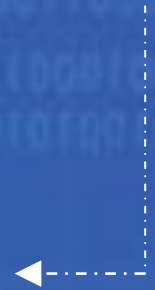


Capítulo

09



# Introducción Orientada a Objetos

Samuel Sepúlveda Cuevas

Departamento de Ingeniería de Sistemas  
Facultad de Ingeniería, Ciencias y Administración

Versión

0.9

## TEMARIO

- 9.1 ¿Qué es la POO?
- 9.2 Conceptos Básicos
- 9.3 Diferencias entre la Programación Estructurada y la POO
- 9.4 Diseño de Software con POO
- 9.5 Conceptos básicos de JAVA
- 9.6 Programando en Java usando POO
- 9.7 Ejercicios resueltos
- 9.8 Ejercicios propuestos
- 9.9 Comentarios finales
- 9.10 Referencias

# Introducción a la Programación Orientada a Objetos

Hasta ahora el esfuerzo de este texto ha estado puesto en tratar de que tú puedas entender los principios básicos que hay tras el funcionamiento del computador y la creación de programas computacionales, en particular con los lenguajes C, C++ y JAVA. Se debe destacar una cosa, los conceptos, metodologías y ejemplos han sido desarrollados basándose en el paradigma de programación estructurada (ver el Capítulo XXX), en cambio el objetivo central de este capítulo en particular será el de mostrarte los fundamentos teóricos de una forma un tanto diferente de “pensar y construir” un programa computacional. Como de seguro leíste atentamente el título de este capítulo, ya sabes de lo que estamos hablando, se trata de la Programación Orientada a Objetos (de ahora en adelante usaremos la sigla POO). En las siguientes secciones se mostrarán conceptos y ejemplos que usaremos para responder preguntas como ¿Qué es la POO? ¿Cuáles son las diferencias entre la programación estructurada y la POO? ¿Cómo se programa con POO y JAVA?

## 9.1 ¿Qué es la POO?

Antes de intentar dar una definición de lo que es o se entiende por POO, es importante tener claro que ésta pretende modelar y luego construir un programa computacional que esté organizado de la misma manera que el problema que estamos solucionando. Lo anterior es relevante pues el “mundo real” donde están los problemas que deseamos solucionar se encuentra compuesto por objetos que se relacionan entre sí, no por funciones o estructuras de datos. Ahora se “debe pensar” en base a los objetos y a como estos se relacionan e intentar desde ahí solucionar el problema.

*“...Cualquier programador lo suficientemente persistente y empeinado conseguirá escribir código al estilo Fortran ó Cobol en cualquier lenguaje de programación que utilice....”*

Anónimo

Desde los inicios de la computación y a medida que transcurre el tiempo, los usuarios exigen más de los sistemas y aplicaciones de software. Crece así la complejidad de los programas y sistemas. Como solución para disminuir dicha complejidad, surge la Programación Estructurada la cual ordena la construcción de programas y sistemas, modulariza su desarrollo, pero ante todo, la programación sigue siendo una secuencia de instrucciones. Así, si aumenta la complejidad, aumentan también los esfuerzos por diseñar y depurar los programas.

El reconocido colapso al que llegaron las situaciones expuestas, se constituye en uno de los detonantes de la conocida Crisis del Software. Así comienza a ponerse atención a un nuevo paradigma, la Orientación a Objetos y la forma de construir programas bajo este paradigma es lo que llamamos la **Programación Orientada a Objetos (POO)**.

## 9.2 Conceptos Básicos

A continuación se presentan una serie de conceptos que definen y permiten entender el cómo funciona la POO.

### a. Objeto:

El concepto básico de la POO es el de objeto, por lo tanto resulta conveniente desde ya dar una respuesta a la pregunta ¿Qué es un objeto? Inicialmente se puede decir que un objeto, según la Real Academia de la Lengua Española (RAE), es “Todo lo que puede ser materia de conocimiento o sensibilidad de parte del sujeto, incluso este mismo”. Lo anterior no está planteado desde una perspectiva computacional, pues si así fuese, podríamos decir que un objeto es:

- Cualquier cosa, real o abstracta, que posea un estado interno y un conjunto de operaciones o métodos.
- Una colección de operaciones encapsuladas que comparten un estado común.

Una forma de representarlo, independiente de sus características físicas podría ser la que se muestra en la Figura 9.0.

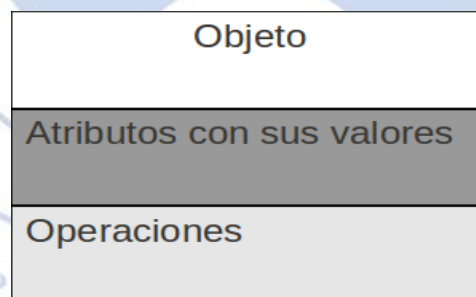


Figura 9.0 Representación de un Objeto, conteniendo su nombre, atributos y operaciones.

### b. Estado:

De las definiciones técnicas anteriores, se observa un elemento común, este es el de estado, que se define como el valor de los atributos del objeto en un cierto instante.

### c. Comportamiento:

Se relaciona con las operaciones y como éstas hacen que se comporte el objeto ante una acción, la cual es provocada por la invocación a un método de la clase.

### d. Atributos:

Son los elementos que definen un objeto, son sus “propiedades”. Conocidas en la programación tradicional como las variables. De lo anterior podemos concluir que un atributo debe entonces definirse con un nombre que lo

represente y un tipo de datos asociado.

**Ejemplo 9.1:**

Definición de un atributo de tipo String llamado numeroCuenta, en lenguaje Java

```
class cuentaCorriente {  
    String numeroCuenta;  
    ...  
}
```

Atributo de tipo String,  
llamado numeroCuenta

Otro elemento a considerar además del tipo de datos y nombre de cada atributo es lo relativo a su nivel de encapsulamiento, concepto que se trata en el punto (g.) de este apartado.

**e. Métodos:**

También llamadas operaciones, son los que determinan aquello que se puede hacer con el objeto. Conocidos en la programación tradicional como las funciones o procedimientos. Puede entenderse como un conjunto de funciones pertenecientes a una clase, que actúan y modifican los atributos del objeto.

**f. Clase:**

Así como un objeto es un concepto con un grado de abstracción mayor que las comunes estructuras de datos, es posible hacer una abstracción aún mayor e introducir el concepto de Clase, entendiéndola como una descripción generalizada que hace referencia a una colección de objetos que poseen las mismas características.

Para nuestro contexto entenderemos la **abstracción** como una aproximación o abordaje del diseño que hace hincapié en los aspectos más importantes de algo, sin preocuparse por los detalles menos importantes.

**Estado de un Objeto y la instanciación de una Clase**

Luego, a partir de los conceptos hasta ahora mencionados, se puede decir que un objeto no es sino una **instancia** de una **clase**.

- Pensar en un **Objeto** como “caso particular” de una **Clase**.
- Los atributos poseen valores específicos en un instante de tiempo, que lo identifican como un único objeto.
- Lo anterior indica que los objetos son “dinámicos”, pues su estado puede cambiar en el tiempo.

**Ejemplo 9.2:**

Objeto de la clase automóvil, un automóvil Fiat 600

Atributo color, hoy día con el valor rojo. Pero que sucedería si decidimos que este automóvil debe ser repintado y mañana el atributo color tenga que tomar el valor azul.



Figura 9.1 Objeto Fiat 600 de color rojo

Figura 9.1 Objeto Fiat 600 de color azul

De las figuras 9.1 y 9.2 se puede observar el concepto de estado del objeto y como éste cambia, enfatizando las características dinámicas de éste, pues en un momento el Fiat 600 es de color rojo y luego mediante la operación Pintar, es ahora de color azul.

A continuación, se muestran 2 ejemplos que nos permiten representar objetos, en diferentes contextos, que quedan definidos a partir de los atributos y los valores que estos tienen en un momento determinado (el estado del objeto), así como las operaciones que nos indican lo que es posible hacer con el objeto (comportamiento del objeto). En la Fig. 9.3 se ilustra un ejemplo de objeto tipo automóvil de una marca y modelo particular, con cada uno de sus atributos con determinados valores. En la Fig9.4 se ilustra un ejemplo de un objeto de tipo lista enlazada de productos\*, donde igualmente los atributos tienen determinados valores en un momento determinado.

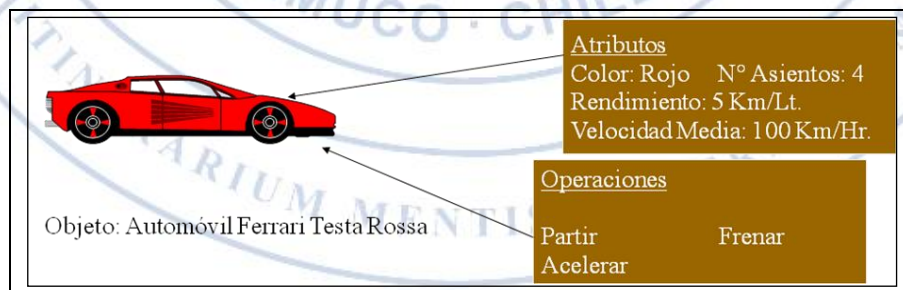


Figura 9.3 Ejemplo de un objeto automóvil Ferrari Testa Rossa, sus atributos y operaciones.



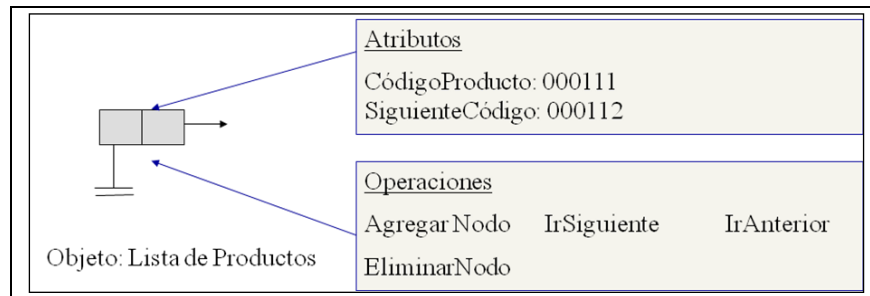


Figura 9.4 Ejemplo de un objeto lista enlazada de productos, sus atributos y operaciones.

La anterior representación gráfica mostrada en las figuras es ahora mostrada de una forma que permite abstraernos de aspectos gráficos y concentrarnos en los aspectos básicos que definen los objetos antes mencionados.

Automóvil Ferrari Testa Rossa
color = rojo asientos = 4 rendimiento_km_ltr = 5 velocidad_media_km_hr = 100
Partir() Frenar() Acelerar()

Figura 9.5 Representación del Objeto Ferrari Testa Rossa

Lista enlazada Lista de Productos
código_producto = 00011 siguiente_código = 00025
agregarNodo() irSiguiente() irAnterior() EliminarNodo()

Figura 9.6 Representación del Objeto Lista de Productos

Lo que se muestra en la Figura 9.5 es un ejemplo de lo dicho en las definiciones **a. y f.**, pues en la columna izquierda se pueden observar varios objetos del tipo persona, donde cada uno de ellos tiene diferentes atributos que los distinguen entre sí como objetos únicos, por ejemplo: *nombre, edad, color de ojos, color de pelo, entre otros*. Es decir son objetos con valores particulares para cada uno de los atributos mencionados. Por otra parte en la columna de derecha de la misma figura se puede ver que haciendo omisión de los valores particulares de cada atributo para cada objeto, nos hemos dado cuenta que todos los objetos comparten efectivamente características comunes y que son todos ellos un tipo particular de persona, de ahí entonces que abstrayéndonos de los detalles, se defina la clase PERSONA que representa genéricamente a todos los objetos que son del mismo tipo.



Figura 9.7 Abstracción de los detalles de los objetos para determinar la clase.

#### g. Encapsulamiento:

Esta característica fundamental de la POO permite a los objetos crear una cápsula a su alrededor, convirtiéndose en una “caja negra” para el resto de los objetos, ocultando la información interna que manipula. Lo anterior permite modificar el código interno de alguna(s) operación(es) y si lo hacemos respetando los argumentos definidos para las entradas y las salidas de dicha operación, es posible modificar y optimizarlo en forma constante, sin efectos laterales o secundarios.

La figura 9.8 ilustra el hecho que gracias al encapsulamiento lo que se busca es proteger tanto como sea posible la integridad de los atributos del objeto y no permitir que estos puedan ser accedidos directamente. De lo anterior es que entonces se definen un conjunto de operaciones con *acceso público*, que permiten acceder a los atributos del objeto. En la figura 9.9 se intenta explicar una de las ventajas del encapsulamiento, pues el código asociado a la operación Suma(n1, n2) podemos modificarlo de acuerdo a nuestras necesidades y si tenemos cuidado en respetar las entradas y salidas que esta operación usa, el resto de los objetos que usen Suma(n1, n2) no se habrán enterado (ni les interesaría hacerlo) de los cambios o mejoras que realicemos al interior de su código.

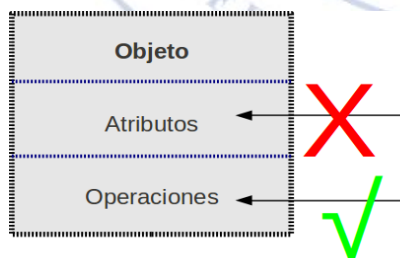


Figura 9.8 Protección del acceso a los Atributos del objeto.

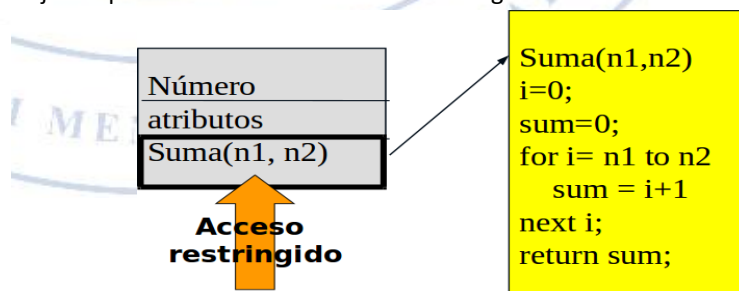


Figura 9.9 Encapsulamiento de una operación y modificación de su código fuente.

### Acceso a los elementos (atributos u operaciones) de una clase

- **Acceso público:** Cualquier miembro público de una clase es accesible desde cualquier parte donde sea accesible el propio objeto.
- **Acceso privado:** Los miembros privados de una clase sólo son accesibles por los propios miembros de la clase y en general por objetos de la misma clase, pero no desde funciones externas o desde funciones de clases derivadas.
- **Acceso protegido:** Con respecto a las funciones externas, es equivalente al acceso privado, pero con respecto a las clases derivadas se comporta como público.

#### **h. Herencia:**

Así como una clase representa genéricamente a un grupo de objetos que comparten características comunes, la herencia permite que varias clases compartan aquello que tienen en común y no repetirlo en cada clase. Consiste en propagar atributos y operaciones a través de las subclases definidas a partir de una clase común.

Nos permite crear estructuras jerárquicas de clases donde es posible la creación de subclases, que incluyan nuevas operaciones y atributos que redefinen los objetos. Estas subclases permiten así, crear, modificar o inhabilitar propiedades, aumentando de esta manera la especialización de la nueva clase.

En la figura 9.10 se muestra un ejemplo del concepto de herencia, donde se tiene una clase llamada *Vehículo* (*superclase*), la cual permite representar de forma genérica a cualquier tipo de vehículo, a partir de la cual a su vez pueden originarse tipos especiales de vehículos, en este ejemplo se tienen las *subclases*: *Sedán*, *Camión* y *Furgón*, cada una de ellas con atributos y operaciones comunes, pero a la vez definen nuevos atributos u operaciones que permiten identificarlo como un tipo particular de vehículo. Entonces es posible decir que un objeto perteneciente a la clase *Sedán* es también un objeto de tipo *Vehículo* o también decir que es una especialización de la superclase, dada la relación de herencia entre ambas clases.

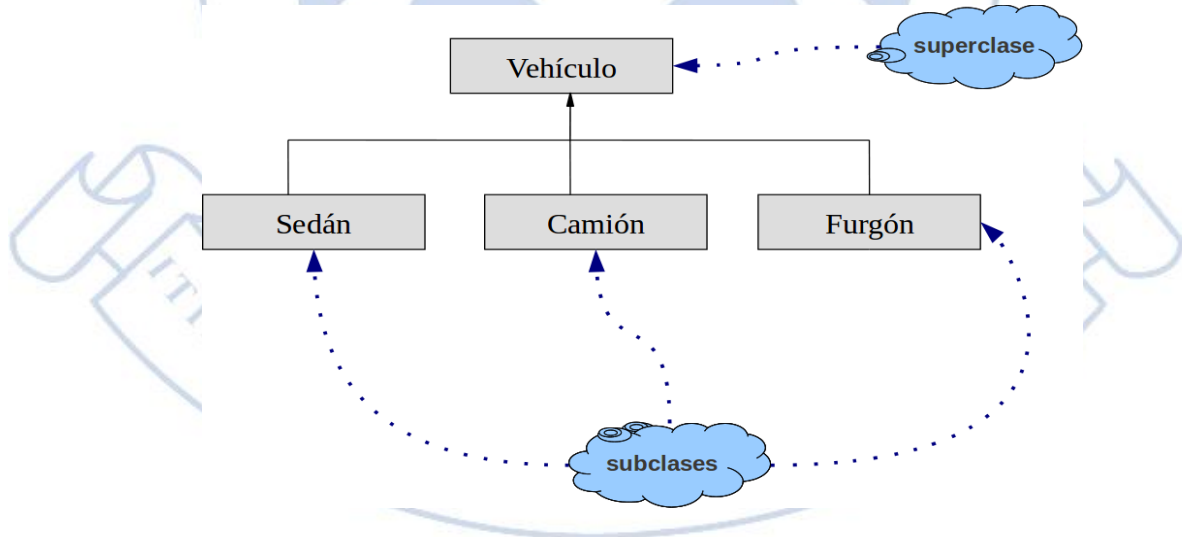


Figura 9.10 Ejemplo de herencia, que muestra la relación entre la clase padre (superclase) y las clases hijas (subclases).

Para reforzar el concepto de herencia, se presenta la figura 9.11, donde se muestran tres clases que ilustran la relación existente entre las súper y subclases, detallando como cada una de las especializaciones de la superclase, pueden agregar nuevos tributos y operaciones potenciando y agregando así más funcionalidades a las subclases. Del mismo ejemplo se puede decir que ambas subclases son especializaciones de la superclase *Persona* y por lo tanto son también tipos particulares de *Persona*, pues agregan atributos y operaciones particulares, ya que tanto la clase *Alumno* como la clase *Trabajador*, comparten los elementos comunes que los identifican como *Persona* (rut, nombre, colorPelo, colorOjos).



Así también se debe notar que ambas subclases comparten las operaciones de la clase Persona, pero que valiéndose del encapsulamiento, cada subclase puede adaptar según sus necesidades. Lo anterior implica que si bien la operación *mostrarDatos()* está presente tanto en la superclase como en las subclases, el código interno no necesariamente es el mismo.

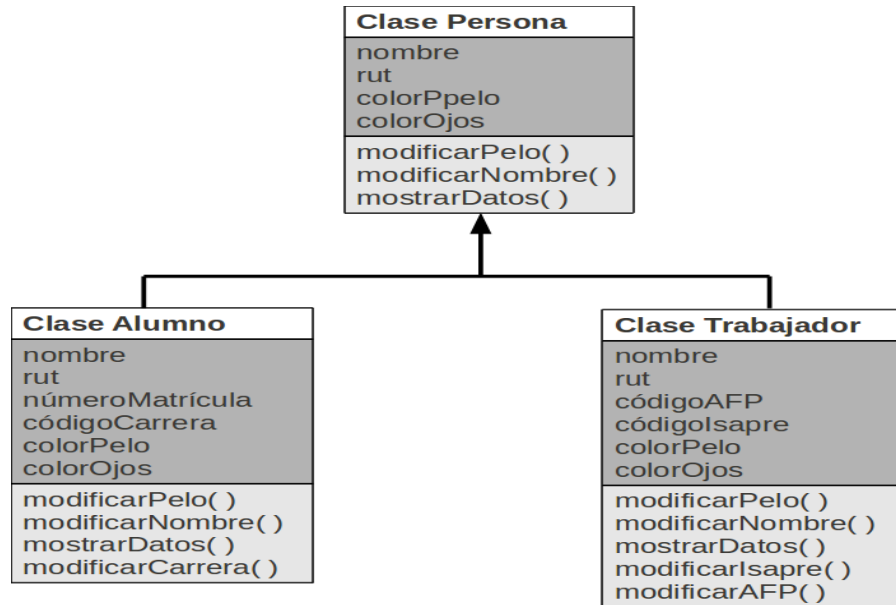


Figura 9.11 Detalle de la herencia, se agregan nuevos atributos y operaciones en las subclases.

### Ejemplo 9.3:

Efectos que se pueden lograr aplicando Herencia y Encapsulamiento.

La Figura 9.12 muestra el detalle de la operación *mostrarDatos()* para cada una de las clases del ejemplo, se debe destacar que cada subclase ha especializado haciendo uso de la herencia, dicha operación, adaptándola a sus propias necesidades.

Por otro lado y gracias al encapsulamiento, dicha especialización se puede lograr modificando el código interno de dicha operación en cada una de las subclases, sin que el resto de los elementos del programa que los lleve a utilizar se tenga que enterar de esto.

```

// Operación mostrarDatos de la clase Persona
public void mostrarDatos() {
    System.out.println("Nombre:" + this.nombre);
    System.out.println("Rut:" + this.rut);
    System.out.println("Color de ojos:" + this.colorOjos);
    System.out.println("Color de pelo:" + this.colorPelo);
}

// Operación mostrarDatos de la clase Alumno
public void mostrarDatos() {
    System.out.println("Nombre:" + this.nombre);
    System.out.println("Rut:" + this.rut);
    System.out.println("Color de ojos:" + this.colorOjos);
    System.out.println("Color de pelo:" + this.colorPelo);
    System.out.println("Código carrera:" + this.codigoCarrera);
    System.out.println("Número Matrícula:" + this.numeroMatricula);
}

// Operación mostrarDatos de la clase Trabajador
public void mostrarDatos() {
    System.out.println("Nombre:" + this.nombre);
    System.out.println("Rut:" + this.rut);
    System.out.println("Color de ojos:" + this.colorOjos);
    System.out.println("Color de pelo:" + this.colorPelo);
    System.out.println("Código AFP:" + this.codigoAFP);
    System.out.println("Código ISAPRE:" + this.codigoIsapre);
}
  
```

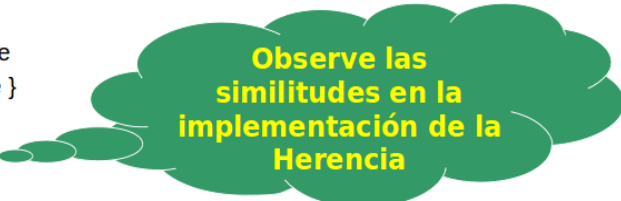
Figura 9.12 Clases, Herencia y Encapsulamiento

**Ejemplo 10.3:**

Implementación de la herencia en diferentes lenguajes de programación.

La idea de este ejemplo es mostrar como el concepto de Herencia se implementa de diferentes formas según la sintaxis de cada lenguaje de programación usado, pero aún así es posible observar que éste tiene las mismas características conceptuales sin importar el lenguaje usado.

- **C++:**  
class C\_Hija : public C\_Padre  
{ implementación de la Clase }
- **JAVA:**  
class C\_Hija extends C\_Padre  
{ implementación de la Clase }
- **Delphi:**  
type  
C\_Hija = class(C\_Padre)  
    implementación de la Clase  
end;
- **ADA:**  
type C\_Hija is new C\_Padre with  
    implementación de la Clase
- **SmallTalk:**  
class C\_Hija superclass C\_Padre  
    implementación de la Clase



**Observe las similitudes en la implementación de la Herencia**

Figura 9.13 Ejemplos de implementación de la Herencia en diferentes LDP

### i. Mensaje:

Es el medio a través del cual se comunican los objetos y es la forma en que estos acceden a los servicios de otro objeto. Lo anterior implica que mediante un mensaje, un objeto le pide a otro que ejecute alguna operación que éste posee, pero quien la pide no (concepto de delegación de servicios). Un objeto sólo puede mandar mensajes a aquellos objetos que lo “conocen”, estos son aquellos con los que están relacionados.

Para entender de mejor manera el concepto de mensaje, creemos importante que te plantees la siguiente pregunta ¿Cómo un objeto ejecuta un mensaje? Una respuesta a esta pregunta podríamos darla usando el siguiente algoritmo.

*Un objeto recibe un mensaje.*

*El objeto busca el mensaje en su propia clase.*

*Si ( el objeto encuentra el mensaje en su clase )*

*lo ejecuta*

*Sino*

*Si ( el objeto “sube por la jerarquía de clases a la que pertenece” y encuentra la 1ª clase que lo implemente )*

*lo ejecuta*

*Sino*

*devuelve un error.*

*Fin Si*

*Fin Si*

Una representación de la interacción entre objetos a través de un mensaje se muestra en la Figura 9.14, donde se

puede observar como ambos objetos se encuentran relacionados. El *objeto de tipo Trabajador* identificado como *Juan* se encuentra asociado al *objeto cA del tipo CajeroAutomatico*. El *mensaje girar dinero* se observa en la flecha desde el *objeto Juan*, con la intención de realizar un giro hacia el *objeto cA*. En este caso el *objeto Juan* desea realizar un giro, él no sabe como, pero si sabe que el *objeto cA* lo puede hacer, luego el *objeto Juan* le hace una petición al *objeto cA* mediante dicho mensaje.

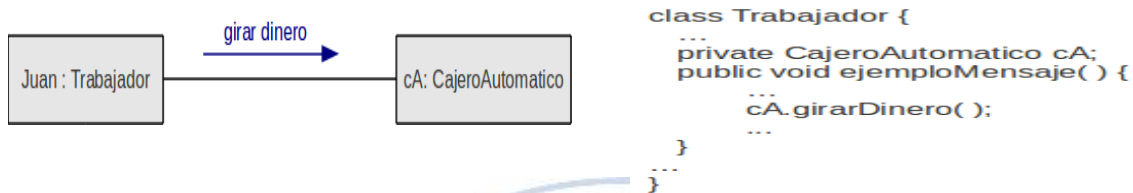


Figura 9.14 Mensaje entre objetos. Figura 9.15 Implementación del mensaje en Java

Para cerrar esta extensa pero, creemos necesaria sección de conceptos básicos, presentamos la Figura 9.16, donde se observan los conceptos fundamentales y como estos se relacionan.

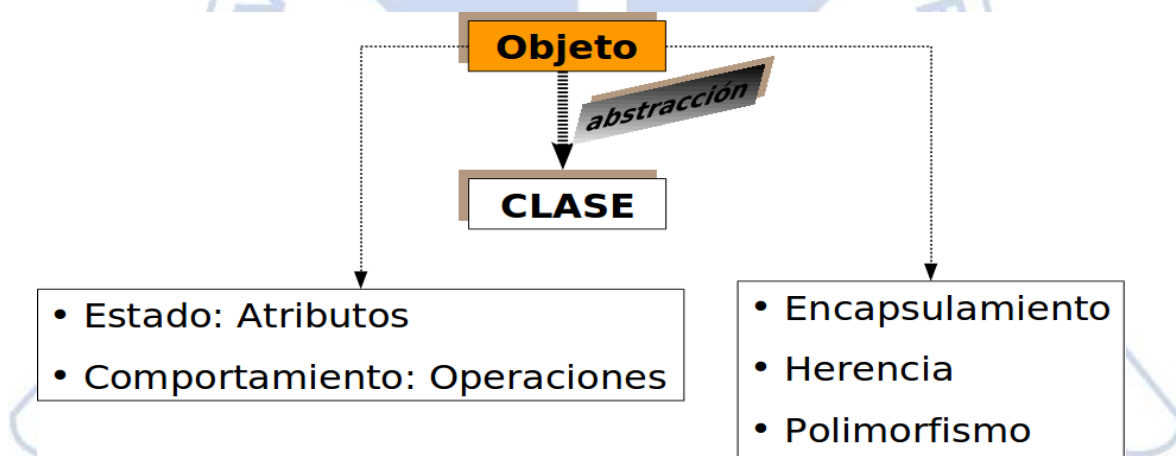


Figura 9.16 Principales conceptos revisados de la POO.

### 9.3 Diferencias entre la Programación Estructurada y la POO

Recordemos que en el Cap. 7 se mencionó lo que entendemos por Programación Estructurada, de forma resumida se entiende que la solución a un problema consiste en determinar y solucionar un conjunto de subproblemas más sencillos que el original, usando lo que se conoce como diseño de programas *top down* o descendente.

Así entonces y haciendo uso del diseño modular, se tiene que la programación estructurada basa sus soluciones en un conjunto de estructuras de datos y funciones que las utilizan. Una representación gráfica de aquello se muestra en la Figura 9.17

Por otro lado, al inicio de este capítulo se planteó la idea de mostrar otra forma de solucionar un problema, la POO entiende la solución a un problema como un conjunto de objetos diferentes entre sí, que se interrelacionan a través de mensajes.

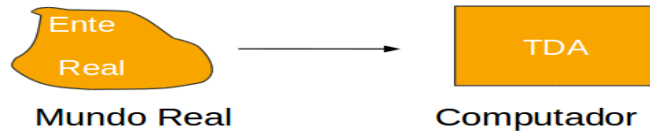
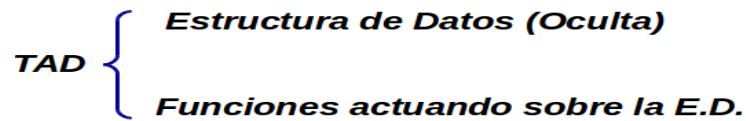


Figura 9.17 Los Tipos Abstractos de Datos y el modelado de soluciones.

Para enfatizar lo mencionado en el párrafo anterior es que se presenta la Figura 9.18, la cual muestra de forma genérica lo sucedido al ejecutar 2 programas que solucionen el mismo problema. El esquema de la izquierda muestra una solución desde una perspectiva de la *Programación Estructurada*, donde un set de funciones accede y manipula las *Estructuras de Datos*. Por su parte el esquema de la derecha muestra la misma solución, pero basado en la POO, se puede observar un conjunto de objetos relacionados que se intercambian mensajes para realizar diferentes tareas.

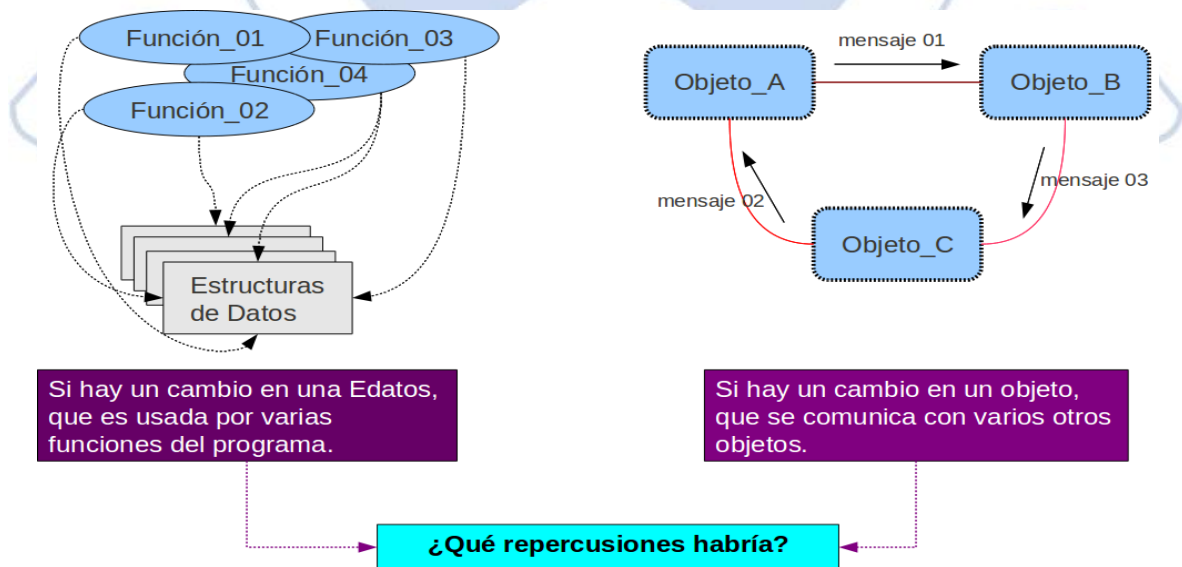


Figura 9.18 Repercusiones en la solución usando Programación Estructurada vs. POO.



### 9.3.1 Caso práctico: Programación Estructurada vs. POO

A continuación se presentará un caso de estudio, el cual se usará como ejemplo para construir y explicar su solución creando dos programas en lenguaje C++, con la salvedad que el primer programa mostrará la solución basándose en las propiedades del paradigma de la programación estructurada, en cambio el segundo programa se creará acorde al paradigma de la POO.

La idea que se persigue es plantear un contexto problema y luego una solución, que se base por un lado en el uso de struct y funciones, en cambio la otra solución se basará en la definición de clases y creación de objetos.

La situación que se presenta consiste en implementar un programa que permita trabajar con fracciones. La solución debe permitir que se puedan realizar las cuatro operaciones aritméticas elementales entre fracciones (sumas, restas, multiplicaciones y divisiones entre fracciones). La salida de impresión por pantalla debe estar de acuerdo con el siguiente formato N/D (N: numerador y D: denominador). Por último, después de cada operación las fracciones deben quedar simplificadas.

#### 9.3.1.1 Solución basada en Programación Estructurada

##### *i. Diseño de la solución*

Identificar la ED necesaria que permita solucionar el problema, en este caso una fracción, un diseño e implementación de ésta, hecha en C++ se muestra en la Fig. 9.19.

```
struct Fraccion {  
    int numerador;  
    int denominador;  
};
```

Figura 9.19 Definición de la ED para manejar fracciones.

A continuación definimos las funciones que interactúan con la ED para poder satisfacer las exigencias del caso, luego se obtiene la implementación en lenguaje C++ de las 7 funciones que se muestran en la Fig. 9.20.

```
Fraccion suma(Fraccion, Fraccion);  
Fraccion resta(Fraccion, Fraccion);  
Fraccion multiplicar(Fraccion, Fraccion);  
Fraccion dividir(Fraccion, Fraccion);  
void imprimirFraccion(Fraccion);  
void leerFraccion(Fraccion &);  
void simplificar(Fraccion &);
```

Figura 9.20 Definición de las funciones que interactúan con la ED para manejar fracciones.

##### *ii. Implementando las funciones*

Una vez definidas las funciones, procedemos con la implementación del código interno para cada una de ellas, cuyo resultado se muestra en la Fig. 9.21.

```

Fraccion suma(Fraccion f1, Fraccion f2) {
    Fraccion aux;

    aux.numerador = f1.numerador * f2.denominador + f1.denominador * f2.numerador;
    aux.denominador = f1.denominador * f2.denominador;
    simplificar(aux);
    return aux;
}

Fraccion resta(Fraccion f1, Fraccion f2) {
    Fraccion aux;

    aux.numerador = f1.numerador * f2.denominador - f1.denominador * f2.numerador;
    aux.denominador = f1.denominador * f2.denominador;
    simplificar(aux);
    return aux;
}

Fraccion multiplicar(Fraccion f1, Fraccion f2) {
    Fraccion aux;

    aux.numerador = f1.numerador * f2.numerador;
    aux.denominador = f1.denominador * f2.denominador;
    simplificar(aux);
    return aux;
}

Fraccion dividir(Fraccion f1, Fraccion f2) {
    Fraccion aux;

    aux.numerador = f1.numerador * f2.denominador;
    aux.denominador = f1.denominador * f2.numerador;
    simplificar(aux);
    return aux;
}

void imprimirFraccion(Fraccion f) {
    cout << f.numerador << '/' << f.denominador;
}

void leerFraccion(Fraccion &f) {
    cout << "\nIngrese numerador: ";
    cin >> f.numerador;
    do {
        cout << "Ingrese denominador: "; //Valida que denominador sea distinto de cero
        cin >> f.denominador;
    } while (f.denominador == 0);
}

void simplificar(Fraccion &f) {
    int mcd = 0; // maximo comun divisor
    int mayor;

    mayor = f.numerador > f.denominador ? f.numerador : f.denominador;

    for (int contador = 2; contador <= mayor; contador++)
        if (f.numerador % contador == 0 && f.denominador % contador == 0)
            mcd = contador;

    if (mcd != 0) {
        f.numerador /= mcd;
        f.denominador /= mcd;
    }
}

```

Figura 9.21 Implementación del código interno para las funciones que interactúan con la ED.

### iii. Crear un programa que use nuestro diseño

Habiendo definido la ED necesaria, así como las funciones que la usen, ahora nos resta crear un programa que nos permita usar la ED y las funciones para operar con fracciones. La sección principal de dicho programa se muestra en la Fig. 9.22.

```
int main () {  
  
    Fraccion c, d, x;  
  
    leerFraccion(c);  
    leerFraccion(d);  
  
    cout << '\n';  
    imprimirFraccion(c);  
    cout << " + ";  
    imprimirFraccion(d);  
    x = suma(c, d);  
    cout << " = ";  
    imprimirFraccion(x);  
    cout << '\n';  
  
    cout << '\n';  
    imprimirFraccion(c);  
    cout << " - ";  
    imprimirFraccion(d);  
    x = resta(c, d);  
    cout << " = ";  
    imprimirFraccion(x);  
    cout << '\n';  
  
    cout << '\n';  
    imprimirFraccion(c);  
    cout << " x ";  
    imprimirFraccion(d);  
    x = multiplicar(c, d);  
    cout << " = ";  
    imprimirFraccion(x);  
    cout << '\n';  
  
    cout << '\n';  
    imprimirFraccion(c);  
    cout << " / ";  
    imprimirFraccion(d);  
    x = dividir(c, d);  
    cout << " = ";  
    imprimirFraccion(x);  
    cout << '\n';  
  
    system("pause>null");  
    return 0;  
}
```



Figura 9.22 Sección principal del programa en C++ que usa variables del tipo de la ED definidas.

### 9.3.1.2 Solución basada en POO

#### *i. Diseño de la solución*

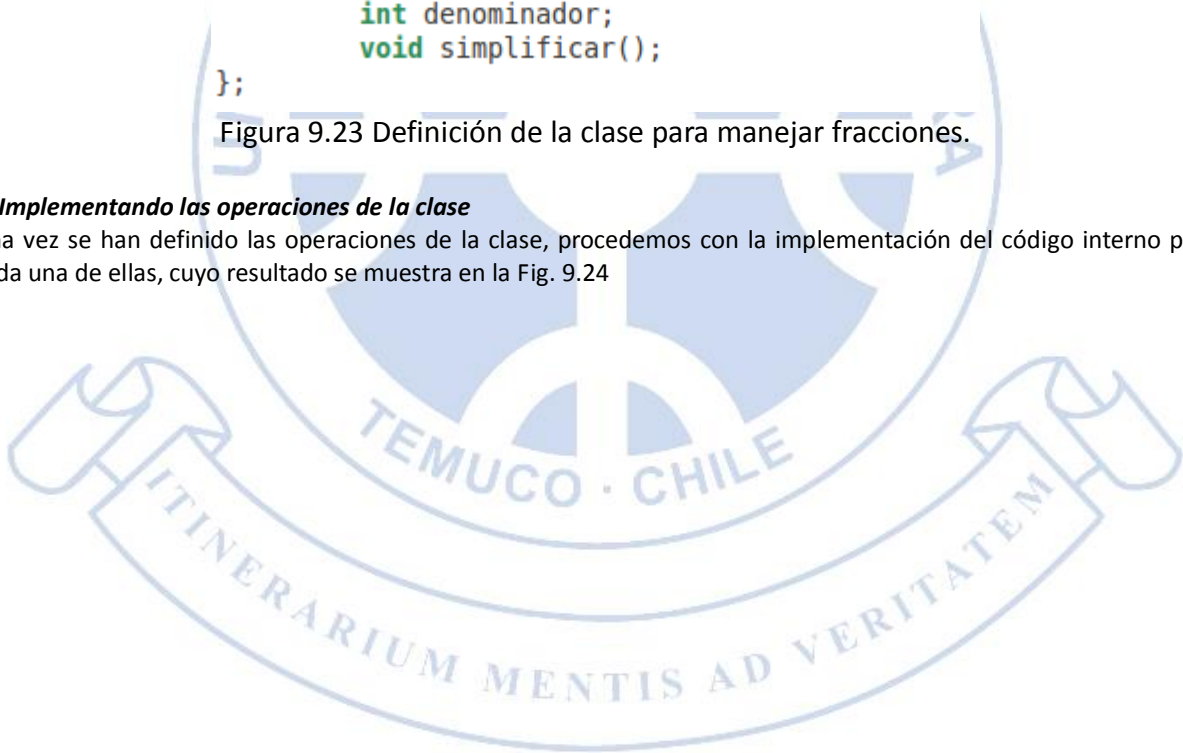
Identificar la clase necesaria que permita solucionar el problema, en este caso una fracción, un diseño e implementación de ésta, hecha en C++ se muestra en la Fig. XXX. Esto implica determinar tanto los atributos, como las operaciones de la misma.

```
class Fraccion {  
public:  
    Fraccion();  
    void suma(Fraccion, Fraccion);  
    void resta(Fraccion, Fraccion);  
    void multiplicar(Fraccion, Fraccion);  
    void dividir(Fraccion, Fraccion);  
    void imprimirFraccion();  
    void leerFraccion();  
private:  
    int numerador;  
    int denominador;  
    void simplificar();  
};
```

Figura 9.23 Definición de la clase para manejar fracciones.

#### *ii. Implementando las operaciones de la clase*

Una vez se han definido las operaciones de la clase, procedemos con la implementación del código interno para cada una de ellas, cuyo resultado se muestra en la Fig. 9.24





```

Fraccion::Fraccion() {
    this->numerador = 0;
    this->denominador = 1;
}

void Fraccion::suma(Fraccion f1, Fraccion f2) {
    this->numerador = f1.numerador * f2.denominador + f1.denominador * f2.numerador;
    this->denominador = f1.denominador * f2.denominador;
    this->simplificar();
}

void Fraccion::resta(Fraccion f1, Fraccion f2) {
    this->numerador = f1.numerador * f2.denominador - f1.denominador * f2.numerador;
    this->denominador = f1.denominador * f2.denominador;
    this->simplificar();
}

void Fraccion::multiplicar(Fraccion f1, Fraccion f2) {
    this->numerador = f1.numerador * f2.numerador;
    this->denominador = f1.denominador * f2.denominador;
    this->simplificar();
}

void Fraccion::dividir(Fraccion f1, Fraccion f2) {
    this->numerador = f1.numerador * f2.denominador;
    this->denominador = f1.denominador * f2.numerador;
    this->simplificar();
}

void Fraccion::imprimirFraccion() {
    cout << this->numerador << '/' << this->denominador;
}

void Fraccion::leerFraccion() {
    cout << "\nIngrese numerador: ";
    cin >> this->numerador;
    do {
        cout << "Ingrese denominador: "; // Valida que denominador sea distinto de cero
        cin >> this->denominador;
    } while(denominador == 0);
}

void Fraccion::simplificar() {
    int mcd = 0; // maximo comun divisor
    int mayor;
    mayor = this->numerador > this->denominador ? this->numerador : this->denominador;
    for ( int contador = 2; contador <= mayor; contador++ )
        if ( this->numerador % contador == 0 && this->denominador % contador == 0 )
            mcd = contador;
    if (mcd != 0) {
        this->numerador /= mcd;
        this->denominador /= mcd;
    }
}

```

Figura 9.24 Implementación de las operaciones de la clase para manipular fracciones.

### iii. Crear un programa que use nuestro diseño

Habiendo definido la clase necesaria, tanto sus atributos como las operaciones, ahora nos resta crear un programa que nos permita usar la clase y sus operaciones para operar con fracciones. La sección principal de dicho programa se muestra en la Fig. 9.25.

```
int main() {
    Fraccion c, d, x;

    c.leerFraccion();
    d.leerFraccion();

    cout << '\n';
    c.imprimirFraccion();
    cout << " + ";
    d.imprimirFraccion();
    x.suma(c, d);
    cout << " = ";
    x.imprimirFraccion();
    cout << '\n';

    cout << '\n';
    c.imprimirFraccion();
    cout << " - ";
    d.imprimirFraccion();
    x.resta(c, d);
    cout << " = ";
    x.imprimirFraccion();
    cout << '\n';

    cout << '\n';
    c.imprimirFraccion();
    cout << " x ";
    d.imprimirFraccion();
    x.multiplicar(c, d);
    cout << " = ";
    x.imprimirFraccion();
    cout << '\n';

    cout << '\n';
    c.imprimirFraccion();
    cout << " / ";
    d.imprimirFraccion();
    x.dividir(c, d);
    cout << " = ";
    x.imprimirFraccion();
    cout << '\n';

    system("pause>null");
    return 0;
}
```

**Luego algunos comentarios:**

Debe quedar claro que:

- la primera solución está basada en una Estructura de Datos, que se construyó usando variables del *tipo de dato struct* y las funciones que la usan son independientes de ésta.
- La segunda solución está basada en clases, que se construyeron usando el *tipo de dato class* y que éstas encapsulan sus atributos y los métodos que los manipulan.

**Preguntas para el alumno:**

¿Cuáles son las principales diferencias que aprecia entre ambas soluciones?

¿Cuál de las dos soluciones le parece “mejor”?

¿Por qué?

Figura 9.25 Sección principal del programa en C++ que objetos de la clase definida.

## 9.4 Diseño de Software con POO

A continuación se presenta un caso, cuya solución se modelará usando la notación gráfica definida por el lenguaje de modelado UML<sup>1</sup>, para representar las clases y sus relaciones que permitan construir luego los programas computacionales que implementen la solución.

Se hará uso de la herramienta BlueJ<sup>2</sup> que permite desplegar elementos de modelado UML.

El caso a revisar es una situación bastante común: un alumno está usando su computador para escribir un documento con la tarea para mañana, que es del tipo archivo de texto y luego desea imprimir el contenido de éste en su impresora láser XJ120.

- Si revisamos con cuidado el caso podemos ver que la solución debe construirse a partir de los objetos tarea del alumno y la impresora láser XJ120.
- Si seguimos mirando con detalle, veremos que dichos objetos son casos particulares de las clases archivo de texto e impresora láser, con lo que tenemos las clases fundamentales para modelar nuestra solución.
- Lo que faltaría ahora es determinar las operaciones necesarias de cada clase para permitir la implementación de nuestra solución. Para el caso de los archivos, se requeriría un método que permitiera que el alumno escriba contenido en el archivo. En el caso de la impresora se requiere un método que sea capaz de imprimir el contenido que le sea entregado por algún objeto que lo requiera, en nuestro caso sería el archivo con la tarea del alumno que éste quiere imprimir.

Resumiendo lo anterior, se tiene que hemos identificado las clases: Archivo de Texto e impresora Láser y al menos un método para cada clase, escribir contenido e imprimir contenido respectivamente. Si ponemos estas características en lo que UML llama diagrama de clases, nuestra solución podría ser modelada como se muestra en la siguiente figura.

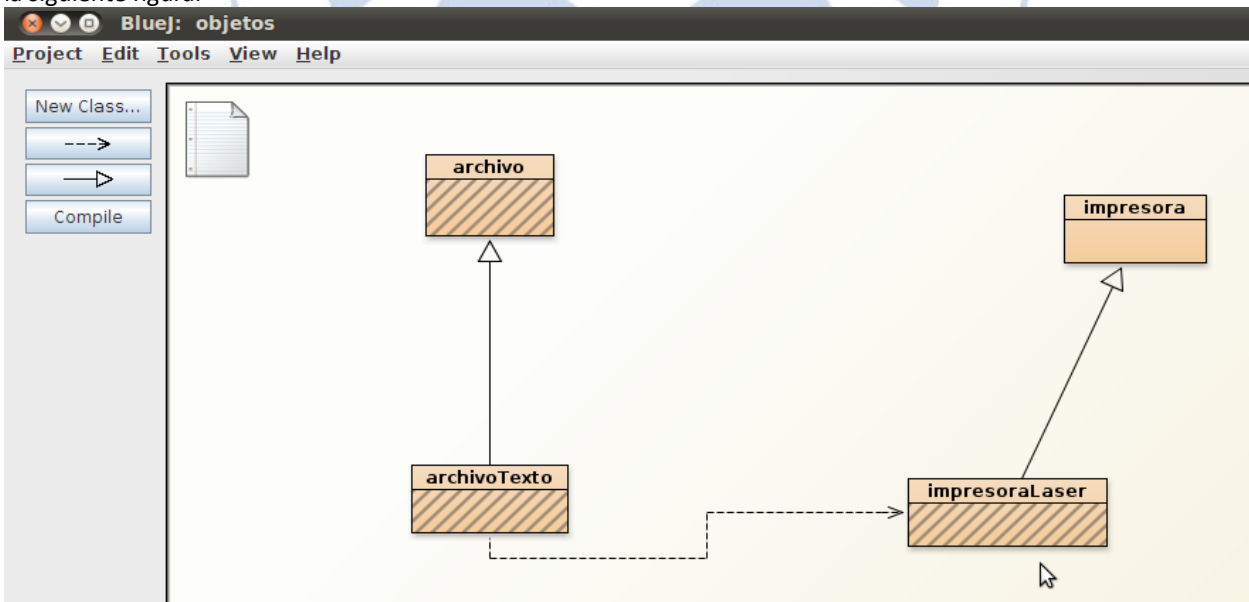


Figura 9.26 Diagrama de clases.

<sup>1</sup> UML, sigla de los términos Unified Modeling language o Lenguaje Unificado de Modelado, definido por el Object Management Group con el fin de modelar un sistema de software. Más detalles en <http://www.uml.org/>

<sup>2</sup> Más detalles respecto de esta herramienta en <http://www.bluej.org/>

Se puede observar del diagrama anterior, que se representa con una flecha de punta triangular la relación particular de clase Padre-clase Hija, en nuestro ejemplo entre la clase *archivo* y la clase *archivo Texto*, que es una especialización de un tipo de *archivo*. Lo mismo sucede con las clases para representar las impresoras.

Otro aspecto interesante es ver la relación que se establece entre las clases *archivoTexto* e *impresoraLaser*, pues los archivos “no saben como imprimir su contenido” y por lo tanto le solicita un servicio a la impresoraLaser para que imprima su contenido.

Otra forma de visualizar lo anteriormente expuesto, es mostrar la relación entre objetos de las clases y ver como estos intercambian mensajes para poder realizar las acciones que requiere el usuario (el alumno). Esto es lo que en UML se llama un diagrama de secuencias.

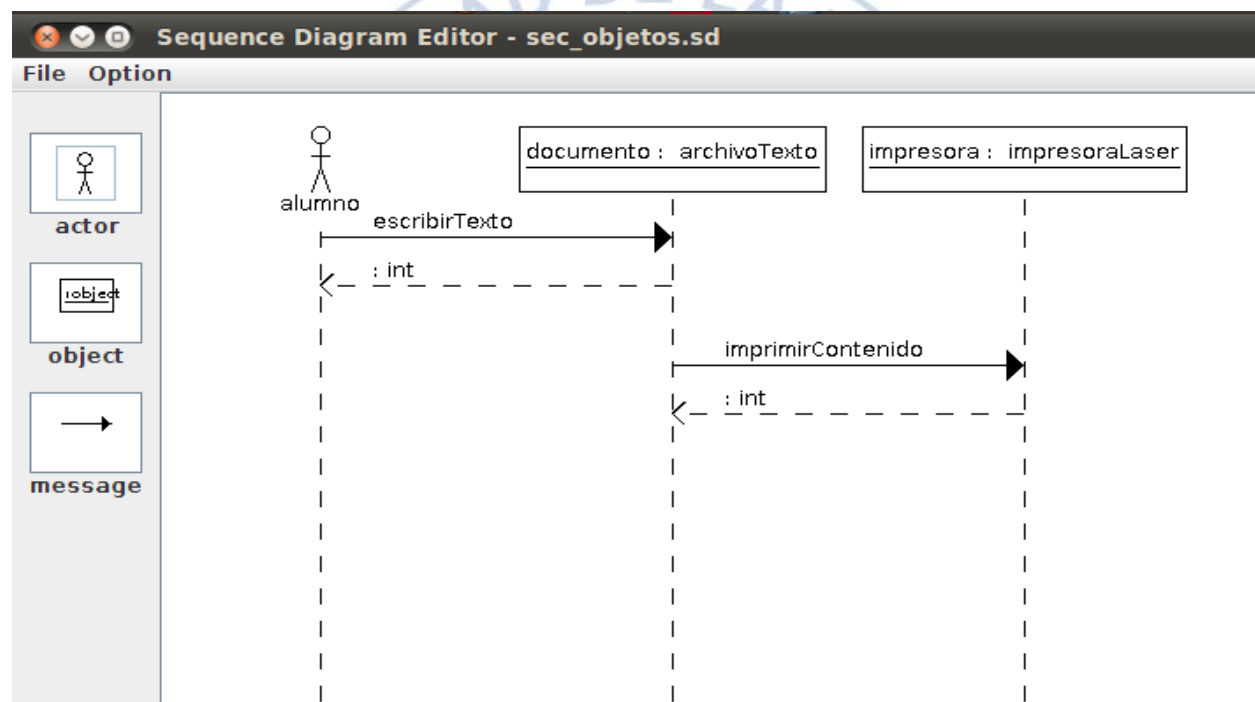


Figura 9.27 Diagrama de secuencias

La figura anterior nos permite ver como los objetos *documento* e *impresora* intercambian mensajes, pidiéndoles que realicen algunos servicios, a través de los métodos que implementa cada clase a la que pertenecen los objetos. En nuestro caso es el alumno quien le pide al objeto documento que escriba el texto llamando al método *escribirTexto* y luego cuando se desea imprimir el archivo, se pide al objeto impresora que lo imprima.

Se debe destacar que el resumen presentado hasta ahora respecto del UML y su uso para modelar soluciones de programas con POO, será ampliado para las carreras con formación en el área informática en cursos superiores de la especialidad como Ingeniería de Software.



## 9.5 Conceptos básicos sobre el lenguaje Java

A continuación se hará una descripción resumida de los orígenes y las principales características técnicas del lenguaje Java, con el objetivo de presentar los orígenes históricos y entender los principales aspectos técnicos que rodean al lenguaje, en particular lo relacionado con el concepto de la Máquina Virtual de Java.

### 9.5.1 Orígenes del lenguaje

Java (Fig. 9.28a) es un lenguaje de POO desarrollado por Sun microsystems (Fig. 10.13a), actualmente propiedad de la empresa Oracle (Fig. 9.28b). Sin embargo, su historia se remonta a la creación de una filial de Sun microsystems llamada FirstPerson, la cual estaba enfocada al desarrollo de aplicaciones para electrodomésticos que desapareció tras un par de éxitos de laboratorio y ningún desarrollo comercial. El nombre del lenguaje se dice que viene del café que tomaban sus creadores, proveniente de la isla de Java (Fig. 9.28c), por otro lado la Fig. 9.28e muestra la mascota del lenguaje Java, llamada Duke.

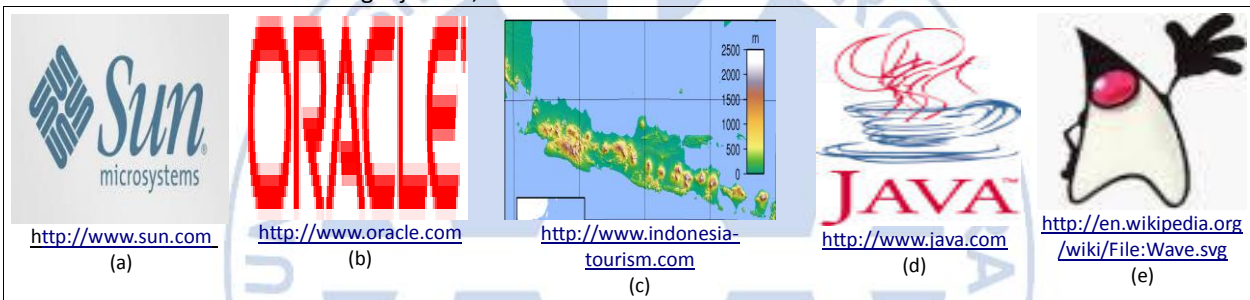


Figura 9.28 Java

Para el desarrollo en el laboratorio, uno de los trabajadores de FirstPerson, James Gosling (<http://nighthacks.com/roller/jag/resource/bio.html>), desarrolló un lenguaje derivado de C++ que intentaba eliminar las deficiencias del mismo y las heredadas de C, llamándolo Oak ([dejar link](#)).

Cuando Sun abandonó el proyecto de FirstPerson, se encontró con este lenguaje y, tras varias modificaciones (entre ellas la del nombre), decidió lanzarlo al mercado en verano de 1995, como Java y como dicen por ahí el resto es historia...

### 9.5.2 Algunas características técnicas de Java

Muchas de las bondades o al éxito que ha alcanzado el lenguaje Java, se atribuyen a un conjunto de características de éste, entre las cuales podemos mencionar:

1. En general es un lenguaje sencillo de escribir y programar, eliminando una de las mayores complejidades de lenguajes como C++ y de C, que tiene relación el manejo y acceso directo a la memoria vía el uso de "punteros".
2. Es un lenguaje independiente de plataforma sobre la cual se desarrollan y ejecutan los programas, pues se basa en el concepto de máquina virtual, por lo que potencialmente un programa hecho en Java se ejecutará igual en un PC con SO MS-Windows que en una estación de trabajo basada en Unix.
3. Destaca también por su seguridad (ver el punto 1.), pues desarrollar programas que accedan ilegalmente a la memoria es una tarea bastante compleja.
4. Posee también capacidad para desarrollar programas usando el *paradigma multihilo* y también integra lo relativo al protocolo TCP/IP, haciendo de Java un lenguaje ideal para desarrollar aplicaciones que funcionen sobre Internet (J2EE, Web Services, SOA, etc.).

5. Java respeta en un 100% el paradigma de POO, pues con excepción de los datos que podríamos llamar de tipo simple (números enteros y flotantes, caracteres y variables booleanas), todo lo demás debe ser definido y manejado como un objeto.

### 10.5.3 El concepto de Máquina Virtual (MV)

En esta sección debiéramos ser capaces de responderte la pregunta ¿Qué es una Máquina Virtual? Podríamos iniciar esta respuesta diciendo que una MV es un componente de software que permite a ciertas aplicaciones ejecutarse tal como si lo hicieran sobre su plataforma de software nativa. Esto implica que las aplicaciones que se ejecutan sobre la MV, no se dan cuenta que no se están ejecutando sobre otras plataformas de software que no son las propias.

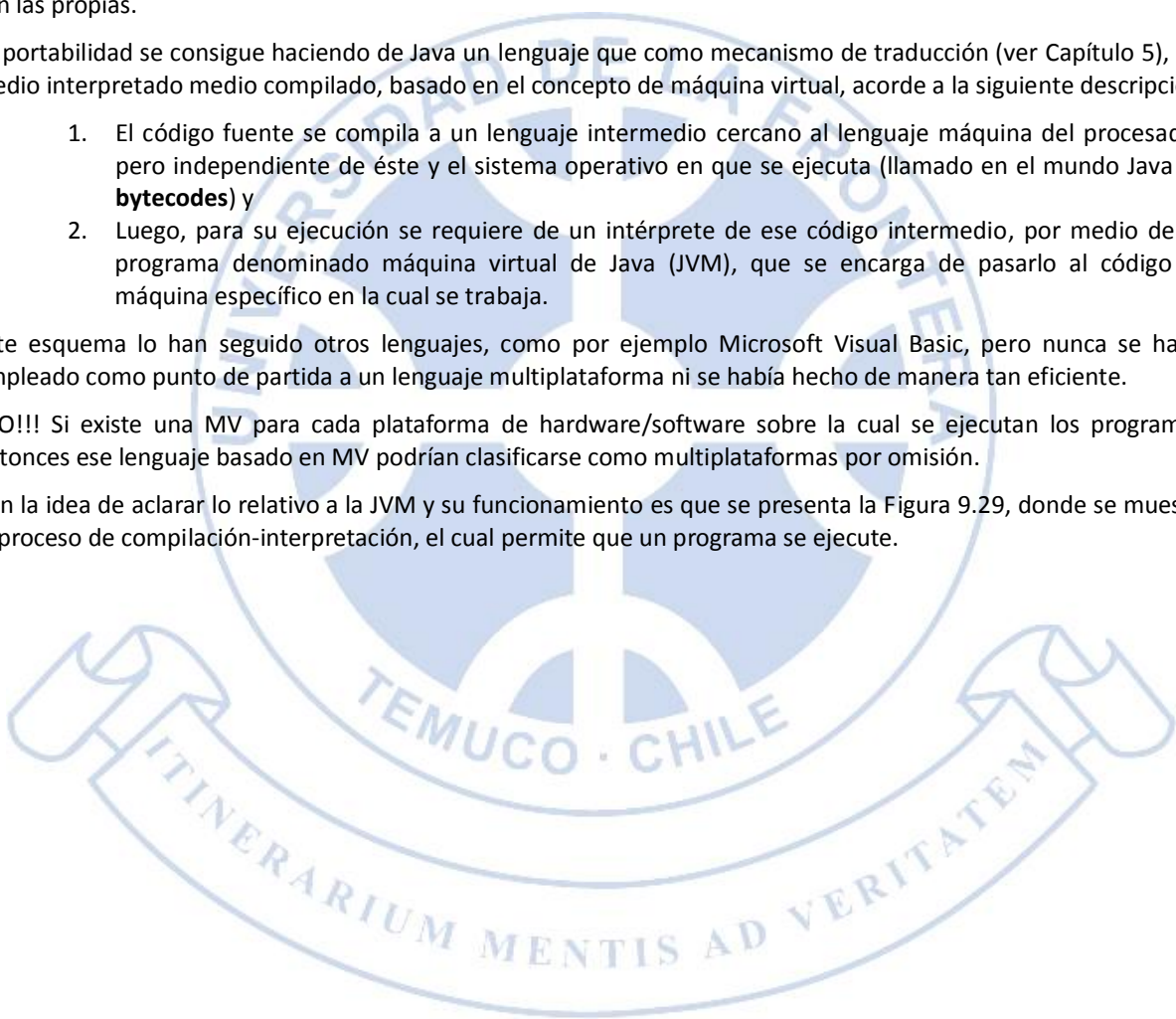
La portabilidad se consigue haciendo de Java un lenguaje que como mecanismo de traducción (ver Capítulo 5), sea medio interpretado medio compilado, basado en el concepto de máquina virtual, acorde a la siguiente descripción:

1. El código fuente se compila a un lenguaje intermedio cercano al lenguaje máquina del procesador, pero independiente de éste y el sistema operativo en que se ejecuta (llamado en el mundo Java los **bytecodes**) y
2. Luego, para su ejecución se requiere de un intérprete de ese código intermedio, por medio de un programa denominado máquina virtual de Java (JVM), que se encarga de pasarlo al código de máquina específico en la cual se trabaja.

Este esquema lo han seguido otros lenguajes, como por ejemplo Microsoft Visual Basic, pero nunca se había empleado como punto de partida a un lenguaje multiplataforma ni se había hecho de manera tan eficiente.

OJO!!! Si existe una MV para cada plataforma de hardware/software sobre la cual se ejecutan los programas, entonces ese lenguaje basado en MV podrían clasificarse como multiplataformas por omisión.

Con la idea de aclarar lo relativo a la JVM y su funcionamiento es que se presenta la Figura 9.29, donde se muestra el proceso de compilación-interpretación, el cual permite que un programa se ejecute.



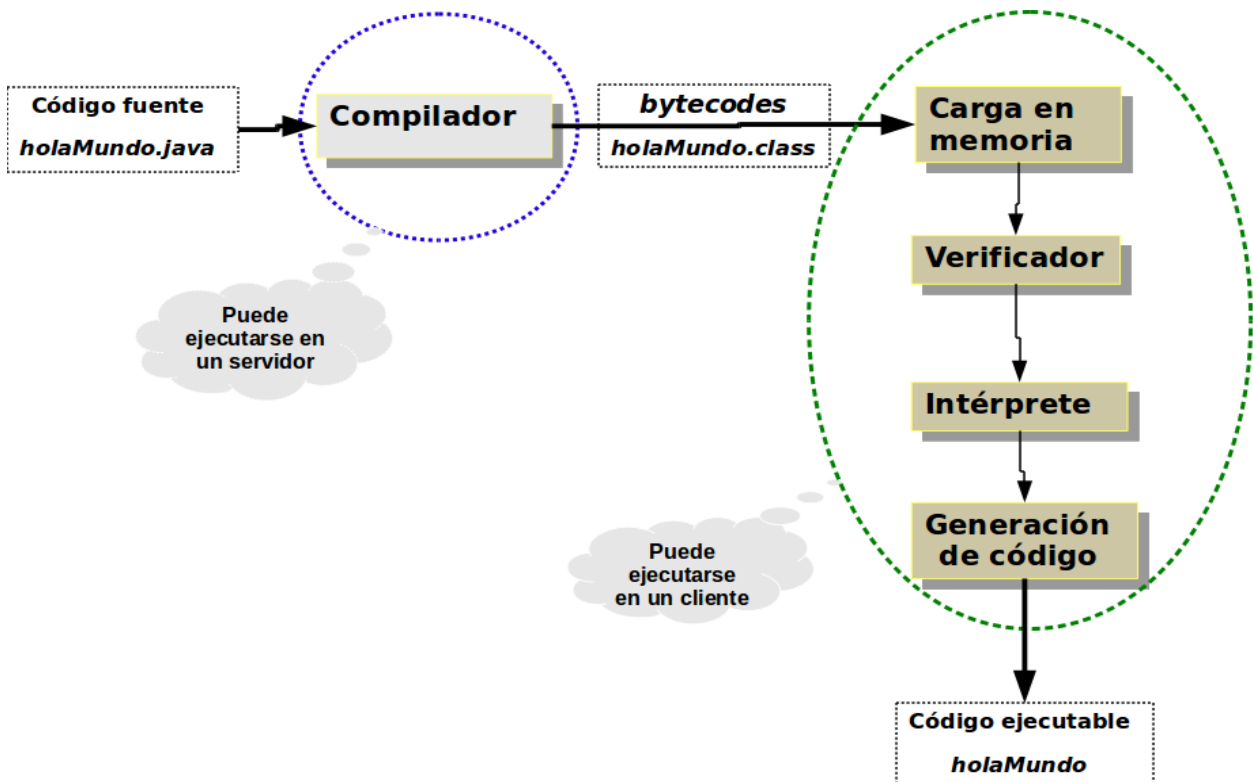


Figura 9.29 Esquema Simplificado Máquina Virtual de Java – JVM.

Un elemento importante a destacar de la Figura 9.29 dice relación con las secciones encerradas en las áreas de color azul y verde, pues si se leen con atención las indicaciones, se puede deducir entonces que las aplicaciones desarrolladas con Java están muy en sintonía con el desarrollo para la Web, pues es posible dejar una parte de la aplicación en un servidor (la sección azul de la Figura 9.29) y distribuir la otra parte de la aplicación entre los clientes que deseen ejecutarla, estos pueden ser PC, notebooks, teléfonos móviles, tablets, etc. (la sección verde de la Figura 9.29).

El detalle del trabajo realizado por la JVM, visto desde la ejecución de un programa particular es lo que se muestra en la Figura 9.30.

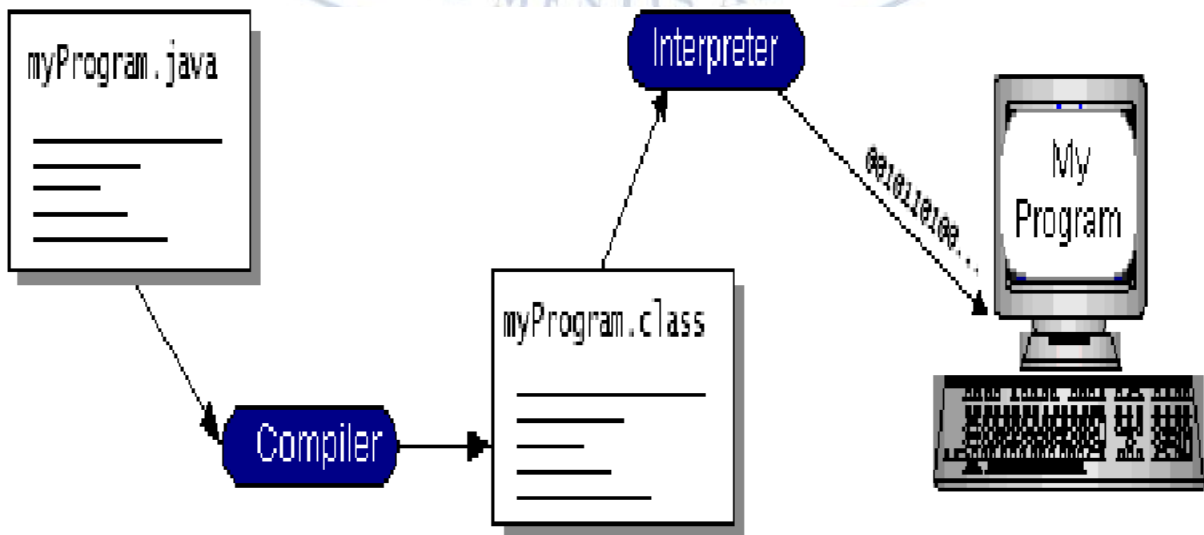


Figura 9.30 Ejecución de un programa usando la JVM.

El uso de la JVM como ya se ha dicho otorga la posibilidad que nuestros programas desarrollados con Java sean multiplataforma, esto significa desde una perspectiva más técnica del lenguaje cumplir con el lema que ha caracterizado al lenguaje *“Write once, run anywhere...”*, lo cual queda reflejado en la Figura 9.31. Esto implica que el programa una vez que ha sido creado, sólo requeriría ser compilado una vez y luego podría ser ejecutado sobre diferentes plataformas de software, si ésta tiene la versión adecuada del intérprete alojado en la JVM. En el ejemplo se tiene el programa `holaMundo.java`, el cual se compila, con lo cual se genera un archivo `holaMundo.class` y que luego para ser ejecutado, debe ser interpretado por la JVM y obtener así el código ejecutable para cada una de las plataformas, en este caso Ms Windows 32 bits, GNU Linux Ubuntu y Mac OS XXXXX

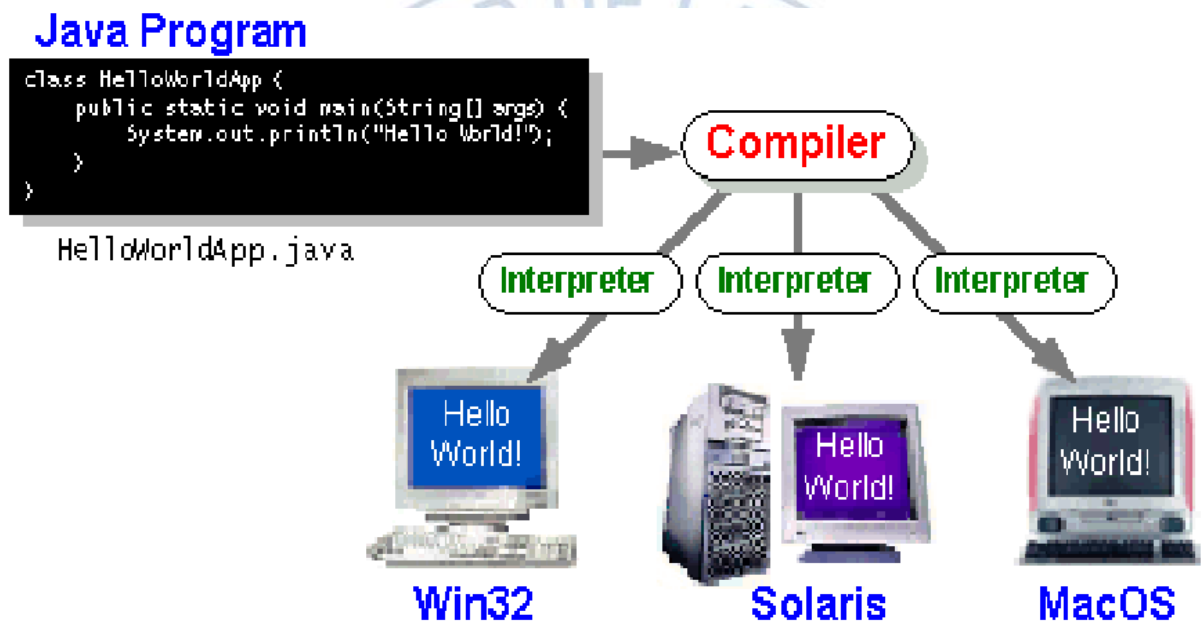


Figura 9.31 Ejecución de un mismo programa en diferentes plataformas, usando cada una su propia versión de la JVM.

## 9.6 Programando en Java usando POO

- ejemplos que muestren uso de encapsulamiento y herencia
- Ejemplo de las fracciones, pero ahora en Java
- uno completo con análisis UML y BlueJ

## 10.7 Ejercicios resueltos

1. ¿Cuándo podemos hablar del Encapsulamiento usado de forma correcta?  
La encapsulación será correcta cuando la interfaz que se muestra al público cumpla las tareas que desea el cliente y no muestre más de lo que éste precisa.
2. ¿Qué debe tener un Lenguaje de Programación para ser considerado Orientado a Objetos (OO)?



Para ser considerados lenguajes OO, deben soportar básicamente:

- Definición de Clases y Creación de objetos
- Ocultamiento de Datos (Encapsulamiento)
- Herencia

Sí es importante resaltar que cualquier programa realizado con un lenguaje OO podría realizarse con un lenguaje estructurado o no OO. Es más, se podrían escribir programas 100% OO usando un lenguaje de desarrollo no OO, pero ¡resultaría bastante COMPLEJO!

3. Usando el lenguaje Java, crearemos una clase que permita manipular los datos de una persona. Estos datos son: *nombre, apellido, rut, edad y su peso corporal*. Se requiere un set de métodos que permitan recuperar el nombre completo de la persona, su edad y saber si la persona es mayor de 25 años, informando esto último por pantalla.

Análisis:

Los atributos de la clase Persona debieran ser encapsulados como **private** y de tipo:

- String para nombre, apellido y rut
- float para el peso corporal
- int para la edad.

Los métodos requeridos debieran ser de tipo public:

- El constructor de la clase *Persona(String nom, String ape, String ru, int ed, float pe)*
- Para retornar el nombre completo *String obtenerNombreCompleto( )*
- Para retornar la edad de la persona *static float obtenerEdad( )*
- Para saber si la persona es mayor de 25 años *static void evaluarEdad( )*

A partir de lo anterior, una primera versión de la clase persona sería lo que se muestra en la Fig. XXXX

```
public class Persona {  
    private String nombre;  
    private String apellido;  
    private String run;  
    static private int edad;  
    private float peso;  
  
    public Persona(String nom, String ape, String ru, int ed, float pe) {  
    }  
  
    public String obtenerNombreCompleto() {  
    }  
  
    public static float obtenerEdad() {  
    }  
  
    public static void evaluarEdad() {  
    }  
}
```

Una vez se han completado los métodos asociados, la clase Persona tendría la siguiente forma cómo se muestra en la Fig. XXX:



```
public class Persona {  
  
    // ATRIBUTOS  
    private String nombre;  
    private String apellido;  
    private String run;  
    static private int edad;  
    private float peso;  
  
    // METODOS  
    public Persona(String nom, String ape, String ru, int ed, float pe) {  
        nombre = nom;  
        apellido = ape;  
        run = ru;  
        edad = ed;  
        peso = pe;  
    }  
  
    public String obtenerNombreCompleto() {  
        return nombre+" "+apellido;  
    }  
  
    public static float obtenerEdad() {  
        return edad;  
    }  
  
    public static void evaluarEdad() {  
        if (Persona.obtenerEdad()>25)  
            System.out.println("La persona tiene más de 25 años");  
        else  
            System.out.println("La persona tiene 25 años o menos");  
    }  
}
```

Lo que faltaría ahora es probar nuestra clase *Persona*, creando un objeto de tipo *Persona* y usando los métodos de la clase. Para esto creamos la clase para nuestro ejemplo llamada *EjemploPersona1*, la cual dentro de su método *main()*, crearemos un objeto de tipo *Persona* y usaremos el método para mostrar el nombre completo de la persona, cuyos datos carguemos en la clase usando el método constructor.

Creamos el *objeto per*, que es de tipo *Persona* y usamos el constructor de la clase para crearlo, con la siguiente instrucción:

```
Persona per = new Persona("Bruno", "Neumann", "11111111-1", 22, 75);
```

Lo que resta luego es mostrar el nombre completo que se ha almacenado en el *objeto per*, con la siguiente instrucción:


```
System.out.println( per.obtenerNombreCompleto( ) );
```

Todo lo anterior queda resumido en el código Java que se muestra en la Fig.XXXX

```
public class EjemploPersona1 {

    public static void main(String args[]) {
        Persona per = new Persona("Bruno", "Neumann", "1111111-1", 22, 75);
        System.out.println(per.obtenerNombreCompleto());
    }
}
```

El código del ejemplo anterior permitiría ver por pantalla el nombre que se ha almacenado en el objeto per. Para verificar este hecho, a continuación se muestra una captura de pantalla de la ejecución de este programa en una consola de Ubuntu Linux.



```
ssc@pc-ssc:~/Dropbox/Libro_FDE/Capitulo10/P00$ ls -l
total 144
-rw-r--r-- 1 ssc ssc 724 2011-05-07 13:54 Alumno.class
-rw-r--r-- 1 ssc ssc 452 2011-05-07 14:11 Alumno.java
-rw-r--r-- 1 ssc ssc 1072 2011-05-07 13:54 EjemploAlumno1.class
-rw-r--r-- 1 ssc ssc 393 2011-05-07 13:54 EjemploAlumno1.java
-rw-r--r-- 1 ssc ssc 596 2011-05-07 14:16 EjemploAlumno2.class
-rw-r--r-- 1 ssc ssc 273 2011-05-07 14:13 EjemploAlumno2.java
-rw-r--r-- 1 ssc ssc 629 2011-05-17 18:00 EjemploPersona1.class
-rw-r--r-- 1 ssc ssc 304 2011-05-07 13:31 EjemploPersona1.java
-rw-r--r-- 1 ssc ssc 287 2011-05-07 13:35 EjemploPersona2.java
-rw-r--r-- 1 ssc ssc 393 2011-05-17 17:40 Persona01.java
-rw-r--r-- 1 ssc ssc 1255 2011-05-07 14:16 Persona.class
-rw-r--r-- 1 ssc ssc 789 2011-05-07 14:16 Persona.java
ssc@pc-ssc:~/Dropbox/Libro_FDE/Capitulo10/P00$ javac EjemploPersona1.java
ssc@pc-ssc:~/Dropbox/Libro_FDE/Capitulo10/P00$ java EjemploPersona1
Bruno Neumann
ssc@pc-ssc:~/Dropbox/Libro_FDE/Capitulo10/P00$
```

4. Pensando en aplicar uno de los conceptos básicos de la POO como es la **herencia**, es que podríamos plantearnos una ampliación de nuestra clase Persona. Supongamos que ahora nos interese gestionar también algunos datos básicos de los alumnos de una universidad, entre estos datos estarían la **carrera** a la que pertenece el alumno y el **número de matrícula** del alumno. Cómo nuevo método podríamos necesitar el recuperar el número de matrícula de alumno. Si nos fijamos con detalle, es posible observar que un alumno es un tipo particular de Persona, pues comparte todos los elementos de una Persona y agrega algunos elementos específicos. En código Java dicha relación de herencia quedaría de la siguiente forma:

```
class Alumno extends Persona {
```

```
    private String carrera;
    private String numeroMatricula;
```

```
    public Alumno(String nom,String ape, String ru, int ed, float pe, String carr, String
    numMat) {
    }
}
```

```

        public String obtenerNumeroMatricula() {
        }
    }

```

Usando esta idea, es que el código completo de la clase alumno quedaría como se muestra en la Fig. XXXX

```

public class Alumno extends Persona {

    private String carrera;
    private String numeroMatricula;

    public Alumno(String nom,String ape, String ru, int ed, float pe, String carr, String numMat) {
        super(nom,ape,ru,ed,pe);
        carrera = carr;
        numeroMatricula = numMat;
    }

    public String obtenerNumeroMatricula() {
        return numeroMatricula;
    }
}

```

Lo que faltaría ahora es probar nuestra clase Alumno, creando un objeto de tipo Alumno y usando los métodos de la clase. Para esto creamos la clase para nuestro ejemplo llamada EjemploAlumno1, la cual dentro de su método *main()*, crearemos un objeto de tipo Alumno y usaremos el método para mostrar el nombre completo y número de matrícula del alumno, cuyos datos carguemos en la clase usando el método constructor.

Creamos el *objeto alu*, que es de tipo Alumno y usamos el constructor de la clase para crearlo, con la siguiente instrucción:

```
Alumno alu = new Alumno("Bruno","Neumann","11111111-1",22,75,"ICII","1111111108");
```

Lo que resta luego es mostrar el nombre completo y el número de la matrícula del alumno que se ha almacenado en el *objeto alu*, con la siguiente instrucción:

```
System.out.println("El alumno " + alu.obtenerNombreCompleto() + " tiene el número de matrícula: " + alu.obtenerNumeroMatricula());
```

Todo lo anterior queda resumido en el código Java que se muestra en la Fig.XXXX

```

public class EjemploAlumno1 {

    public static void main(String args[]) {
        Alumno alu = new Alumno("Bruno","Neumann","11111111-1",22,75,"ICII","1111111108");
        System.out.println("El alumno " + alu.obtenerNombreCompleto() + " tiene el número de matrícula: " + alu.obtenerNumeroMatricula());
    }
}

```

El código del ejemplo anterior permitiría ver por pantalla el nombre y número de matrícula que se ha almacenado en el *objeto alu*. Para verificar este hecho, a continuación se muestra una



captura de pantalla de la ejecución de este programa en una consola de Ubuntu Linux.

```

ssc@pc-ssc:~/Dropbox/Libro_FDE/Capitulo10/P00$ ls -l
total 144
-rw-r--r-- 1 ssc ssc 538 2011-05-17 19:26 Alumno.class
-rw-r--r-- 1 ssc ssc 452 2011-05-07 14:11 Alumno.java
-rw-r--r-- 1 ssc ssc 932 2011-05-17 19:27 EjemploAlumno1.class
-rw-r--r-- 1 ssc ssc 402 2011-05-17 19:27 EjemploAlumno1.java
-rw-r--r-- 1 ssc ssc 596 2011-05-07 14:16 EjemploAlumno2.class
-rw-r--r-- 1 ssc ssc 273 2011-05-07 14:13 EjemploAlumno2.java
-rw-r--r-- 1 ssc ssc 629 2011-05-17 18:01 EjemploPersonal.class
-rw-r--r-- 1 ssc ssc 304 2011-05-07 13:31 EjemploPersonal.java
-rw-r--r-- 1 ssc ssc 287 2011-05-07 13:35 EjemploPersona2.java
-rw-r--r-- 1 ssc ssc 393 2011-05-17 17:40 Persona01.java
-rw-r--r-- 1 ssc ssc 1255 2011-05-07 14:16 Persona.class
-rw-r--r-- 1 ssc ssc 789 2011-05-07 14:16 Persona.java
ssc@pc-ssc:~/Dropbox/Libro_FDE/Capitulo10/P00$ javac EjemploAlumno1.java
ssc@pc-ssc:~/Dropbox/Libro_FDE/Capitulo10/P00$ java EjemploAlumno1
El alumno Bruno Neumann tiene el numero de matricula: 1111111108
ssc@pc-ssc:~/Dropbox/Libro_FDE/Capitulo10/P00$
  
```

5. A continuación se presenta un ejemplo que es modelado usando UML mediante un diagrama de clases y luego se construye un programa basado en este diseño usando BlueJ.

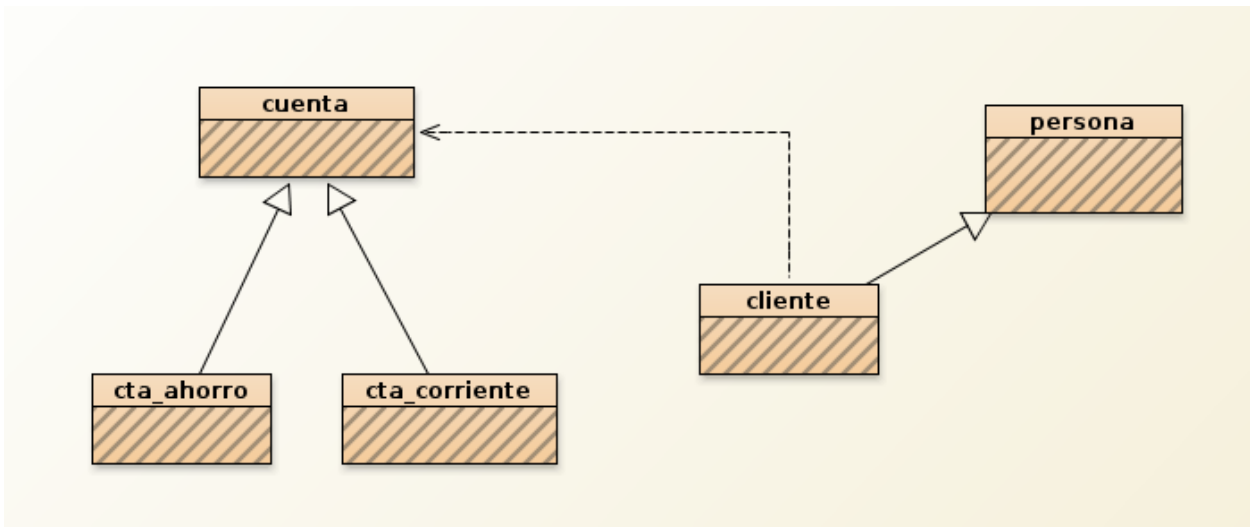
Una *persona* siendo *cliente* de un banco puede tener una *cuenta*, que puede ser del tipo *corriente* o de *ahorro*. Dichas cuentas, en general, tienen un código de cuenta, un saldo y se les puede depositar dinero o girar dinero, lo cual obviamente afecta el valor actual del saldo. Además una cuenta corriente posee un tope máximo de dinero que puede ser girado en cada transacción de \$500.000. Las cuentas de ahorro por su parte tienen un contador de giros que se realicen, pues no pueden hacerse más de 5.

Por otro lado sabemos que toda persona tiene al menos un nombre y apellido, más un rut. Los clientes además tienen un código de cliente y tienen una cuenta asociada, ya sea de ahorro o una cuenta corriente.

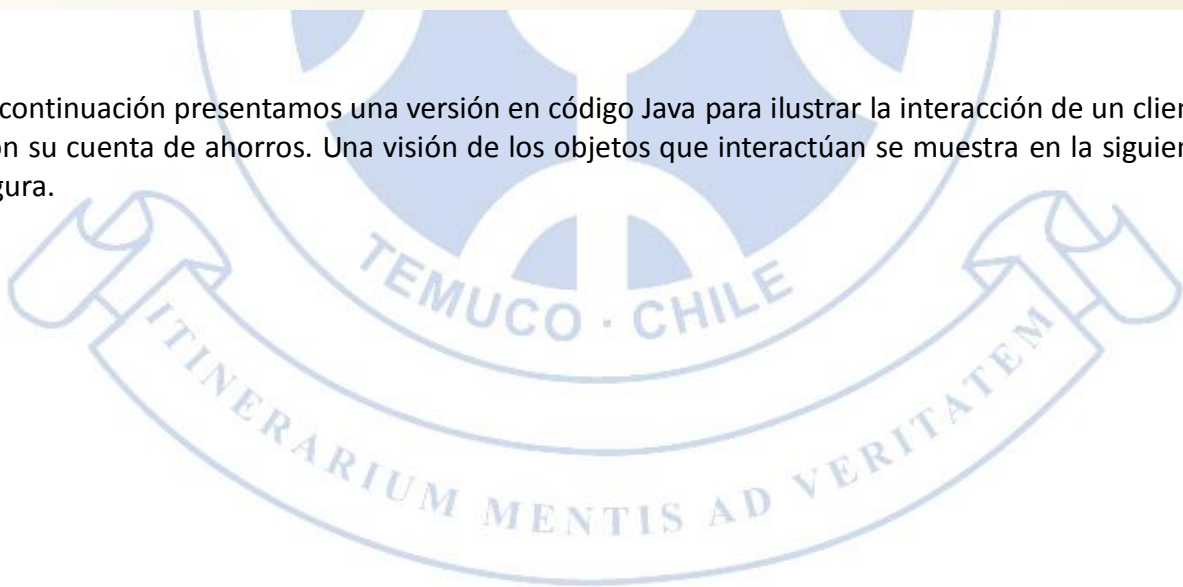
Analizando el caso anterior, se tienen las siguientes clases, con sus atributos y operaciones:

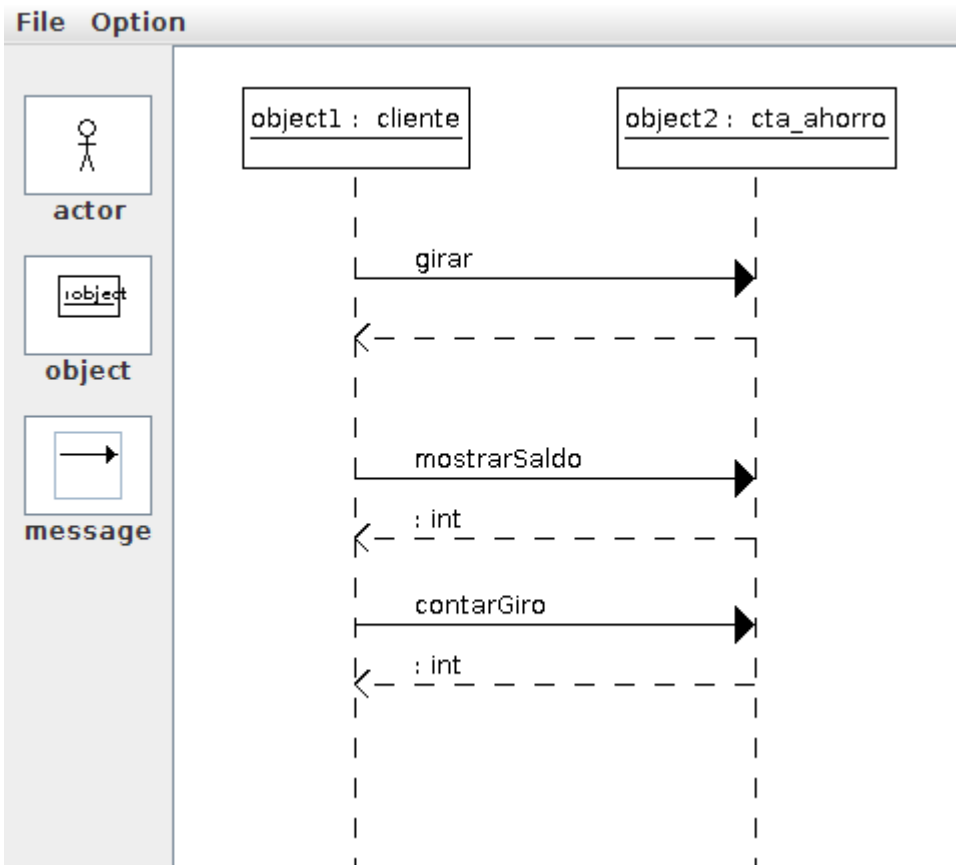
- Clase persona, atributos: nombre, apellido, rut. Operaciones: mostrar sus datos.
- Clase cliente es un tipo de persona, atributos: código de cliente, cuenta asociada. Operaciones: mostrar sus datos.
- Clase cuenta, atributos: código de cuenta, saldo. Operaciones: girar y depositar dinero.
- Clase cuenta corriente es un tipo de cuenta, atributos: tope máximo de giro. Operaciones: verificar giro.
- Clase cuenta de ahorro es un tipo de cuenta, atributos: contador de giros. Operaciones: contar número de giros.

Una aproximación del modelado del caso anterior usando diagramas de clases UML sería como se muestra la siguiente figura.



A continuación presentamos una versión en código Java para ilustrar la interacción de un cliente con su cuenta de ahorros. Una visión de los objetos que interactúan se muestra en la siguiente figura.





AQUÍ LOS CÓDIGOS DE CADA CLASE Y LUEGO EN EL MAIN CREAMOS LOS OBJETOS Y USAMOS ALGUNAS DE LAS OPERACIONES, PARA DEPOSITAR, GIRAR, ETC.

### 10.8 Ejercicios propuestos

#### i.) Conceptos

1. Se le pide que sea capaz de reconocer objetos, atributos y sus operaciones. Relacionar y diferenciar el concepto de Clases y Objetos. A partir de las cosas presentes en la sala de clases, reconozca al menos 3 objetos: Establezca 2 atributos, con sus valores y reconozca 2 operaciones. Luego establezca las clases a las cuales pertenecerían sus objetos.
2. Si bien se ha dicho en las secciones anteriores que la herencia permite *definir una clase a partir de una o más clases* ya existentes. Estas modificaciones consisten habitualmente en *añadir nuevos miembros* (atributos u operaciones), a la clase que se está definiendo. También es posible *redefinir* atributos u operaciones ya existentes, así como también es posible eliminar atributos u operaciones, pero esto último indicaría un mal modelado de las clases. ¿Por qué? Justifique su respuesta, puede usar esquemas o diagramas de clases para su justificación.
3. Ítem de Verdadero o Falso. Responda si las siguientes aseveraciones son verdaderas o

falsas (V o F). En caso de considerarla falsa justifique su respuesta, caso contrario, dé un ejemplo que respalde su respuesta.

- \_\_\_ Todo objeto posee atributos con valores propios.
- \_\_\_ El estado de un objeto puede variar mientras éste exista en memoria.
- \_\_\_ Todo objeto pertenece a una o más clases.
- \_\_\_ Un objeto representa genéricamente a un grupo de clases con características comunes.

## ii.) Programación con Java usando POO

1. Construir un programa que contenga una clase que contenga los atributos y al menos 2 métodos que permitan calcular las siguientes series matemáticas dado el valor  $n$ .

$$S1 = 1^2 + 2^2 + 3^2 + 4^2 + \dots + (n-1)^2 + n^2, n = 0, 1, 2, \dots$$

### 10.9 Comentarios finales

Al finalizar este último capítulo del libro, es bueno hacer un resumen de lo mencionado en las secciones anteriores.

En la sección 10.1 y 10.2 se presentaron los conceptos básicos de la POO entre los cuales destacan el concepto de *objeto* y *clase*, así como los conceptos, elementos y características que los rodean. La sección 10.3 intentó argumentar y mostrar las principales diferencias que existen entre dos formas distintas de enfrentar el análisis, diseño e implementación de una solución, éstas son la programación estructurada y la POO.

La sección 10.4 presentó los aspectos básicos del diseño de software con POO y cómo se pueden representar las soluciones usando dos diagramas del UML, los diagramas de clases y de secuencias. Hacemos nuevamente el énfasis de que esta revisión es un vistazo muy elemental del UML y el modelado de software con POO, cuyos conceptos avanzados deberán ser tratados con mucho más detalle en cursos de la especialidad.

Dentro la de sección 10.5 se establecieron los conceptos básicos del lenguaje Java, su funcionamiento interno y algunos aspectos técnicos. Es en la sección 10.6 dónde se plantean los elementos básicos que permiten crear programas computacionales con el lenguaje Java usando el paradigma de la POO.

Con el fin de resumir las principales ideas y conceptos discutidos a lo largo de este capítulos que se presenta en la siguiente figura un mapa conceptual. Este mapa presenta los que reconocemos como conceptos clave de la POO, las consideraciones fundamentales sobre el lenguaje de programación Java y los principales aspectos considerados a la hora de construir una solución basada en POO.

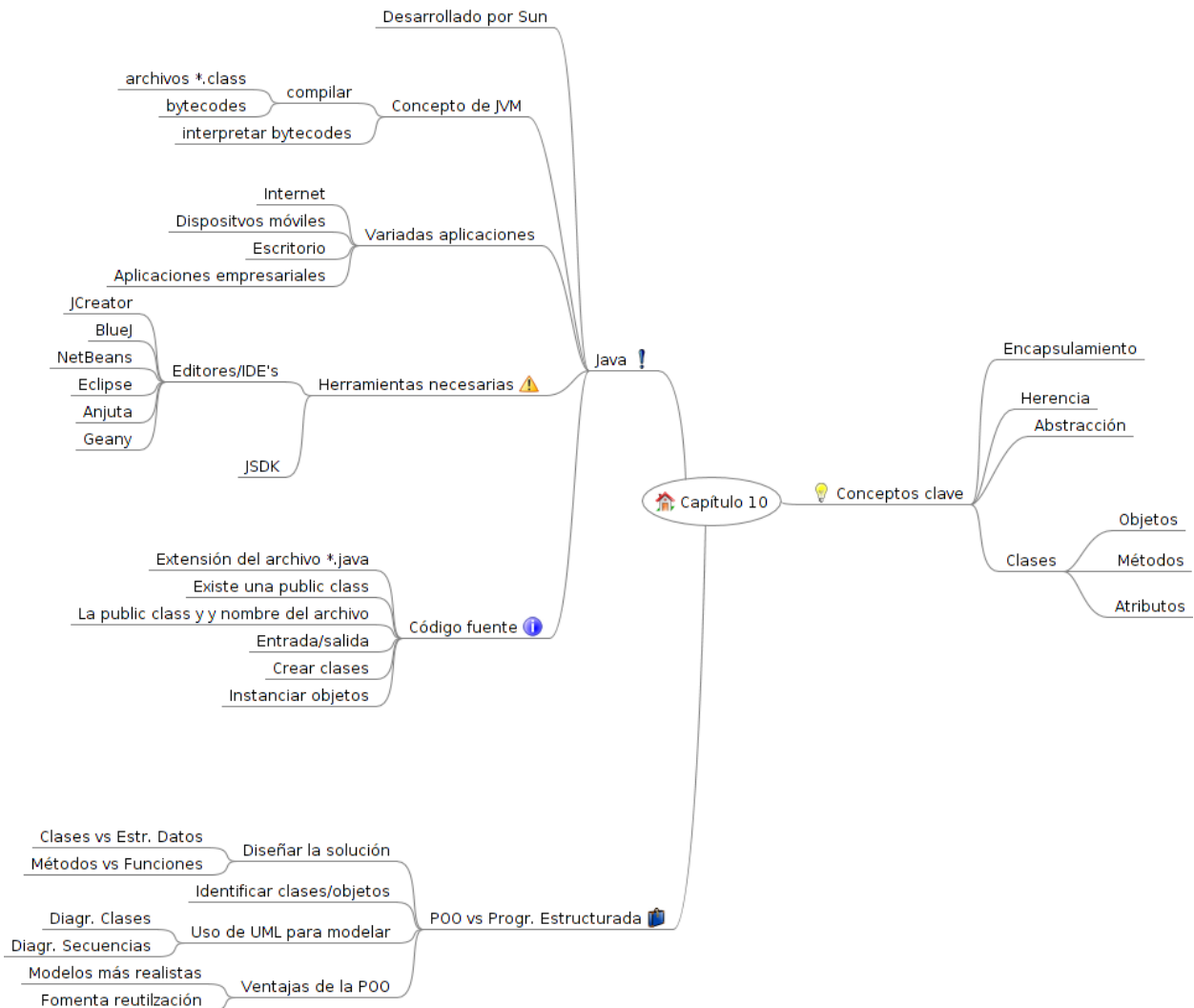


Figura 10.xx Mapa conceptual con resumen de los conceptos revisados.

### 10.10 Referencias

- Como programar en Java, 7ª Edición, Harvey M. Deitel, Paul J. Deitel, Pearson Educación, 2004.
- Metodologías orientadas a objetos y Java, David Santo Orcero, Revista Todo Linux, pp. 68-71, número 2, año 1, marzo 2001.
- Fundamentos del Diseño y la Programación Orientada a Objetos, Sergio Fernández, Editorial McGraw Hill, 1995.
- Fundamentos de programación: aprendizaje activo basado en casos : un enfoque moderno usando Java, UML, Objetos y Eclipse, 1ª Edición, Pearson Educación, 2006.
- UML para Programadores Java, Robert Martin, Pearson Educacion, 2004.
- Lenguajes de programación: diseño e implementación, Terrence W. Pratt, Marvin V. Zelkowitz, 3ª Edición, Prentice-Hall Hispanoamericana, 1998.