");
62
63                  // tipo de dato simple es convertido a Integer:
64          stackRef.push(1);
65          System.out.println(stackRef.peek().getClass());
66
67          System.out.println(isafeStackRef.pop());
68          System.out.println(istackRef.pop());
69
70          System.out.println(objRef.getClass());
71      }
72  }
```

Listing 72. Ejemplo de interfaz en Java.

Por otro lado, una interfaz también puede ser utilizada para definir nuevos tipos. Una interfaz así o una clase que implementa a una interfaz de este estilo es conocida como **Supertipo**.

Es importante resaltar tres diferencias en las relaciones de herencia y como esta funciona entre clases e interfaces:

1. **Implementación lineal de jerarquía de herencia entre clases**: una clase extiende a otra clase.

2. **Jerarquía de herencia múltiple entre interfaces**: una interfaz extiende otras interfaces.

3. **Jerarquía de herencia múltiple entre interfaces y clases**: una clase implementa interfaces.

Java 8 permite el uso de métodos implementados en interfaces. Estos pueden ser métodos estáticos o un tipo especial de método llamado *Default/Defender*[5].

[5] https://www.techempower.com/blog/2013/03/26/everything-about-java-8/, https://stackoverflow.com/questions/18286235/what-is-the-default-implementation-of-method-defined-in-an-interface

13.5. Herencia: Implementación en Python

El manejo de herencia en Python se hace de la siguiente manera[6]:

Sintaxis

```
class <ClaseDerivada>(<ClaseBase>):
    <instrucciones>
```

Si la clase se encuentra en otro módulo:

```
class <ClaseDerivada>(<Módulo>.<ClaseBase>):
    <instrucciones>
```

Todas las clases en Python 3 derivan de la clase *object*. Los métodos de la clase base pueden ser redefinidos. Un objeto ejecutando métodos puede ser que llame a métodos de la clase base o métodos redefinidos en la jerarquía de herencia, de manera similar a los métodos virtuales en C++.

Si fuera necesario llamar a un método de clase base que se haya redefinido en una clase derivada puede hacerse.

Sintaxis

```
<ClaseBase>.<método>(self, <argumentos>)
```

Dos métodos útiles en herencia son $isinstance(< objeto >, < clase >)$, el cual devuelve verdadero si un objeto pertenece a una clase o subclase especificada (directa, indirecta o virtual); de forma similar, $issubclass(< clase >, < infoclase >)$ regresa verdadero si $< clase >$ es una subclase (directa, indirecta o virtual) de $< infoclase >$[7].

Ejemplo:

```
1 class Persona:
2         def __init__(self, nombre, apellido):
3                 self.nombre = nombre
4                 self.apellido = apellido
5
6         def nombreCompleto(self):
```

[6] Multiple Inheritance, https://docs.python.org/3/tutorial/classes.html#inheritance

[7] Aquí, una clase es considerada subclase de si misma.

```
7                 return self.nombre + " " + self.apellido
8
9   class Empleado(Persona):
10
11          def __init__(self, nombre, apellido, staffnum):
12                  Persona.__init__(self,nombre, apellido)
13                  self.staffnum = staffnum
14
15          def getEmpleado(self):
16                  return self.nombreCompleto() + ", " + self.staffnum
17
18  # script de ejecución inicial
19  x = Persona("Una", "Persona")
20  y = Empleado("Un", "Empleado", "121212")
21
22  print(x.nombreCompleto())
23  print(y.getEmpleado())
```

Listing 73. Ejemplo de herencia en Python.

13.5.1. Inicializadores de superclase

En Python también es posible llamar al método inicializador de la superclase usando *super* seguido de la lista de argumentos.

Ejemplo:

```
1   class Persona:
2           def __init__(self, nombre, edad, sexo):
3                   self.nombre = nombre
4                   self.edad = edad
5                   self.sexo = sexo
6
7           def __str__(self):
8                   return self.nombre + " " + str(self.edad)+ " "+ self.sexo
9
10  class Estudiante(Persona):
11          def __init__(self, nombre, edad, sexo, matr, horas):
12                  # en Python 3 no se necesita pasar self al llamar
13                  # a super() ni la sintaxis super(subclase, self)
14                  # de Python 2
15                  super().__init__(nombre, edad, sexo)
16                  self.matricula = matr
```

```
17                 self.horas = horas
18
19 #creando dos objetos
20 a = Persona("Ironman", 37, "m")
21 b = Estudiante("Spiderman", 36, "m", "000-13-5031", 24)
22
23 print(a)
24 print(b)
```

Listing 74. Ejemplo con inicializadores de superclase en Python.

13.5.2. Herencia Múltiple (*mixins*) en Python

Python soporta una forma limitada de herencia múltiple. Algunos autores se refieren a la implementación de Python de herencia múltiple como mixins[8]. La definición de una clase con herencia múltiple tiene la siguiente forma:

Sintaxis

```
class <ClaseDerivada>(<ClaseBase1>, <ClaseBase2>, ... ):
    <instrucciones>
```

Ejemplo:

```
1  class Base1 :
2          def __init__(self, x):
3                  self.value=x
4
5          def getData(self):
6                  return self.value
7
8  class Base2 :
9          def __init__(self, c):
10                 self.letter=c
11
12         def getData(self):
13                 return self.letter
```

[8]Mixins and Python, https://www.ianlewis.org/en/mixins-and-python

```
14
15 # herencia múltiple
16 class Derived (Base1, Base2) :
17
18         def __init__(self, i, c, f):
19                 Base1.__init__(self, i)
20                 Base2.__init__(self, c)
21                 self.real=f
22
23         def getReal(self) :
24                 return self.real
25
26 #script de inicio de ejecución
27 b1 = Base1(10)
28 b2 = Base2('Z')
29 d = Derived(7, 'A', 3.5)
30
31 print("Objeto b1 contiene entero ", b1.getData())
32 print("Objeto b2 contiene caracter ", b2.getData())
33 print("Objeto d contiene ", d.getData())
```

Listing 75. Ejemplo de mixins en Python.

En Python, la jerarquía de herencia es definida de derecha a izquierda por lo que hay que tener en cuenta que el resultado puede variar si una clase de más a la izquierda redefine métodos o atributos. Por eso se considera a la clase de la extrema derecha como la "clase base principal", mientras que las otras clases agregan características (es por este aspecto por el que se maneja a veces como mixins). *En caso de redefinir un método, se ejecutará el que se encuentre más a la extrema izquierda (el último en ser redefinido)*

Capítulo 14

Asociaciones entre clases

14.1. Introducción

Una clase puede estar relacionada con otra clase, o en la práctica un objeto con otro objeto.

Concepto
En el modelado de objetos a la relación entre clases se le conoce como **asociación**; mientras que a la relación entre objetos se llama instancia de una asociación.

Ejemplo:

*Una clase **Estudiante** está asociada con una clase **Universidad**.* Una asociación es una **conexión** física o conceptual entre objetos. Las relaciones[1] se consideran de naturaleza **bidireccional**; es decir, ambos lados de la asociación tienen acceso a clase del otro lado de la asociación. Sin embargo, algunas veces únicamente es necesaria una asociación en una dirección (unidireccional).

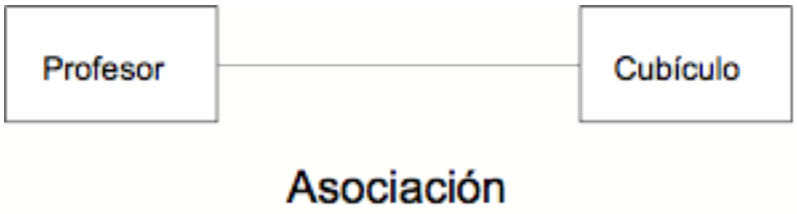


Fig. 14.1. Ejemplo de asociación en UML

[1] El término de relación es usado muchas veces como sinónimo de asociación, debido a que el concepto surge de las relaciones en bases de datos relacionales. Sin embargo el término más apropiado es el de asociación, ya que existen en objetos otros tipos de relaciones, como la relación de agregación y la de herencia.

Comúnmente las asociaciones se representan en los lenguajes de programación orientados a objetos como apuntadores o referencias. Donde un apuntador a una clase B en una clase A indicaría la asociación que tiene A con B; aunque no así la asociación de B con A.

Para una asociación bidireccional es necesario al menos un par de apuntadores, uno en cada clase. Para una asociación unidireccional basta un solo apuntador en la clase que mantiene la referencia.

En el caso de las asociaciones se asumirá que cada objeto puede seguir existiendo de manera independiente, a menos que haya sido creado por el objeto de la clase asociada, en cuyo caso deberá ser eliminado por el destructor del objeto que la creó. Es decir:

Explicación
Si el objeto **A** crea al objeto **B**, es responsabilidad de **A** eliminar a la instancia **B** antes de que **A** sea eliminada. En caso contrario, si **B** es independiente de la instancia **A**, **A** debería enviar un mensaje al objeto **B** para que asigne *NULL* al apuntador de **B** o para que tome una medida pertinente, de manera que no quede apuntando a una dirección inválida.

Es importante señalar que las medidas que se tomen pueden variar de acuerdo a las necesidades de la aplicación, pero bajo ningún motivo se deben dejar accesos a áreas de memoria no permitidas o dejar objetos ”volando”, sin que nadie haga referencia a ellos.

Mencionamos a continuación estructuras clásicas que pueden ser vistas como una asociación:

1. Ejemplo de asociación **unidireccional**: lista ligada.
2. Ejemplo de asociación **bidireccional**: lista doblemente ligada.

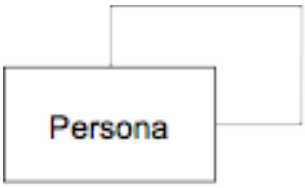
14.2. Asociaciones reflexivas

Es posible tener un tipo de asociación conocida como asociación **reflexiva**.

Concepto
Si una clase mantiene una asociación consigo misma se dice que es una **asociación reflexiva**.

Ejemplo: *Persona* puede tener asociaciones entre sí, si lo que nos interesa es representar a las personas que guardan una relación entre sí, por ejemplo si son parientes. Es decir, un objeto mantiene una asociación con otro objeto de la misma clase.

En términos de implementación significa que la clase tiene una referencia a si misma. De nuevo podemos poner de ejemplo a la clase Nodo en una lista ligada.



Asociación reflexiva

Fig. 14.2. Asociación reflexiva

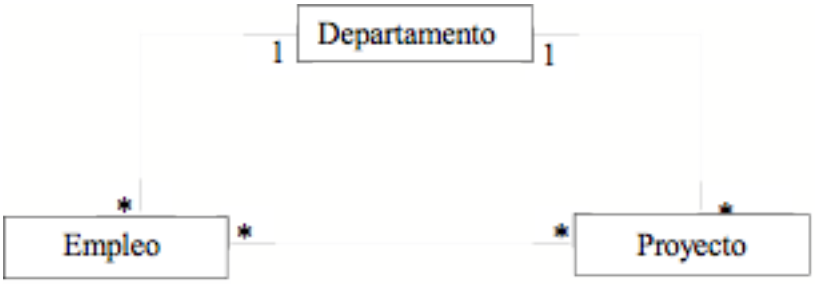


Fig. 14.3. Ejemplo de multiplicidad

14.3. Multiplicidad de una asociación

La **multiplicidad** de una asociación especifica cuantas instancias de una clase se pueden asociar a una sola instancia de otra clase.

Se debe determinar la multiplicidad para cada clase en una asociación.

14.3.1. Tipos de asociaciones según su multiplicidad

"uno a uno": donde dos objetos se asocian de forma exclusiva, uno con el otro. Ejemplo: Uno: Un alumno tiene una boleta de calificaciones. Uno: Una boleta de calificaciones pertenece a un alumno.

"uno a muchos": donde uno de los objetos puede estar asociado con muchos otros objetos. Ejemplo: Uno: un libro solo puede estar prestado a un alumno. Muchos: Un usuario de la biblioteca puede tener muchos libros prestados.

"muchos a muchos": donde cada objeto de cada clase puede estar asociado con muchos otros objetos. Ejemplo: Muchos: Un libro puede tener varios autores. Muchos: Un autor puede tener varios libros.

Podemos apreciar en un diagrama las diversas multiplicidades:

Finalmente, es importante señalar que el control de las asociaciones no se encuentra en general apoyado por los lenguajes de programación, a pesar de ser una necesidad natural en el modelado orientado a objetos, por lo que toda la responsabilidad recae sobre el programador.

14.4. Asociaciones en C++

Ejemplo: un programa que guarda una asociación bidireccional entre clases A y B.

```
class A{
        //lista de atributos

        B       *pB;
};

class B{
        //lista de atributos
        A       *pA;
};
```

En el ejemplo anterior se presenta una asociación bidireccional, por lo que cada clase tiene su respectivo apuntador a la clase contraria de la asociación. Además, deben proporcionarse métodos de acceso a la clase asociada por medio del apuntador.

14.4.1. Multiplicidad de una asociacion en C++

La forma de implementar en C++ este tipo de relaciones puede variar, pero la más común es por medio de apuntadores a objetos. Suponiendo que tenemos asociaciones bidireccionales:

- **"uno a uno"**. Un apuntador de cada lado de la asociación, como se ha visto anteriormente.
- **"uno a muchos"**. Un apuntador de un lado y un arreglo de apuntadores a objetos definido dinámica o estáticamente.

```
class A{
        ...
        B *pB;
};

class B{
        A *p[5];
//ó
A **p;
}
```

Otra forma es manejar una clase que agrupe a pares de direcciones en un objeto independiente de la clase. Por ejemplo una lista o tabla de referencias.

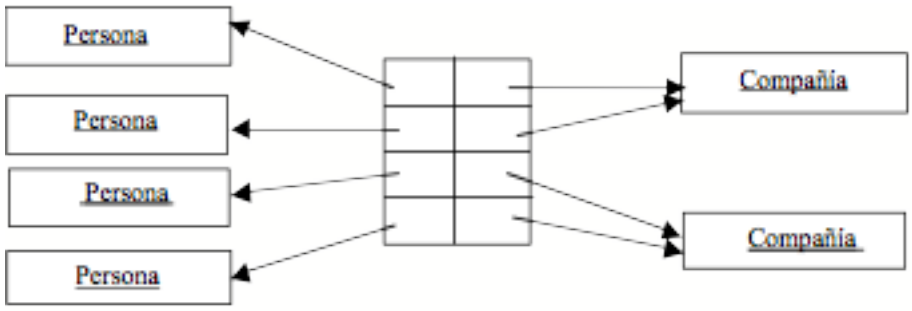


Fig. 14.4. Implementación de multiplicidad mediante tabla de referencias

- **"muchos a muchos"**. Normalmente se utiliza un objeto u objetos independientes que mantiene las asociaciones entre los objetos, de manera similar a la gráfica anterior.

Actividad
Ejercicio 1: Crear una clase Alumno que contiene un atributo matrícula, grupo, carrera y un objeto nombre. Además mantiene una asociación bidireccional de "uno a muchos" con objetos de una clase Libro. La clase Libro contiene el nombre del libro, editorial, edición, año y tiene un arreglo de 3 objetos de la clase nombre para identificar al autor. El método *prestamo*() establecería la asociación y el método *devolución*() eliminaría la asociación entre un Alumno y un Libro. El código de prueba debe presentar un menú que permita establecer y deshacer asociaciones entre alumnos y libros. **Ejercicio 2:** Modificar el ejercicio anterior y en lugar de que grupo sea un atributo simple, deberá ser una clase Grupo que contenga el semestre, salón asignado y que mantenga una relación con un máximo de 30 alumnos. Cuando se elimine al último alumno, el grupo debe desaparecer.

Ejemplo: Se muestra un código simplificado para manejo de asociaciones.
Clase Libro

```
1  #ifndef LIBRO_H_
2  #define LIBRO_H_
3
4  class Persona;
5
6  class Libro {
7          public:
8          char nombre[10];
9          Persona *pPersona;
```

```
10
11          Libro();
12          ~Libro();
13  };
14
15  #endif /*LIBRO_H_*/
16
17
18  #include <iostream>
19  #include "Persona.h"
20  #include "Libro.h"
21
22  Libro::Libro(){
23          nombre[0]='\0';
24          pPersona=NULL;
25  }
26
27  Libro::~Libro(){
28          if(pPersona!=NULL)
29          for(int i=0; i<5; i++)
30                  if (pPersona->pLibrosPres[i]==this)
31                          pPersona->pLibrosPres[i]=NULL;
32
33  }
```

Clase Persona

```
1   #ifndef PERSONA_H_
2   #define PERSONA_H_
3
4   class Libro;
5
6   class Persona {
7           public:
8           Libro *pLibrosPres[5];
9
10          Persona();
11          ~Persona();
12  };
13
14  #endif /*PERSONA_H_*/
15
16  #include <iostream>
```

```
17 #include "Libro.h"
18 #include "Persona.h"
19
20 Persona::Persona(){
21         int i;
22
23         for(i=0; i<5; i++)
24                 pLibrosPres[i]=NULL;
25 }
26
27 Persona::~Persona(){
28                 int i;
29         for(i=0; i<5; i++)
30                 if(pLibrosPres[i]!=NULL)
31                           pLibrosPres[i]->pPersona=NULL;
32 }
```

14.5. Asociaciones en Java

Como en Java el manejo de objetos es mediante referencias, la implementación de la asociación se simplifica en la medida que la sintaxis de Java es más simple.

Ejemplo: un código que guarda una asociación bidireccional entre clases A y B.

```
1  class A{
2          //lista de atributos
3
4          B         pB;
5  }
6
7  class B{
8          //lista de atributos
9          A         pA;
10 }
```

En el ejemplo anterior se presenta una relación bidireccional, por lo que cada clase tiene su respectiva referencia a la clase contraria de la relación. Además, deben proporcionarse métodos de acceso a la clase relacionada por medio de la referencia.

Una asociación unidireccional del ejemplo anterior sería más simple. Veamos el código si se requiere únicamente una relación de A a B.

Ejemplo:

```
1  class A{
2          //lista de atributos
3          B         pB;
4  }
5
6  class B{
7          //lista de atributos
8  }
```

Recordar
Si el objeto **A** crea al objeto B, es responsabilidad de **A** eliminar a la instancia **B** antes de que **A** sea eliminada. En caso contrario, si **B** es independiente de la instancia **A**, **A** debería enviar un mensaje al objeto **B** para que asigne *null* al apuntador de **B** o para que tome una medida pertinente, de manera que no quede apuntando a una dirección inválida.

En Java, ya que cuenta con un recolector de basura, la importancia radicaría en asegurarnos de no mantener enlaces a objetos que ya no son necesarios.

14.5.1. Multiplicidad de una asociación en Java

La forma de implementar en Java este tipo de relaciones puede variar, pero la más común es por medio de referencias a objetos. Suponiendo que tenemos relaciones bidireccionales:

- **"uno a uno"**. Una referencia de cada lado de la relación, como se ha visto anteriormente.
- **"uno a muchos"**. Una referencia de un lado y un arreglo de referencias a objetos del otro lado.

```
class A{
        ...
        B pB;
}
class B{
        A p[];
}
```

Al igual que en C++, es posible manejar una clase independiente que agrupe a pares de direcciones en un objeto independiente de la clase[2]. Por ejemplo, en una estructura de lista.

- **"muchos a muchos"**. Normalmente se utiliza un objeto u objetos independientes que mantiene las relaciones entre los objetos, de manera similar a la solución descrita en el punto anterior.

Ejemplo: Se muestra un código simplificado para manejo de asociaciones.

```
//clase Libro
class Libro {
        private String nombreLibro;
        public Alumno pAlumno;

        public Libro(){
                //al momento de crearse la instancia, no existe
                // relación con ningún Alumno
                pAlumno=null;
        }

        protected void finalize(){
                //si es diferente de null, el libro está asignado a algún Alumno
```

[2]Ver figura en tema correspondiente de C++

```
14                  if(pAlumno!=null)
15                          //busca la referencia de Alumno a Libro para ponerla en nu
16                          for(int i=0; i<5; i++)
17                                  if (pAlumno.pLibrosPres[i]==this)
18                                          pAlumno.pLibrosPres[i]=null;
19          }
20  }
21
22  //clase Alumno
23  class Alumno {
24          public Libro pLibrosPres[];
25          public Alumno(){
26                  int i;
27                  //se asume una multiplicidad de 5
28                  pLibrosPres = new Libro[5]
29                  for(i=0; i<5; i++)
30                          pLibrosPres[i]=null;
31          }
32
33          protected void finalize(){
34
35                  //pone en null todas las asociaciones de los Libros
36                  // a su instancia de Alumno que se elimina
37                  for(int i=0; i<5; i++)
38                          if(pLibrosPres[i]!=null)
39                                  pLibrosPres[i].pAlumno=null;
40          }
41  }
```

Este es un ejemplo parcial de cómo se soluciona el manejo de asociaciones entre clases, ya que además se deben de agregar métodos para establecer y eliminar la asociación, en ambas clases si es una asociación bidireccional, o en una clase únicamente si se trata de una asociación unidireccional. Esos deben de ser los únicos métodos que tengan el control sobre los atributos que mantienen la asociación y no deberían ser manejados directamente, por lo que no deben ser públicos como aquí se presentaron.

Una definición más completa - sin implementación - de la clase *Libro* se aprecia a continuación:

```
1   class Libro {
2           private String nombreLibro;
3           private String clave;
4           public Alumno pAlumno;
5
```

```
6           public Libro(){         }
7
8           public Libro( Alumno pAlumno){         }
9
10          public String getNombreLibro(){         }
11
12          public void setNombreLibro(String n){        }
13
14          public String getClave(){         }
15
16          public void setClave(String cve){         }
17
18          public boolean setAsociacion(Alumno pAlumno){         }
19
20          public boolean unsetAsociacion(){         }
21
22          public Alumno getAlumno(){        }
23
24          protected finalize {}
25  }
```

Este sería un estilo más apropiado para el desarrollo de asociaciones, aunque existen otros más elaborados.

Capítulo 15

Objetos compuestos

Algunas veces una clase no puede modelar adecuadamente una entidad basándose únicamente en tipos de datos simples. Los LPOO permiten a una clase **contener** objetos. Un objeto forma parte directamente de la clase en la que se encuentra declarado.

El objeto compuesto es una especie de relación, pero con una asociación más fuerte con los objetos relacionados. A la noción de objeto **compuesto** se le conoce también como objeto **complejo** o **agregado**.

Concepto
Rumbaugh define a la agregación como *"una forma fuerte de asociación, en la cual el objeto agregado está formado por componentes. Los componentes forman parte del agregado. El agregado, es un objeto extendido que se trata como una unidad en muchas operaciones, aún cuando conste físicamente de varios objetos menores."[11]*

Ejemplo: Un automóvil se puede considerar ensamblado o agregado, donde el motor y la carrocería serian sus componentes.

El concepto de agregación puede ser relativo a la conceptualización que se tenga de los objetos que se quieran modelar.

Dicho concepto implica obviamente cierta dependencia entre los objetos, por lo que hay que tener en cuenta que pasa con los objetos que son parte del objeto compuesto cuando éste último se destruye. En general tenemos dos opciones:

1. Cuando el objeto agregado se destruye, los objetos que lo componen no tienen necesariamente que ser destruidos.

2. Cuando el agregado es destruido también sus componentes se destruyen.

Un objeto que es parte de otro objeto, puede a su vez ser un objeto compuesto. De esta forma podemos tener múltiples niveles. Un objeto puede ser un **agregado recursivo**, es decir, tener un objeto de su misma clase.

Ejemplo: Directorio de archivos.

Sin embargo, la forma en que se implemente la agregación puede no permitir la agregación recursiva.

15.1. Objetos compuestos en C++

Por el momento vamos a considerar la segunda opción mencionada anteriormente, por ser más fácil de implementar y porque es la acción natural de los objetos que se encuentran embebidos como un atributo más en una clase.

Ejemplo:

```
1  class Nombre {
2          private:
3            char paterno[20],
4            materno[20],
5            nom[15];
6          public:
7          set(char *, char*, char *);
8          ...
9  };
10
11 class Persona {
12         private:
13         int edad;
14         Nombre nombrePersona;
15         ...
16 };
```

Al crear un objeto compuesto, cada uno de sus componentes es creado con sus respectivos constructores. Para inicializar esos objetos componentes tenemos dos opciones:

1. En el constructor del objeto compuesto llamar a los métodos *set* correspondientes a la modificación de los atributos de los objetos componentes.

2. Pasar en el constructor del objeto compuesto los argumentos a los constructores de los objetos componentes.

Sintaxis

```
<clase>::<constructor>(<lista de argumentos>)
   : <objeto componente 1>(<lista de argumentos sin el tipo>),...
```

donde la lista de argumentos del objeto compuesto debe incluir a los argumentos de los objetos componentes, para que puedan ser pasados en la creación del objeto.

Tema sugerido
Apéndice X: UMLGEC++

Ejemplo:

```
#include <iostream>
#include <string.h>
using namespace std;

class Nombre {
        char *nombre, *paterno, *materno;
        public:
                Nombre(char *n, char *p, char*m){
                        nombre=new char[strlen(n)+1];
                        paterno=new char[strlen(p)+1];
                        materno=new char[strlen(m)+1];
                        strcpy(nombre, n);
                        strcpy(paterno, p);
                        strcpy(materno, m);
                }
                ~Nombre(){
                        cout<<"destructor de Nombre: "<<nombre<<endl;
                        delete []nombre;
                        delete []paterno;
                        delete []materno;
                }
};

class Persona{
        Nombre miNombre;
        int        edad;
        public:
        Persona(char *n, char *p, char*m): miNombre(n, p, m){
                edad=0;
        }
};

int main() {
        Persona *per1;
        per1= new Persona("uno", "dos", "tres");
```

```
36          Persona per2("Bob", "the", "builder");
37          delete per1;
38          return 0;
39  }
```

Listing 76. Ejemplo de composición en C++.

15.2. Objetos compuestos en Java

En Java, puede no existir mucha diferencia entre la implementación de la asociación y la agregación, debido a que en Java los objetos siempre son manejados por referencias, pero el concepto se debe tener en cuenta para su manejo, además de ser relevante a nivel de diseño de software.

Recordemos que en general hay dos opciones para el manejo de la agregación:

1. Cuando el objeto agregado se destruye, los objetos que lo componen no tienen necesariamente que ser destruidos.

2. Cuando el agregado es destruido también sus componentes se destruyen.

Al igual que en C++, vamos a considerar la segunda opción, por ser más fácil de implementar y es la acción natural de los objetos que se encuentran embebidos como un atributo más una clase.

Ejemplo:

```
1  class Nombre {
2          private String paterno;
3          private String materno;
4          private String nom;
5
6          public        set(String pat, String mat, String n) {
7                  ...
8          }
9                  ...
10 }
11
12 class Persona {
13         private int edad;
14         private Nombre nombrePersona;
15         ...
16 }
```

A diferencia de lo que sucede en C++, los atributos compuestos no tienen memoria asignada, es decir, los objetos compuestos no han sido realmente creados en el momento en que se crea el objeto componente. Es responsabilidad del constructor del objeto componente inicializar los objetos miembros o compuestos, si es que así se requiere.

Para inicializar esos objetos componentes tenemos dos opciones:

1. En el constructor del objeto compuesto llamar a los métodos set correspondientes a la modificación de los atributos de los objetos componentes, esto claro está, después de asignarle la memoria a los objetos componentes.

2. Llamar a algún constructor especializado del objeto componente en el momento de crearlo.

Ejemplo:

```
1  //Programa Persona
2  class Nombre {
3          private String nombre,
4                  paterno,
5                  materno;
6          public Nombre(String n, String p, String m){
7                          nombre= new String(n);
8                          paterno= new String(p);
9                          materno= new String(m);
10         }
11 }
12
13 public class Persona{
14         private Nombre miNombre;
15         private int edad;
16         public Persona(String n, String p, String m) {
17                 miNombre= new Nombre(n, p, m);
18                 edad=0;
19         }
20
21         public static void main(String args[]) {
22                 Persona per1;
23                 per1= new Persona("uno", "dos", "tres");
24                 Persona per2= new Persona("mi nombre", "mi apellido", "otro apellido");
25         }
26 }
```

Listing 77. Ejemplo de composición en Java.

Pero también es posible que un objeto sea un agregado recursivo, es decir, tener como parte de su componente un objeto de su misma clase. Considerar por ejemplo un directorio de archivos, donde cada directorio puede contener, además de archivos, a otros directorios[1].

[1] Lo importante aquí es considerar en que solo existe la **posibilidad** de contener un objeto de si mismo. Si esto fuera una condición obligatoria y no opcional, estaríamos definiendo un **objeto infinito**. Este problema se ve reflejado en lenguajes como C++, donde la forma más simple de implementar la agregación es definiendo un objeto al cual se le asigna espacio en tiempo de compilación, generando entonces el problema de que cada objeto debe reservar memoria para sus componentes, por lo que el compilador no permite que de esta manera se autocontenga. En Java esto no generaría problema porque implícitamente todos los atributos que no son datos simples requieren de una asignación de memoria dinámica.

Capítulo 16

Polimorfismo

El polimorfismo es la capacidad de ofrecer una interfaz para distintos tipos, de manera que un tipo polimórfico es al que se le pueden aplicar operaciones con distintos tipos. Existen distintos tipos de polimorfismo:

- **Polimorfismo ad-hoc[12].** Es cuando una función tiene un conjunto de implementaciones distintas sobre un rango de tipos de datos y sus combinaciones. Este tipo de polimorfismo es soportado en muchos lenguajes por medio de la sobrecarga y es también conocido como **polimorfismo estático**.
- **Polimorfismo paramétrico[12] .** Cuando se escribe código sin especificar el tipo que va a ser usado. En POO es conocido como programación genérica. En programación funcional es llamado simplemente polimorfismo.
- **Polimorfismo de subtipo o de inclusión**[1]**[13] .** Es el tipo de polimorfismo más común, en el que un conjunto de instancias de distintas clases están relacionadas por una superclase. Tan común que es lo que muchas veces se explica como polimorfismo. Tamnién conocido como **polimorfismo dinámico**.

[1]"Polymorphic types are types whose operations are applicable to values of more than one type."

Capítulo 17

Polimorfismo de subtipos

Concepto
”La capacidad de polimorfismo permite crear programas con mayores posibilidades de expansiones futuras, aún para procesar en cierta forma objetos de clases que no han sido creadas o están en desarrollo.” [5]

Concepto
El **polimorfismo** se define como la capacidad de objetos de clases diferentes, relacionados mediante herencia, a responder de forma distinta a una misma llamada de un método. [10]

Tener en cuenta que no es lo mismo que simplemente redefinir un método de clase base en una clase derivada, pues como se vio anteriormente, si se tiene a un apuntador de clase base y a través de el se hace la llamada a un método, se ejecuta el método de la clase base independientemente del objeto referenciado por el apuntador. Este no es un comportamiento polimórfico.

17.1. Polimorfismo y funciones virtuales C++

En C++, el polimorfismo se implementa a través de clases derivadas y **funciones virtuales**. Al hacer una solicitud de un método, a través de un apuntador a clase base para usar un método virtual, C++ determina el método que corresponda al objeto de la clase a la que pertenece, y no el método de la clase base.

Concepto
Una función virtual es un método miembro declarado como **virtual** en una clase base y siendo este método redefinido en una o más clases derivadas.

Las funciones virtuales son muy especiales, debido a que cuando una función es accedida por un apuntador a una clase base, y éste mantiene una referencia a un objeto de una clase derivada, el programa determina en tiempo de ejecución a que función llamar, de acuerdo al tipo de objeto al que se apunta. Esto se conoce como **ligadura tardía**[1] y el compilador de C++ incluye en el código máquina el manejo de ese tipo de asociación de métodos.

La utilidad se da cuando se tiene un método en una clase base, y éste es declarado virtual. De esta forma, cada clase derivada puede tener su propia implementación del método si es que así lo requiere la clase; y si un apuntador a clase base hace referencia a cualquiera de los objetos de clases derivadas, se determina dinámicamente cual de todos los métodos debe ejecutar.

La sintaxis en C++ implica declarar al método de la clase base con la palabra reservada *virtual*, redefiniendo ese método en cada una de las clases derivadas.

Al declarar un método como virtual, este método se conserva así a través de toda la jerarquía de herencia, del punto en que se declaró hacia abajo. Aunque de este modo no es necesario volver a usar la palabra virtual en ninguno de los métodos inferiores del mismo nombre, es posible declararlo de forma explícita para que el programa sea más claro.

Es importante señalar que las funciones virtuales que sean redefinidas en clases derivadas, deben tener además de la misma firma que la función virtual base, el mismo tipo de retorno.

[1] Término opuesto a **ligadura temprana** o **ligadura estática**, la cual asocia los métodos en tiempo de compilación.

Sintaxis

```
class base {
        virtual <tipo> <método> (<parámetros);
};
```

Ejemplo:

```
1  //ejemplo funciones virtuales
2  #include <iostream>
3  using namespace std;
4
5  class base {
6  public:
7          virtual void quien(){
8                  cout<<"base\n";
9          }
10 };
11
12 class primera: public base {
13 public:
14         void quien(){
15                 cout<<"primera\n";
16         }
17 };
18
19 class segunda: public base {
20 public:
21         void quien(){
22                 cout<<"segunda\n";
23         }
24 };
25
26 class tercera: public base {
27 };
28
29 class cuarta: public base {
30 public:
31         //No se vale con un tipo de dato diferente
32         /*int quien(){
```

```
33                  cout<<"cuarta\n";
34                  return 1;
35          }*/
36  };
37
38  int main() {
39          base objBase, *pBase;
40          primera obj1;
41          segunda obj2;
42          tercera obj3;
43          cuarta  obj4;
44
45          pBase=&objBase;
46          pBase->quien();
47
48          pBase=&obj1;
49          pBase->quien();
50
51          pBase=&obj2;
52          pBase->quien();
53
54          pBase=&obj3;
55          pBase->quien();
56
57          pBase=&obj4;
58          pBase->quien();
59
60          return 0;
61  }
```

Listing 78. Ejemplo de polimorfismo en C++.

Hay que hacer notar que las funciones virtuales pueden seguirse usando sin apuntadores, mediante un objeto de la clase. De esta forma, el método a ejecutar se determina de manera estática; es decir, en tiempo de compilación (**ligadura estática**). Obviamente el método a ejecutar es aquel definido en la clase del objeto o el heredado de su clase base, si la clase derivada no lo redefinió.

La sobrecarga no utiliza ligadura dinámica. Esta es resuelta en tiempo de compilación. Si se declara en una clase derivada un método con otro tipo de dato como retorno, el compilador manda un error, ya que esto no es permitido.

Si se declara un método con el mismo nombre pero diferentes parámetros, la función virtual queda desactivada de ese punto hacia abajo en la jerarquía de herencia.

17.1.1. Clase abstracta y clase concreta en C++

Existen clases que son útiles para representar una estructura en particular, pero que no van a tener la necesidad de generar objetos directamente a partir de esa clase, éstas se conocen como **clases abstractas**, o de manera más apropiada como **clases base abstractas**, puesto que sirven para definir una estructura jerárquica.

La clase base abstracta entonces, tiene como objetivo proporcionar una clase base que ayude al modelado de la jerarquía de herencia, aunque esta sea muy general y no sea práctico tener instancias de esa clase.

Por lo tanto, de una clase abstracta no se pueden tener objetos, mientras que en clases a partir de las cuales se puedan instanciar objetos se conocen como **clases concretas**.

En C++, una clase se hace abstracta al declarar **al menos uno** de los métodos virtuales como puro. Un método o función virtual pura es aquel que en su declaración tiene el inicializador de $= 0$.

Sintaxis

```
virtual <tipo> <nombre>(<parámetros>) =0;    //virtual pura
```

Es importante tener en cuenta que una clase sigue siendo abstracta hasta que no se implemente la función virtual pura, en una de las clases derivadas. Si no se hace la implementación, la función se hereda como virtual pura y por lo tanto la clase sigue siendo considerada como abstracta.

Aunque no se pueden tener objetos de clases abstractas, si se pueden tener apuntadores a objetos de esas clases, permitiendo una manipulación de objetos de las clases derivadas mediante los apuntadores a la clase abstracta.

17.1.2. Destructores virtuales

Cuando se aplica la instrucción *delete* a un apuntador de clase base, será ejecutado el destructor de la clase base sobre el objeto, independientemente de la clase a la que pertenezca. La solución es declarar al destructor de la clase base como virtual. De esta forma al borrar a un objeto se ejecutará el destructor de la clase a la que pertenezca el objeto referenciado, a pesar de que los destructores no tengan el mismo nombre.

Un constructor no puede ser declarado como virtual.

Ejemplos de funciones virtuales y polimorfismo:

Ejemplo:**Programa de cálculo de salario.**

```
1 // EMPLEADO.H
2 // Abstract base class Employee
3 #ifndef EMPLEADO_H_
4 #define EMPLEADO_H_
5
```

```
6  class Employee {
7          public:
8             Employee(const char *, const char *);
9             ~Employee();
10            const char *getFirstName() const;
11            const char *getLastName() const;
12
13            virtual float earnings() const = 0; // virtual pura
14            virtual void print() const = 0;     // virtual pura
15         private:
16            char *firstName;
17            char *lastName;
18 };
19
20 #endif /*EMPLEADO_H_*/
21
22 // EMPLEADO.CPP
23 #include <iostream>
24 #include <string>
25 #include <assert.h>
26 #include "empleado.h"
27
28 Employee::Employee(const char *first, const char *last)
29 {
30    firstName = new char[ strlen(first) + 1 ];
31    assert(firstName != 0);
32    strcpy(firstName, first);
33
34    lastName = new char[ strlen(last) + 1 ];
35    assert(lastName != 0);
36    strcpy(lastName, last);
37 }
38
39 Employee::~Employee()
40 {
41    delete [] firstName;
42    delete [] lastName;
43 }
44
45 const char *Employee::getFirstName() const
46 {
47
```

```
48    return firstName;
49 }
50
51 const char *Employee::getLastName() const
52 {
53       return lastName;
54 }
55 // JEFE.H
56 // Clase drivada de empleado
57 #ifndef JEFE_H_
58 #define JEFE_H_
59
60 #include "empleado.h"
61
62 class Boss : public Employee {
63         public:
64            Boss(const char *, const char *, float = 0.0);
65            void setWeeklySalary(float);
66            virtual float earnings() const;
67            virtual void print() const;
68         private:
69            float weeklySalary;
70 };
71
72 #endif /*JEFE_H_*/
73
74 // JEFE.CPP
75 #include <iostream>
76 #include "jefe.h"
77 using namespace std;
78 Boss::Boss(const char *first, const char *last, float s)
79    : Employee(first, last)
80 { weeklySalary = s > 0 ? s : 0; }
81
82 void Boss::setWeeklySalary(float s)
83 { weeklySalary = s > 0 ? s : 0; }
84
85 float Boss::earnings() const { return weeklySalary; }
86
87 void Boss::print() const
88 {
89    cout << "\n             Jefe: " << getFirstName()
```

```
90          << ' ' << getLastName();
91  }
92
93
94  // COMIS.H
95  // Trabajador por comisión derivado de Empleado
96  #ifndef COMIS_H_
97  #define COMIS_H_
98  #include "empleado.h"
99
100 class CommissionWorker : public Employee {
101         public:
102            CommissionWorker(const char *, const char *,
103                     float = 0.0, float = 0.0, int = 0);
104            void setSalary(float);
105            void setCommission(float);
106            void setQuantity(int);
107            virtual float earnings() const;
108            virtual void print() const;
109         private:
110           float salary;     // salario base por semana
111            float commission; // comisión por cada venta
112            int quantity;     // cantidad de elementos vendidos por semana
113 };
114
115 #endif /*COMIS_H_*/
116
117 // COMIS.CPP
118 #include <iostream>
119 #include "comis.h"
120 using namespace std;
121
122 CommissionWorker::CommissionWorker(const char *first,
123    const char *last, float s, float c, int q)
124    : Employee(first, last)
125 {
126    salary = s > 0 ? s : 0;
127    commission = c > 0 ? c : 0;
128    quantity = q > 0 ? q : 0;
129 }
130
131
```

```
132 void CommissionWorker::setSalary(float s)
133 { salary = s > 0 ? s : 0; }
134
135 void CommissionWorker::setCommission(float c)
136 { commission = c > 0 ? c : 0; }
137
138 void CommissionWorker::setQuantity(int q)
139 { quantity = q > 0 ? q : 0; }
140
141 float CommissionWorker::earnings() const
142 { return salary + commission * quantity; }
143
144 void CommissionWorker::print() const
145 {
146    cout << "\nTrabajador  por comision: " << getFirstName()
147         << ' ' << getLastName();
148 }
149
150 // PIEZA.H
151 // Trabajador por pieza derivado de Empleado
152 #ifndef PIEZA_H_
153 #define PIEZA_H_
154
155 #include "empleado.h"
156
157 class PieceWorker : public Employee {
158         public:
159            PieceWorker(const char *, const char *,
160                float = 0.0, int = 0);
161            void setWage(float);
162            void setQuantity(int);
163            virtual float earnings() const;
164            virtual void print() const;
165         private:
166            float wagePerPiece; // pago por cada pieza
167            int quantity;       // piezas por semana
168 };
169
170 #endif /*PIEZA_H_*/
171
172 // PIEZA.CPP
173 #include <iostream>
```

```
174 #include "pieza.h"
175 using namespace std;
176
177 // Constructor for class PieceWorker
178 PieceWorker::PieceWorker(const char *first,
179              const char *last, float w, int q)
180    : Employee(first, last)
181 {
182    wagePerPiece = w > 0 ? w : 0;
183    quantity = q > 0 ? q : 0;
184 }
185
186 void PieceWorker::setWage(float w)
187 { wagePerPiece = w > 0 ? w : 0; }
188
189 void PieceWorker::setQuantity(int q)
190 { quantity = q > 0 ? q : 0; }
191
192 float PieceWorker::earnings() const
193 { return quantity * wagePerPiece; }
194
195 void PieceWorker::print() const {
196    cout << "\n     Tabajador por pieza: " << getFirstName()
197         << ' ' << getLastName();
198 }
199
200 // HORA.H
201 // Trabajador por hora derivado de Empleado
202 #ifndef HORA_H_
203 #define HORA_H_
204
205 #include "empleado.h"
206
207 class HourlyWorker : public Employee {
208         public:
209            HourlyWorker(const char *, const char *,
210                 float = 0.0, float = 0.0);
211            void setWage(float);
212            void setHours(float);
213            virtual float earnings() const;
214            virtual void print() const;
215         private:
```

```
216             float wage;   // salario por hora
217             float hours;  // horas trabajadas en la semana
218 };
219
220 #endif /*HORA_H_*/
221
222 // HORA.CPP
223 #include <iostream>
224 #include "hora.h"
225 using namespace std;
226
227 HourlyWorker::HourlyWorker(const char *first, const char *last,
228                            float w, float h)
229    : Employee(first, last)
230 {
231    wage = w > 0 ? w : 0;
232    hours = h >= 0 && h < 168 ? h : 0;
233 }
234
235 void HourlyWorker::setWage(float w) { wage = w > 0 ? w : 0; }
236
237 void HourlyWorker::setHours(float h)
238 { hours = h >= 0 && h < 168 ? h : 0; }
239
240 float HourlyWorker::earnings() const { return wage * hours; }
241
242 void HourlyWorker::print() const
243 {
244    cout << "\n    Trabajador por hora: " << getFirstName()
245         << ' ' << getLastName();
246 }
247
248 // main.cpp
249 #include <iostream>
250 #include <iomanip>
251 #include "empleado.h"
252 #include "jefe.h"
253 #include "comis.h"
254 #include "pieza.h"
255 #include "hora.h"
256 using namespace std;
257
```

```
258 int main(){
259    // formato de salida
260    cout << setprecision(2);
261 
262    Employee *ptr;  // apuntador a clase base
263 
264    Boss b("John", "Smith", 800.00);
265    ptr = &b;  // apuntador de clase base apuntando a objeto de clase derivada
266    ptr->print();                            // ligado dinámico
267    cout << " ganado $" << ptr->earnings(); // ligado dinámico
268    b.print();                               // ligado estático
269    cout << " ganado $" << b.earnings();    // ligado estático
270 
271    CommissionWorker c("Sue", "Jones", 200.0, 3.0, 150);
272    ptr = &c;
273    ptr->print();
274    cout << " ganado $" << ptr->earnings();
275    c.print();
276    cout << " ganado $" << c.earnings();
277 
278    PieceWorker p("Bob", "Lewis", 2.5, 200);
279    ptr = &p;
280    ptr->print();
281    cout << " ganado $" << ptr->earnings();
282    p.print();
283    cout << " ganado $" << p.earnings();
284 
285    HourlyWorker h("Karen", "Precio", 13.75, 40);
286    ptr = &h;
287    ptr->print();
288    cout << " ganado $" << ptr->earnings();
289    h.print();
290    cout << " ganado $" << h.earnings();
291 
292    cout << endl;
293    return 0;
294 }
```

Listing 79. Ejemplo de funciones virtuales y polimorfismo en C++. Programa de cálculo de salario..

Ejemplo:**Programa de figuras geométricas con una interfaz abstracta Shape (For-**

ma)

```
// Figura.H
#ifndef figura_H
#define figura_H

class Shape {
public:
   virtual float area() const { return 0.0; }
   virtual float volume() const { return 0.0; }
   virtual void printShapeName() const = 0; // virtual pura
};
#endif

// Punto.H
#ifndef PUNTO_H_
#define PUNTO_H_
#include <iostream>
#include "figura.h"

class Point : public Shape {
   friend ostream &operator<<(ostream &, const Point &);
public:
   Point(float = 0, float = 0);
   void setPoint(float, float);
   float getX() const { return x; }
   float getY() const { return y; }
   virtual void printShapeName() const { cout << "Punto: "; }
private:
   float x, y;
};
#endif /*PUNTO_H_*/

// Punto.CPP
#include <iostream.h>
#include "punto.h"
Point::Point(float a, float b)
{
   x = a;
   y = b;
}

```

```
41 void Point::setPoint(float a, float b)
42 {
43    x = a;
44    y = b;
45 }
46
47 ostream &operator<<(ostream &output, const Point &p)
48 {
49    output << '[' << p.x << ", " << p.y << ']';
50    return output;
51 }
52
53 // Circulo.H
54 #ifndef CIRCULO_H_
55 #define CIRCULO_H_
56 #include "punto.h"
57
58 class Circle : public Point {
59    friend ostream &operator<<(ostream &, const Circle &);
60 public:
61    Circle(float r = 0.0, float x = 0.0, float y = 0.0);
62
63    void setRadius(float);
64    float getRadius() const;
65    virtual float area() const;
66    virtual void printShapeName() const { cout << "Circulo: "; }
67 private:
68    float radius;
69 };
70 #endif /*CIRCULO_H_*/
71
72 // Circulo.CPP
73 #include <iostream>
74 #include <iomanip>
75 #include "circulo.h"
76 using namespace std;
77
78 Circle::Circle(float r, float a, float b)
79    : Point(a, b)
80  { radius = r > 0 ? r : 0; }
81
82 void Circle::setRadius(float r) { radius = r > 0 ? r : 0; }
```

```
83
84  float Circle::getRadius() const { return radius; }
85
86  float Circle::area() const { return 3.14159 * radius * radius; }
87
88  ostream &operator<<(ostream &output, const Circle &c)
89  {
90     output << '[' << c.getX() << ", " << c.getY()
91            << "]; Radio=" << setprecision(2) << c.radius;
92
93     return output;
94  }
95
96  // Cilindro.H
97  #ifndef CILINDRO_H_
98  #define CILINDRO_H_
99  #include "circulo.h"
100
101 class Cylinder : public Circle {
102    friend ostream &operator<<(ostream &, const Cylinder &);
103 public:
104    Cylinder(float h = 0.0, float r = 0.0,
105             float x = 0.0, float y = 0.0);
106
107    void setHeight(float);
108    virtual float area() const;
109    virtual float volume() const;
110    virtual void printShapeName() const { cout << "Cilindro: "; }
111 private:
112    float height;   // altura del cilindro
113 };
114 #endif /*CILINDRO_H_*/
115
116 // Cilindro.CPP
117 #include <iostream>
118 #include <iomanip>
119 #include "cilindro.h"
120
121 Cylinder::Cylinder(float h, float r, float x, float y)
122    : Circle(r, x, y)
123 { height = h > 0 ? h : 0; }
124
```

```
125 void Cylinder::setHeight(float h)
126    { height = h > 0 ? h : 0; }
127
128 float Cylinder::area() const
129 {
130       return 2 * Circle::area() +
131           2 * 3.14159 * Circle::getRadius() * height;
132 }
133
134 float Cylinder::volume() const
135 {
136    float r = Circle::getRadius();
137    return 3.14159 * r * r * height;
138 }
139
140 ostream &operator<<(ostream &output, const Cylinder& c)
141 {
142    output << '[' << c.getX() << ", " << c.getY()
143           << "]; Radio=" << setprecision(2) << c.getRadius()
144           << "; Altura=" << c.height;
145    return output;
146 }
147
148 // main.CPP
149 #include <iostream>
150 #include <iomanip>
151 using namespace std;
152
153 #include "figura.h"
154 #include "punto.h"
155 #include "circulo.h"
156 #include "cilindro.h"
157
158 int main(){
159    Point point(7, 11);
160    Circle circle(3.5, 22, 8);
161    Cylinder cylinder(10, 3.3, 10, 10);
162
163    point.printShapeName();     // ligado estático
164    cout << point << endl;
165
166    circle.printShapeName();
```

```
167     cout << circle << endl;
168
169     cylinder.printShapeName();
170     cout << cylinder << "\n\n";
171     cout << setprecision(2);
172     Shape *ptr;                 // apuntador de clase base
173
174     // apuntador de clase base referenciando objeto de clase derivada
175     ptr = &point;
176     ptr->printShapeName();      // ligado dinámico
177     cout << "x = " << point.getX() << "; y = " << point.getY()
178          << "\nArea = " << ptr->area()
179          << "\nVolumen = " << ptr->volume() << "\n\n";
180
181     ptr = &circle;
182     ptr->printShapeName();
183     cout << "x = " << circle.getX() << "; y =" << circle.getY()
184          << "\nArea = " << ptr->area()
185          << "\nVolumen = " << ptr->volume() << "\n\n";
186
187     ptr = &cylinder;
188     ptr->printShapeName();      // dynamic binding
189     cout << "x = " << cylinder.getX() << "; y = " << cylinder.getY()
190          << "\nArea = " << ptr->area()
191          << "\nVolumen = " << ptr->volume() << endl;
192     return 0;
193 }
```

Listing 80. Ejemplo de polimorfismo en C++. Programa de figuras geométricas con una interfaz abstracta Shape (Forma).

17.2. Polimorfismo y clases abstractas Java

El polimorfismo es implementado en Java a través de clases derivadas y clases **abstractas**.

Concepto
Recordar: El **polimorfismo** se define como la capacidad de objetos de clases diferentes, relacionados mediante herencia, a responder de forma distinta a una misma llamada de un método.

Al hacer una solicitud de un método, a través de una variable de referencia a clase base para usar un método, Java determina el método que corresponda al objeto de la clase a la que pertenece, y no el método de la clase base.

Los métodos en Java - a diferencia de C++ - tienen este comportamiento por default, debido a que cuando un método es accedido por una referencia a una clase base, y esta mantiene una referencia a un objeto de una clase derivada, el programa determina **en tiempo de ejecución** a que método llamar, de acuerdo al tipo de objeto al que se apunta.

Esto como ya se ha visto, se conoce como **ligadura tardía** y permite otro nivel de reutilización de código, resaltado por la simplificación con respecto a C++ de no tener que declarar al método como virtual.

Ejemplo:

```
1  //ejemplo Prueba
2  class base {
3          public void quien(){
4                  System.out.println("base");
5          }
6  }
7
8  class primera extends base {
9          public void quien(){
10                 System.out.println("primera");
11         }
12 }
13
14 class segunda extends base {
15         public void quien(){
16                 System.out.println("segunda");
17         }
18 }
```

```
19
20 class tercera extends base { }
21
22 class cuarta extends base {
23         /*public int quien(){   No se vale con un tipo de dato diferente
24                 System.out.println("cuarta");
25                  return 1;
26          }*/
27 }
28
29 public class Prueba {
30         public static void main(String args[]) {
31                 base objBase= new base(), pBase;
32                 primera obj1= new primera();
33                 segunda obj2= new segunda();
34                 tercera obj3= new tercera();
35                 cuarta  obj4= new cuarta();
36
37                 pBase=objBase;
38                 pBase.quien();
39
40                 pBase=obj1;
41                 pBase.quien();
42
43                 pBase=obj2;
44                 pBase.quien();
45
46                 pBase=obj3;
47                 pBase.quien();
48
49                 pBase=obj4;
50                 pBase.quien();
51         }
52 }
```

Listing 81. Ejemplo de polimorfismo en Java.

Como se aprecia en el ejemplo anterior, en caso de que el método no sea redefinido, se ejecuta el método de la clase base.

Es importante señalar que – al igual que en C++- los métodos que sean redefinidos en clases derivadas, deben tener además de la misma firma que método base, el mismo tipo de retorno. Si se declara en una clase derivada un método con otro tipo de dato como retorno, se generará un error en tiempo de compilación.

17.2.1. Clase abstracta y clase concreta en Java

Concepto
Recordar: Una **clase base abstracta**, es aquella que es definida para especificar características generales que van a ser aprovechadas por sus clases derivadas, pero no se necesita instanciar a dicha superclase.

Sintaxis

`abstract class ClaseAbstracta {`
`  //código de la clase`
`}`

Además, existe la posibilidad de contar con métodos abstractos:

Concepto
Un **método abstracto** lleva la palabra reservada *abstract* y contiene sólo el nombre y su firma. No necesita implementarse, ya que esto es tarea de las subclases.

Si una clase contiene al menos un método abstracto, toda la clase es considerada abstracta y es conveniente, por claridad, declararla como tal. Es posible claro, declarar a una clase como abstracta sin que tenga métodos abstractos.

Ejemplo básico para un método abstracto:

```
1 abstract class ClaseAbstracta {
2
3         public abstract void noTengoCodigo( int x);
4
5 }
```

Si se crea una subclase de una clase que contiene un método abstracto, deberá de especificarse el código de ese método; de lo contrario, el método seguirá siendo abstracto y por consecuencia también lo será la subclase[2].

[2]En C++, una clase se hace abstracta al declarar al menos uno de los métodos virtuales como puro.

Aunque no se pueden tener objetos de clases abstractas, si se pueden tener referencias a objetos de esas clases, permitiendo una manipulación de objetos de las clases derivadas mediante las referencias a la clase abstracta.

El uso de clases abstractas **fortalece** al polimorfismo, al poder partir de clases definidas en lo general, sin implementación de código, pero pudiendo ser agrupadas todas mediante variables de referencia a las clases base.

Ejemplos de clases abstractas y polimorfismo:

Programa de cálculo de salario

```
1  // Clase base abstracta Employee
2  public abstract class Employee {
3          private String firstName;
4          private String lastName;
5
6             // Constructor
7             public Employee( String first, String last )  {
8        firstName = new String ( first );
9        lastName = new String( last );
10            }
11
12            public String getFirstName()  {
13                    return new String( firstName );
14         }
15
16            public String getLastName()   {
17                    return new String( lastName );
18         }
19
20            // el metodo abstracto debe de ser implementado por cada
21            // clase derivada de Employee para poder ser
22            // instanciadas las subclases
23            public abstract double earnings();
24 }
25
26 // Clase Boss class derivada de Employee
27 public final class Boss extends Employee {
28         private double weeklySalary;
29
30 public Boss( String first, String last, double s) {
31                 super( first, last );  // llamada al constructor de clase base
32                 setWeeklySalary( s );
33            }
```

```
34
35          public void setWeeklySalary( double s ){
36                  weeklySalary = ( s > 0 ? s : 0 );
37          }
38
39          // obtiene pago del jefe
40          public double earnings() {
41                  return weeklySalary;
42          }
43
44          public String toString() {
45                  return "Jefe: " + getFirstName() + ' ' +
46               getLastName();
47          }
48  }
49
50  // Clase PieceWorker derivada de Employee
51  public final class PieceWorker extends Employee {
52          private double wagePerPiece; // pago por pieza
53          private int quantity;        // piezas por semana
54
55          public PieceWorker( String first, String last,
56                          double w, int q )    {
57                  super( first, last );
58                  setWage( w );
59                  setQuantity( q );
60          }
61
62          public void setWage( double w )    {
63                  wagePerPiece = ( w > 0 ? w : 0 );
64          }
65
66          public void setQuantity( int q )    {
67                  quantity = ( q > 0 ? q : 0 );
68          }
69
70          public double earnings()   {
71                  return quantity * wagePerPiece;
72          }
73
74          public String toString()   {
75                  return "Trabajador por pieza: " +
```

```
76              getFirstName() + ' ' + getLastName();
77          }
78  }
79
80  // Clase HourlyWorker derivada de Employee
81  public final class HourlyWorker extends Employee {
82          private double wage;   // pago por hora
83          private double hours;  // horas trabajadas por semana
84
85          public HourlyWorker( String first, String last,
86                          double w, double h )   {
87                  super( first, last );
88                  setWage( w );
89                  setHours( h );
90          }
91
92          public void setWage( double w )   {
93                  wage = ( w > 0 ? w : 0 );
94          }
95
96          public void setHours( double h )   {
97                  hours = ( h >= 0 && h < 168 ? h : 0 );
98          }
99
100         public double earnings()    {
101                 return wage * hours;
102         }
103
104         public String toString()    {
105                 return "Trabajador por hora: " +
106             getFirstName() + ' ' + getLastName();
107         }
108 }
109
110 // Clase CommissionWorker derivada de Employee
111 public final class CommissionWorker extends Employee {
112         private double salary;      // salario base por semana
113         private double commission;  // monto por producto vendido
114         private int quantity;       // total de productos vendidos por semana
115
116         public CommissionWorker( String first, String last,
117                             double s, double c, int q)   {
```

```
118                 super( first, last );
119                 setSalary( s );
120                 setCommission( c );
121                 setQuantity( q );
122         }
123
124         public void setSalary( double s )   {
125                 salary = ( s > 0 ? s : 0 );
126         }
127
128         public void setCommission( double c )    {
129                 commission = ( c > 0 ? c : 0 );
130         }
131
132         public void setQuantity( int q )   {
133                 quantity = ( q > 0 ? q : 0 );
134         }
135
136         public double earnings()   {
137                 return salary + commission * quantity;
138         }
139
140         public String toString()   {
141                 return "Trabajador por Comision : " +
142              getFirstName() + ' ' + getLastName();
143         }
144 }
145
146 // Programa de ejemplo Polimorfismo
147 public class Polimorfismo {
148         public static void main( String rgs[] ) {
149                 Employee ref;  // referencia de clase base
150                 Boss b;
151                 CommissionWorker c;
152                 PieceWorker p;
153                 HourlyWorker h;
154                 b = new Boss( "Alan", "Turing", 800.00 );
155                 c = new CommissionWorker( "Ada", "Lovelace",
156                                 400.0, 3.0, 150);
157                 p = new PieceWorker( "Grace", "Hopper", 2.5, 200 );
158                 h = new HourlyWorker( "James", "Gosling", 13.75, 40 );
159
```

```
160                 ref = b;  // referencia de superclase a objeto de subclase
161                 System.out.println( ref.toString() + " ganó $" +
162                     ref.earnings() );
163                 System.out.println( b.toString() + " ganó $" +
164                     b.earnings() );
165 
166                 ref = c;  // referencia de superclase a objeto de subclase
167                 System.out.println( ref.toString() + " ganó $" +
168                     ref.earnings() );
169                 System.out.println( c.toString() + " ganó $" +
170                     c.earnings() );
171 
172                 ref = p;  // referencia de superclase a objeto de subclase
173                 System.out.println( ref.toString() + " ganó $" +
174                     ref.earnings() );
175                 System.out.println( p.toString() + " ganó $" +
176                     p.earnings() );
177 
178                 ref = h;  // referencia de superclase a objeto de subclase
179                 System.out.println( ref.toString() + " ganó $" +
180                     ref.earnings() );
181                 System.out.println( h.toString() + " ganó $" +
182                     h.earnings() );
183         }
184 }
```

Listing 82. Ejemplo de clase abstracta y polimorfismo en Java. Programa de cálculo de salario

Ejemplo:**Programa de figuras geométricas con una clase abstracta Shape (Forma)**

```
1  // Definicion de clase base abstracta Shape
2  public abstract class Shape {
3  
4          public double area() {
5                  return 0.0;
6          }
7  
8          public double volume() {
9                  return 0.0;
10         }
11 
12         public abstract String getName();
```

```
13 }
14
15 // Definicion de clase Point
16 public class Point extends Shape {
17         protected double x, y; // coordenadas del punto
18
19         public Point( double a, double b ) { setPoint( a, b ); }
20
21         public void setPoint( double a, double b )   {
22                 x = a;
23                 y = b;
24         }
25
26         public double getX() { return x; }
27
28         public double getY() { return y; }
29
30         public String toString()
31       { return "[" + x + ", " + y + "]"; }
32
33         public String getName() {
34                 return "Punto";
35         }
36 }
37
38 // Definicion de clase Circle
39 public class Circle extends Point {  // hereda de Point
40         protected double radius;
41
42         public Circle()   {
43                 super( 0, 0 );
44                 setRadius( 0 );
45         }
46
47         public Circle( double r, double a, double b )   {
48                 super( a, b );
49                 setRadius( r );
50         }
51
52         public void setRadius( double r )
53       { radius = ( r >= 0 ? r : 0 ); }
54
```

```
55          public double getRadius() { return radius; }
56
57          public double area() { return 3.14159 * radius * radius; }
58
59          public String toString()
60        { return "Centro = " + super.toString() +
61                 "; Radio = " + radius; }
62
63          public String getName() {
64                   return "Circulo";
65          }
66  }
67
68  // Definicion de clase Cylinder
69  public class Cylinder extends Circle {
70          protected double height;  // altura del cilindro
71
72          public Cylinder( double h, double r, double a, double b )         {
73                   super( r, a, b );
74                   setHeight( h );
75          }
76
77          public void setHeight( double h ){
78                   height = ( h >= 0 ? h : 0 );
79          }
80
81          public double getHeight() {
82                   return height;
83          }
84
85          public double area()   {
86                   return 2 * super.area() +
87                            2 * 3.14159 * radius * height;
88          }
89
90          public double volume() {
91                   return super.area() * height;
92          }
93
94          public String toString(){
95                   return super.toString() + "; Altura = " + height;
96          }
```

```
97
98          public String getName() {
99                  return "Cilindro";
100         }
101 }
102
103 // Codigo de prueba
104 public class Polimorfismo02 {
105
106         public static void main (String args []) {
107                 Point point;
108                 Circle circle;
109                 Cylinder cylinder;
110                 Shape arrayOfShapes[];
111
112                 point = new Point( 7, 11 );
113                 circle = new Circle( 3.5, 22, 8 );
114                 cylinder = new Cylinder( 10, 3.3, 10, 10 );
115
116                 arrayOfShapes = new Shape[ 3 ];
117
118                 // asigno las referencias de los objetos de subclase
119                 // a un arreglo de superclase
120                 arrayOfShapes[ 0 ] = point;
121                 arrayOfShapes[ 1 ] = circle;
122                 arrayOfShapes[ 2 ] = cylinder;
123
124                 System.out.println( point.getName() + ": " + point.toString());
125
126                 System.out.println( circle.getName() + ": " + circle.toString());
127
128                 System.out.println( cylinder.getName() + ": " + cylinder.toString());
129
130                 for ( int i = 0; i < 3; i++ ) {
131                         System.out.println( arrayOfShapes[ i ].getName() +
132                                         ": " + arrayOfShapes[ i ].toString());
133                         System.out.println( "Area = " + arrayOfShapes[ i ].area() );
134                         System.out.println( "Volume = " + arrayOfShapes[ i ].volume() );
135                 }
136         }
137 }
```

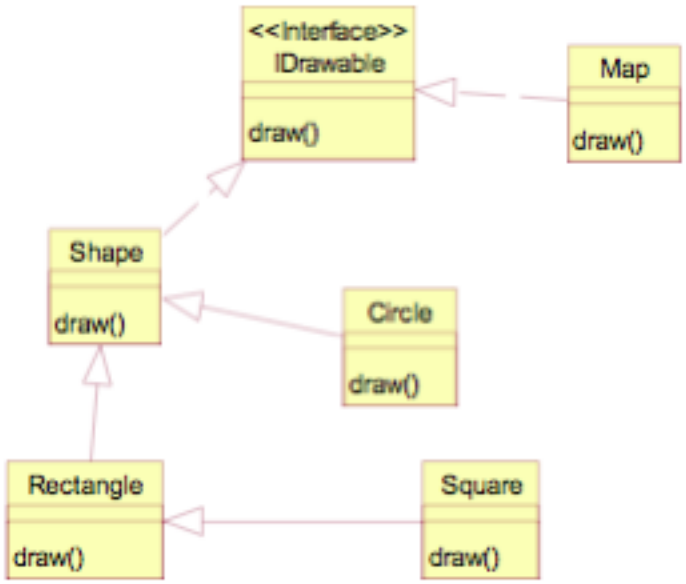


Fig. 17.1. Ejemplo de polimorfismo con interfaces en Java

Listing 83. Ejemplo de polimorfismo en Java. Programa de figuras geométricas con una clase abstracta Shape (Forma)

Clase abstracta o interfaz

17.2.2. Ejemplo de Polimorfismo con una Interfaz en Java

Los programas anteriores estaban basados en clases y clases abstractas. Sin embargo, también es posible tener variables de referencia a interfaces, a través de las cuales se implemente el polimorfismo. El siguiente programa muestra otra estructura clásica de clases "gráficas", todas contienen su propia implementación de *draw*(), y son organizadas en dos arreglos de ejemplo: uno de la clase principal, y el segundo del tipo de la interfaz.

Ejemplo:

```
1  //programa Polimorfismo
2  interface IDrawable {
3      void draw();
4  }
5
6  class Shape implements IDrawable {
7      public void draw() { System.out.println("Dibujando Figura."); }
8  }
9
10 class Circle extends Shape {
```

```
11      public void draw() { System.out.println("Dibujando Circulo."); }
12  }
13
14  class Rectangle extends Shape {
15      public void draw() { System.out.println("Dibujando Rectangulo."); }
16  }
17
18  class Square extends Rectangle {
19      public void draw() { System.out.println("Dibujando cuadrado."); }
20  }
21
22  class Map implements IDrawable {
23      public void draw() { System.out.println("Dibujando mapa."); }
24  }
25
26  public class Polimorfismo03 {
27      public static void main(String args[]) {
28          Shape[] shapes = {new Circle(), new Rectangle(), new Square()};
29          IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()};
30
31          System.out.println("Dibujando figuras:");
32          for (int i = 0; i < shapes.length; i++)
33              shapes[i].draw();
34
35          System.out.println("Dibujando elementos dibujables:");
36          for (int i = 0; i < drawables.length; i++)
37              drawables[i].draw();
38      }
39  }
```

Listing 84. Ejemplo de polimorfismo con interfaces en Java.

17.3. Polimorfismo en Python

Debido a que Python es un lenguaje tipado dinámicamente, el polimorfismo se ejecuta en automático. De hecho, cada operación es una operación polimórfica en Python. Cualquier método aplicada a un objeto funcionará mientras la clase del objeto la soporte, siendo determinado en tiempo de ejecución si es posible llevarse a cabo la operación.

Ejemplo:

```
1  # Ejemplo de polimorfismo en Python
2  class base:
3      def quien(self):
4          print('base')
5  class primera (base):
6      def quien(self):
7          print('primera')
8  class segunda (base):
9      def quien(self):
10         print('segunda')
11 class tercera (base):
12     def foo(self):
13         return
14 #script de ejecución
15 objBase = base()
16 obj1 = primera()
17 obj2 = segunda()
18 obj3 = tercera()
19
20 pBase= objBase
21 pBase.quien()
22 pBase= obj1
23 pBase.quien()
24 pBase= obj2
25 pBase.quien()
26 pBase= obj3
27 pBase.quien()
```

Listing 85. Ejemplo de polimorfismo en Python.

17.3.1. Clases abstractas y polimorfismo en Python

El concepto de clases abstractas no esta implementado directamente en Python. Sin embargo, si proporciona un módulo para clases bases abstractas (ABC – Abstract Base

Class)[3]. El módulo proporciona una metaclase[4] ABCMeta y una clase de ayuda ABC[5].

Sintaxis

```
from abc import ABC

class <MiClaseAbstracta>(ABC):
    <resto de la clase>
```

Ejemplo:

```
1  #polimorfismo y clase abstracta en Python
2  from abc import ABC, abstractmethod
3
4  class Shape(ABC):
5      def area(self):
6          return 0
7      def volume(self):
8          return 0
9
10     @abstractmethod
11     def getName(self):
12         pass
13
14 class Point (Shape):
15     def __init__(self, a, b):
16         self.x = a
17         self.y = b
18
19     def setPoint(self, a, b):
20         self.x = a
21         self.y = b
22
23     def getX(self):
24         return self.x
25
```

[3] https://docs.python.org/3/library/abc.html
[4] https://realpython.com/python-metaclasses/
[5] El concepto de metaclases en Python va más allá del alcance del curso.

```
26      def getY(self):
27          return self.y
28
29      def toString(self):
30          return "[" + str(self.x) + ", " + str(self.y) + "]"
31
32      def getName(self):
33          return 'Punto'
34
35  class Circle (Point):
36      def __init__(self, r, a, b):
37          super().__init__(a, b)
38          self.radius = r
39
40      def setRadius(self, r):
41          if r>=0 :
42              self.radius = r
43          else:
44              self.radius = 0
45
46      def area(self):
47          return 3.14159 * self.radius * self.radius
48
49      def toString(self):
50          return "Centro = " + super().toString() + "; Radio = " + str(self.radius)
51
52      def getName(self):
53          return 'Círculo'
54
55  class Cylinder (Circle):
56      def __init__(self, h, r, a, b):
57          super().__init__(r, a, b)
58          self.height = h
59
60      def setHeight(self, h):
61          if h>=0:
62              self.height  = h
63          else:
64              self.height = 0
65
66      def getHeight(self):
67          return self.height
```

```
68
69      def area(self):
70          return 2 * super().area() + 2 * 3.14159 * self.radius * self.height
71
72      def volume(self):
73          return super().area() * self.height
74
75      def toString(self):
76          return super().toString() + "; Altura = " + str(self.height)
77
78      def getName(self):
79          return 'Cilindro'
80
81  #script de prueba
82  point = Point(7, 11)
83  circle = Circle(3.5, 22, 8)
84  cylinder = Cylinder(10, 3.3, 10, 10)
85  arrayOfShapes = [point, circle, cylinder]
86
87  print(point.getName() + ": " + point.toString())
88  print(circle.getName() + ": " + circle.toString())
89  print(cylinder.getName() + ": " + cylinder.toString())
90
91  for sh in arrayOfShapes:
92      print(sh.getName() + ": " + sh.toString())
93      print( "Area = " + str(sh.area()) )
94      print( "Volume = " + str(sh.volume()) )
```

Listing 86. Ejemplo de clases abstractas y polimorfismo en Python.

Capítulo 18

Polimorfismo paramétrico: programación genérica

La programación genérica favorece la reutilización de código, permitiendo que se generen objetos específicos para un tipo a partir de **clases genéricas**. Las clases genéricas son conocidas también como **plantillas de clase** o **clases parametrizadas**.

18.1. Plantillas de clase en C++

Antes se mencionó el uso de plantillas en C++ aplicado a funciones. El concepto de plantillas es aplicable también a la programación orientada a objetos en C++a través de **plantillas de clase**.

El uso de plantillas de clase no es diferente al uso de plantillas en operaciones no orientadas a objetos:

Sintaxis

```
template <class T>

o

template <typename T>
```

Veamos el ejemplos clásicos aprovechando el uso de plantillas.
Ejemplo:

```
1  #include <iostream>
2
3  using namespace std;
4
5  template <class T, class U>
6  class Pair {
7  private:
8      T first;
9      U second;
10 public:
11     Pair(T f, U second) {
12         first=f;
13         this->second=second;
14
15     }
16
17     T getFirst() {
18         return first;
19
20     }
```

```
21      U getSecond() {
22          return second;
23      }
24  };
25
26  int main() {
27      Pair<float, int> *pair = new Pair<float, int>(10.5,2);
28
29      cout<<"Obten el primer elemento:" << pair->getFirst()<<endl;
30      cout<<"Obten el segundo elemento:"<< pair->getSecond()<<endl;
31      return 0;
32  }
```

Listing 87. Ejemplo de plantillas en C++.

Ejemplo:

```
1   // stack.h
2   // Clase de plantilla Pila
3   #ifndef STACK_H_
4   #define STACK_H_
5
6   template< class T >
7   class Stack {
8   public:
9           Stack( int = 10 );
10          ~Stack() { delete [] stackPtr; }
11          char push( const T& );
12          char pop( T& );
13  private:
14          int size;
15          int top;
16          T *stackPtr;
17
18          char isEmpty() const { return top == -1; }
19          char isFull() const { return top == size - 1; }
20  };
21
22
23  template< class T >
24  Stack< T >::Stack( int s )
25  {
```

```
26          size = s > 0 ? s : 10;
27          top = -1;
28          stackPtr = new T[ size ];
29  }
30
31  template< class T >
32  char Stack< T >::push( const T &pushValue )
33  {
34          if ( !isFull() ) {
35                  stackPtr[ ++top ] = pushValue;
36                  return 1;
37          }
38          return 0;
39  }
40
41  template< class T >
42  char Stack< T >::pop( T &popValue )
43  {
44          if ( !isEmpty() ) {
45                  popValue = stackPtr[ top-- ];
46                  return 1;
47          }
48          return 0;
49  }
50
51  #endif /*STACK_H_*/
52
53  // Ejemplo uso de plantillas de clase
54  #include <iostream>
55  #include "stack.h"
56
57  using namespace std;
58
59  int main() {
60          Stack< double > doubleStack( 5 );
61          double f = 1.1;
62          cout << "Insertando elementos en doubleStack \n";
63
64          while ( doubleStack.push( f ) ) {
65                  cout << f << ' ';
66                  f += 1.1;
67          }
```

```
68
69          cout << "\nLa pila está llena. No se puede insertar el elemento " << f
70                    << "\n\nSacando elementos de doubleStack\n";
71
72          while ( doubleStack.pop( f ) )
73                  cout << f << ' ';
74
75          cout << "\nLa pila está vacía. No se pueden eliminar más elementos\n";
76
77          Stack< int > intStack;
78          int i = 1;
79          cout << "\nInsertando elementos en intStack\n";
80
81          while ( intStack.push( i ) ) {
82                  cout << i << ' ';
83                  ++i;
84          }
85
86          cout << "\nLa pila está llena.  " << i
87                    << "\n\nSacando elementos de intStack\n";
88
89          while ( intStack.pop( i ) )
90                  cout << i << ' ';
91
92          cout << "\nLa pila está vacía. No se pueden eliminar más elementos \n";
93          return 0;
94  }
```

Listing 88. Ejemplo, una pila con plantillas en C++.

Las plantillas de clase ayudan a la reutilización de código, al permitir varias versiones de clases para un tipo de dato a partir de clases genéricas. A estas clases específicas se les conoce como **clases de plantilla**.

Concepto
Una clase de plantilla es entonces como una instanciación de una plantilla de clase.

Con respecto a la herencia en combinación con el uso de plantillas, se deben tener en cuenta las siguientes situaciones[14]:

- Una plantilla de clase se puede derivar de una clase de plantilla.

- Una plantilla de clase se puede derivar de una clase que no sea plantilla.
- Una clase de plantilla se puede derivar de una plantilla de clase.
- Una clase que no sea de plantilla se puede derivar de una plantilla de clase.

En cuanto a los miembros estáticos, cada clase de plantilla que se crea a partir de una plantilla de clases mantiene sus propias copias de los miembros estáticos.

18.2. Standard Template Library (STL)

Las plantillas de clase son una herramienta muy poderosa en C++. Esto ha llevado a desarrollar lo que se conoce como STL. STL es el acrónimo de *Standard Template Library*, y es una biblioteca de C++ que proporciona un conjunto de clases contenedoras, iteradores y de algoritmos genéricos:

- Las clases contenedoras incluyen vectores, listas, deques, conjuntos, multiconjuntos, multimapas, pilas, colas y colas de prioridad.
- Los iteradores son generalizaciones de apuntadores: son objetos que apuntan a otros objetos. Son usados normalmente para iterar sobre un conjunto de objetos. Los iteradores son importantes porque son típicamente usados como interfaces entre las clases contenedoras y los algoritmos.
- Los algoritmos genéricos incluyen un amplio rango de algoritmos fundamentales para los más comunes tipos de manipulación de datos, como ordenamiento, búsqueda, copiado y transformación.
- STL es una biblioteca estándar de ANSI/ISO desde julio de 1994.

La STL está altamente parametrizada, por lo que casi cada componente en la STL es una plantilla[15] . Podemos usar por ejemplo la plantilla $vector < T >$ para hacer uso de vectores sin necesidad de preocuparnos del manejo de memoria:

```
1     vector<int> v(3);      // Declara un vector de 3 elementos.
2         v[0] = 7;
3         v[1] = v[0] + 3;
4         v[2] = v[0] + v[1];    // v[0] == 7, v[1] == 10, v[2] == 17
```

Los algoritmos proporcionados por la STL ayudan a manipular los datos de los contenedores[15]. Por ejemplo, podemos invertir el orden de los elementos de un vector, usando el algoritmo *reverse*():

```
reverse(v.begin(), v.end()); // v[0]==17, v[1]==10, v[2]==7
```

Ejemplo:

```
#ifndef STACK_HPP_
#define STACK_HPP_
#include <vector>

template <typename T>
class Stack {
  private:
    std::vector<T> elems;     // elementos

  public:
    void push(T const&);
    void pop();
    T top() const;            // regresa elemento en el tope
    bool empty() const {      // regresa si la pila esta vacia
        return elems.empty();
    }
};

template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem);    // añade una copia de elem
}

template<typename T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        std::cout<<"Stack<>::pop(): pila vacia";
        return;
    }
    elems.pop_back();         // remueve el ultimo elemento
}

template <typename T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        std::cout<<"Stack<>::top(): pila vacia";
    }
    return elems.back();      // regresa copia del elemento en el tope
```

```
42 }
43 #endif /*STACK_HPP_*/
44
45 #include <iostream>
46 #include <string>
47 #include <cstdlib>
48 #include "stack.hpp"
49
50 int main()
51 {
52         Stack<int>         intStack;       // pila de enteros
53         Stack<std::string> stringStack;    // pila de strings
54
55         // manipulapila de enteros
56         intStack.push(7);
57         std::cout << intStack.top() << std::endl;
58
59         // manipula pila de strings
60         stringStack.push("hola");
61         std::cout << stringStack.top() << std::endl;
62         stringStack.pop();
63         stringStack.pop();
64 }
```

Listing 89. Ejemplo de STL.

18.3. Clases Genéricas en Java

Java 1.5 introdujo finalmente el uso de clases genéricas (*generics*)[16]. El uso de clases genéricas es una característica poderosa usada en otros lenguajes, siendo C++ el ejemplo más conocido que soporta programación genérica mediante el uso de plantillas o *templates*.

Sintaxis
`class NombreClase <Lista de parámetros de tipos> { ... }`

Ejemplo:

```
1  class Pair<T, U> {
2    private final T first;
3    private final U second;
4    public Pair(T first, U second) { this.first=first; this.second=second; }
5    public T getFirst() { return first; }
6    public U getSecond() { return second; }
7  }
8
9  public class PairExample {
10         public static void main(String[] args) {
11
12                 Pair<String, Integer> pair = new Pair<String, Integer>("one",2);
13
14                 // no acepta tipos de datos básicos o primitivos
15                 //Pair<String, int> pair2 = new Pair<String, Integer>("one",2);
16
17                 // siguiente linea generaría un warning de seguridad de tipos
18                 //Pair<String, Integer> pair3 = new Pair("one",2);
19
20                 System.out.println("Obtén primer elemento:" + pair.getFirst());
21                 System.out.println("Obtén segundo elemento:" + pair.getSecond());
22         }
23 }
```

Listing 90. Ejemplo de clases genéricas en Java.

Es también posible parametrizar interfaces, como se muestra a continuación.

Sintaxis
`interface NombreInterfaz <Lista de parámetros de tipos> { ... }`

Ejemplo:

```
interface IPair<T, U>{
        public T getFirst();
        public U getSecond();
}

class Pair<T, U> implements IPair<T, U>{
  private final T first;
  private final U second;
  public Pair(T first, U second) { this.first=first; this.second=second; }
  public T getFirst() { return first; }
  public U getSecond() { return second; }
}

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

                IPair<String, Integer> ipair = new Pair<String, Integer>("one",2);

                System.out.println("Obtén primer elemento:"+ipair.getFirst());
                System.out.println("Obtén segundo elemento:"+ipair.getSecond());
        }

}
```

Listing 91. Ejemplo clases e interfaces genéricas en Java.

Un requerimiento para el uso tipos genéricos en Java es que no pueden usarse tipos de datos primitivos, porque los tipos primitivos o básicos no son subclases de Object[17]. Por lo que sería ilegal por ejemplo querer instanciar $Pair < int, String >$. La ventaja es que el uso de la clase *Object* significa que solo un archivo de clase (*.class*) necesita ser generado por cada clase genérica[18].

Restriciones de las clases genéricas, ver[1].

18.4. Biblioteca de Clases Genéricas en Java

Al igual que C++ con la STL, Java tiene un conjunto de clases genéricas predefinidas. Su uso, de manera similar que con las clases genéricas definidas por el programador, no está permitido para tipos primitivos, por lo que solo objetos podrán ser contenidos. Las principales clases genéricas en Java son, como en la STL, clases contenedoras o coleccio-

[1] https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createArrays

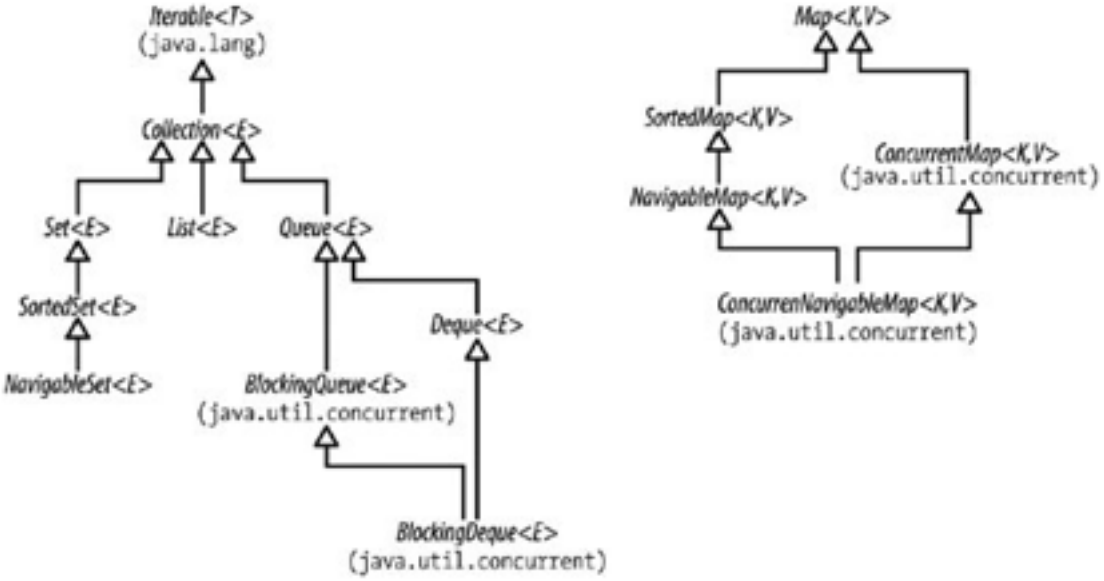


Fig. 18.1. Biblioteca genérica en Java

nes[2]. El *Java Collections Framework* (JCF) es un conjunto de interfaces y clases definidos en los paquetes *java.util* y *java.util.concurrent*.

Las interfaces del JCF son:

- *Collection*. Contiene la funcionalidad básica requerida en casi cualquier colección de objetos (con excepción de *Map*)
- *Set*. Es una colección sin duplicados, donde el orden es no significante. Sin embargo contiene un método que devuelve el conjunto ordenado (*SortedSet*).
- *Queue*. Define el comportamiento básico de una estructura de cola.
- *List*. Es una colección donde el orden es significativo, permitiendo además valores duplicados.
- *Map*. Define una colección donde un valor clave es asociado para almacenar y recuperar elementos.

La siguiente figura muestra las principales interfaces de la JCF[19]:

Los iteradores son objetos que te permiten recorrer una colección de objetos, obteniendo o removiendo elementos. Un objeto iterador implementa la interfaz *Iterator* o la interfaz

[2] Las colecciones en Java eran implementadas antes de la versión 1.5 pero sin el uso de clases genéricas. El uso de versiones anteriores de colecciones con colecciones genéricas es permitido por compatibilidad hacia atrás pero debe tenerse especial cuidado pues hay situaciones que el compilador no puede validar.

ListIterator. En general, para usar un iterador para recorrer una colección se debe: 1. Obtener un iterador al inicio de la colección llamando al método *iterator*() de la colección. 2. Definir un ciclo que haga la llamada a *hasNext*(). El ciclo iterará mientras el método sea verdadero. 3. En el ciclo, obtener cada elemento llamando al método *next*().

Ejemplo:

```
1  // Usando la interfaz Collection
2  import java.util.List;
3  import java.util.ArrayList;
4  import java.util.Collection;
5  import java.util.Iterator;
6
7  public class CollectionTest {
8          private static final String[] colors =
9         { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
10         private static final String[] removeColors =
11        { "RED", "WHITE", "BLUE" };
12
13     // crea ArrayList, añade Colors y la manipula
14     public CollectionTest() {
15             List< String > list = new ArrayList< String >();
16         List< String > removeList = new ArrayList<String>();
17
18         // añade elementos del arreglo colors a list
19         for ( String color : colors )
20              list.add( color );
21
22        // añade elementos del arreglo removeColors a removeList
23        for ( String color : removeColors )
24              removeList.add( color );
25
26        System.out.println( "ArrayList: " );
27        // despliega contenido de list
28        for ( int count = 0; count < list.size(); count++ )
29              System.out.printf( "%s ", list.get( count ) );
30
31        // remueve de list colores contenidos en removeList
32        removeColors( list, removeList );
33
34        System.out.println( "\n\nArrayList después de llamar removeColors: " );
35        // despliega contenido de list
36        for ( String color : list )
```

```
37              System.out.printf( "%s ", color );
38      } // end CollectionTest constructor
39
40      // remueve colores especificados en collection2 de collection1
41      private void removeColors(
42         Collection< String > collection1, Collection< String > collection2 ) {
43                 // obtiene iterator
44                 Iterator< String > iterator = collection1.iterator();
45
46                 // mientras colección tiene elementos
47                 while ( iterator.hasNext() )
48                           if ( collection2.contains( iterator.next() ) )
49                                iterator.remove(); // remueve color actual
50      }
51
52      public static void main( String args[] )  {
53            new CollectionTest();
54      }
55  }
```

Listing 92. Ejemplo usando la JCF en Java.

18.4.1. Ejemplos complementarios de Clases Genéricas en Java

Ejemplo de una pila genérica simple[3]:

```
1   import java.util.*;
2
3   public class PilaGenérica <T> {
4           private ArrayList<T> pila = new ArrayList<T> ();
5       private int tope = 0;
6
7       public int size () {
8                 return tope;
9       }
10
11      public void push (T elemento) {
12                pila.add (tope++, elemento);
13      }
14
```

[3]Basado en: http://cs.fit.edu/~ryan/java/programs/generic/GenericStack-java.html

```
15      public T pop () {
16          return pila.remove (--tope);
17      }
18
19      public static void main (String[] args) {
20          PilaGenérica<Integer> p = new PilaGenérica<Integer> ();
21
22          p.push (17);
23          int i = p.pop ();
24          System.out.format ("%4d%n", i);
25       }
26   }
```

Listing 93. Ejemplo de una pila genérica simple en Java.

Ejemplo de una pila genérica como lista ligada[4]:

```
1   import java.util.Iterator;
2   import java.util.NoSuchElementException;
3
4   public class Pila<Elemento> implements Iterable<Elemento> {
5       private int tamaño;           // tamaño de la pila
6       private Nodo primer;      // tope de la pila
7
8       // clase anidada Nodo
9       private class Nodo {
10          private Elemento elemento;
11          private Nodo siguiente;
12      }
13
14      // Crea una pila vacia.
15      public Pila() {
16          primer = null;
17          tamaño = 0;
18      }
19
20      // Esta vacia la pila?
21      public boolean estaVacia() {
22          return primer == null;
23      }
24
```

[4]Ejemplo basado de: http://introcs.cs.princeton.edu/java/43stack/Stack.java.html

```
25      // Regresa el número de elementos en la pila
26      public int getTamaño() {
27          return tamaño;
28      }
29
30      // Añade elemento a la pila.
31      public void push(Elemento elemento) {
32          Nodo viejoPrimer = primer;
33
34       primer = new Nodo();
35          primer.elemento = elemento;
36          primer.siguiente = viejoPrimer;
37          tamaño++;
38      }
39
40     /**
41       *Regresa el elemento en el tope de la pila y lo elimina.
42       *Lanza una excepción si no hay elemento porque la pila este vacia.*/
43      public Elemento pop() {
44          if (estaVacia())
45                  throw new RuntimeException("Pila vacía");
46          Elemento elemento = primer.elemento;    // guarda elemento para retornarlo
47          primer = primer.siguiente;  // elimina el primer nodo
48          tamaño--;
49          return elemento;                  // regresa elemento
50      }
51
52     /*** Regresa el elemento en el tope de la pila sin modificarla.
53       * Lanza una excepción si la pila esta vacía.*/
54      public Elemento ver() {
55          if (estaVacia())
56                  throw new RuntimeException("Pila vacía");
57          return primer.elemento;
58      }
59
60     /*** Regresa representación en cadena.*/
61      public String toString() {
62          StringBuilder s = new StringBuilder();
63          for (Elemento elemento : this)
64              s.append(elemento + " ");
65          return s.toString();
66      }
```

```
67      //  Regresa un iterador a la pila que itera a través de los elementos en orden LIFO
68      public Iterator<Elemento> iterator()  {
69              return new ListIterator();
70              }
71    // Iterador, no se implementa remove() dado que es opcional
72      private class ListIterator implements Iterator<Elemento> {
73          private Nodo actual = primer;
74
75          public boolean hasNext()  {
76                  return actual != null;
77          }
78
79          public void remove() {
80                  throw new UnsupportedOperationException();
81          }
82
83          public Elemento next() {
84              if (!hasNext())
85                      throw new NoSuchElementException();
86              Elemento elemento = actual.elemento;
87              actual = actual.siguiente;
88              return elemento;
89          }
90      }
91
92      public static void main(String[] args) {
93          Pila<String> s = new Pila<String>();
94
95          String elemento1 = "un texto";
96          String elemento2 = "otro elemento";
97          String elemento3 = "xxxx";
98
99          s.push(elemento1);
100         s.push(elemento2);
101         s.push(elemento3);
102
103         while (!s.estaVacia()) {
104            System.out.println(s.pop());
105            System.out.println("(" + s.getTamaño() + " elemento(s) quedan en la pila)");
106         }
107     }
108 }
```

Listing 94. Ejemplo de una pila genérica como lista ligada en Java.