



Programación Orientada a Objetos. Nivel II

Mayo, 2018



Objetivos del nivel

- Aprender a usar clases y objetos.
- Aprender a programar métodos en un pseudolenguaje orientado a objetos.
- Entender los modificadores de acceso de las clases.
- Usar constructores.

Prerrequisitos del nivel

- Programación Orientada a Objetos Nivel I
- Lógica de Programación Nivel II

Acerca de este manual

Este manual pertenece al Centro de Asesoramiento y Desarrollo Informático C.A. (CADIF1). Para obtener más información sobre este u otros cursos visite nuestra sitio Web www.cadif1.com, escribanos a la dirección de correo cadi@cadif1.com o visítenos en nuestra sede ubicada en la Av. Pedro León Torres con calle 59, Centro Comercial Sotavento, piso 2 oficina 27, Barquisimeto estado Lara, Venezuela. Tf. 0251-7179247, 0251-4410268.

Las marcas mencionadas en este manual son propiedad de sus respectivos dueños.
Copyright 2018. Todos los derechos reservados.



Contenido del nivel

Capítulo 1. Definiendo Clases

- 1.1.- Nombre.
- 1.2.- Los Atributos.
- 1.3.- Declarar Objetos.

Capítulo 2. Los Objetos

- 2.1.- Instanciar Clases.
- 2.2.- Manipular Atributos.
- 2.3.- El Valor Null.

Capítulo 3. Los Métodos. Parte 1

- 3.1.- Definición de Métodos.
- 3.2.- Manipulación de Atributos.
- 3.3.- Ejecución de Métodos.

Capítulo 4. Atributos Compuestos

- 4.1.- Definición de Atributos Compuestos.
- 4.2.- Acceso Interno.
- 4.3.- Acceso Externo.

Capítulo 5. Los Métodos. Parte 2

- 5.1.- Los Parámetros.
- 5.2.- Paso de Parámetros.
- 5.3.- Valor de Retorno.

Capítulo 6. Modificadores de Acceso

- 6.1.- Modificador Público.
- 6.2.- El Ocultamiento.
- 6.3.- Modificador Privados.

Capítulo 7. Getters y Setters

- 7.1.- Getter.
- 7.2.- Setter.
- 7.3.- Uso de Getters y Setters.

Capítulo 8. Instrucciones Condicionales

- 8.1.- Operadores Relacionales.
- 8.2.- Condiciones.
- 8.3.- Comparando Objetos.

Capítulo 9. Instrucciones Iterativas

- 9.1.- Ciclos.
- 9.2.- Iterando en Los Métodos.
- 9.3.- Iterando al Llamar Métodos.

Capítulo 10. Constructores

- 10.1.- Concepto.
- 10.2.- Constructor Por Defecto.
- 10.3.- Constructor Con Parámetros.

Capítulo 11. Métodos. Parte 3

- 11.1.- Sobrecarga de Métodos.
- 11.2.- Ejecutar Métodos Sobre Cargados.
- 11.3.- Sobre Carga de Constructores.

Capítulo 12. Colecciones. Parte 1

- 12.1.- Definición de Una Colección.
- 12.2.- Agregar Objetos.
- 12.3.- Acceso a Atributos y Métodos.

Capítulo 13. Colecciones. Parte 2

- 13.1.- Atributos de Tipo Colección.
- 13.2.- Acceso Externo a Atributos y Métodos.
- 13.3.- Objetos Por Parámetro.

Capítulo 14. Herencia. Parte 1

- 14.1.- Concepto.
- 14.2.- Representación en un Diagrama de Clases.
- 14.3.- Herencia en Pseudocódigo.

Capítulo 15. Herencia. Parte 2

- 15.1.- Sobre Escritura de Métodos.
- 15.2.- Métodos Abstractos.
- 15.3.- Clases Abstractas.



Capítulo 1. DEFINIENDO CLASES

1.1.- Nombre

En la programación orientada a objetos luego del análisis del problema y de la aplicación de la abstracción, en la codificación el primer paso que se debe realizar es definir las clases. Cuando se define una clase se definir:

- el nombre de la clase.
- los atributos.
- los métodos.

El nombre de la clase debe ser suficiente general y explícita por si misma. Se usa un nombre en singular que generalice a los objetos que agrupa. También se estila que el nombre de la clase comience en una letra en mayúscula.

Todos los lenguajes de programación tienen sus reglas particulares para definir clases. En este curso, se usará un pseudocódigo para escribir programas orientados a objetos. Se usará la palabra "clase" (la mayoría de los lenguajes usan la palabra "class") seguido del nombre de la clase y terminando con las palabras "fin clase". Por ejemplo:

```
clase Alumno
```

```
fin clase
```

Entre las palabras "clase" y "fin clase" se definen los atributos y los métodos de la clase.

En un mismo programa se pueden definir tantas clases como sea necesario, dependiendo del problema que se esté intentando resolver. Cada clase debe tener su propio inicio y fin, por ejemplo:

```
clase Alumno
```

```
fin clase
```

```
clase Profesor
```

```
fin clase
```

1.2.- Los Atributos

Un atributo es una variable que se declara internamente en una clase. El nombre de un atributo debe ser coherente con el valor que almacenará y no puede haber 2 atributos con el mismo nombre en una clase. Los atributos de una clase pueden ser simples (o primitivos), es decir, de un tipo de dato básico como: entero, real, lógico, carácter, o también pueden ser más complejos (un objeto de otra clase). En esta primera parte del curso se usarán sólo tipos de datos simples. Por ejemplo:

```
clase Alumno
```

```
definir nombre como caracter
```

```
definir nota como numerico
```

```
definir aprobado como logico
```

```
fin clase
```

Siguiendo la misma metodología aprendida en lógica de programación, al analizar un problema se determinan las entradas y las salidas. Los atributos que se declaran en la clase dependerán de este análisis, por lo tanto, los atributos mínimos que debería tener una clase son las entradas y las salidas del problema.

1.3.- Declarar Objetos

Básicamente una clase es un tipo de dato. Un objeto es una variable de un tipo de dato de alguna clase definida previamente. Por lo tanto, definir un objeto no es más que definir una variable del tipo de dato de la clase a la cual pertenece. El nombre de la variable que identifica al objeto es arbitrario. Para definir variables, se usará la palabra reservada "definir", como en el siguiente ejemplo:

```
algoritmo ejemplo  
    definir alum1 como Alumno  
  
    fin algoritmo
```

En este ejemplo se está definiendo en el cuerpo principal del programa una variable (un objeto) con el nombre "alum1" que pertenece a la clase "Alumno" (es de tipo Alumno). Se pueden definir tantos objetos de la misma clase como sea necesario para resolver el problema, por ejemplo:

```
algoritmo ejemplo  
    definir alum1, alum2 como Alumno  
  
    fin algoritmo
```

También se pueden definir objetos de distintas clases, por ejemplo:

```
algoritmo ejemplo  
    definir alum1, alum2 como Alumno  
    definir prof1 como Profesor  
  
    fin algoritmo
```

Capítulo 2. LOS OBJETOS

2.1.- Instanciar Clases

La instancia es la creación de un objeto a partir de una clase. La palabra instanciar viene de una mala traducción de la palabra "instance", que en inglés significa ejemplar. Para crear un objeto se tiene que escribir una instrucción especial que puede ser distinta dependiendo el lenguaje de programación que se emplee, pero será algo parecido a esto:

```
algoritmo ejemplo
definir miCarro como Carro

miCarro= nuevo Carro()

fin algoritmo
```

Un ejemplo de una instancia de la clase Carro es la siguiente:



Con la palabra "nuevo" se crea la instancia de la clase que se coloca luego de la palabra "nuevo". La clase que se coloca luego de la palabra "nuevo" debe coincidir con el tipo de dato en la declaración del objeto (aunque más adelante se también puede ser una sub clase del tipo de dato del objeto).

En los lenguajes de programación donde se declaran las variables (como C++, Java, C#), se puede confundir la diferencia entre declaración e instantiación. En los lenguajes donde no se declaran las variables, solo se hace la instantiación (como JavaScript y Php).

2.2.- Manipular Atributos

A través de los objetos se puede acceder a los atributos de la clase a la que pertenece. Esto se logra haciendo uso del operador punto (".") (esto depende del lenguaje de programación). Por ejemplo:

```
clase Alumno
    definir nombre como caracter
    definir nota como numero
fin clase

algoritmo ejemplo
    definir alum1 como Alumno
    alum1 = nuevo Alumno()

    alum1.nombre = "jenny"
    alum1.nota = 20
fin algoritmo
```

Por medio de un objeto de una clase se puede realizar cualquier operación sobre los atributos de la clase: leer su valor por teclado, mostrar el valor, asignarle otra variable. Por ejemplo:

```

algoritmo ejemplo
  definir alum1 como Alumno
  alum1 = nuevo Alumno()

  mostrar "Introduzca el nombre: "
  leer alum1.nombre

  mostrar "El nombre del alumno es " alum1.nombre
fin algoritmo
```

Al tener varios objetos de la misma clase, cada objeto tiene los valores de sus atributos independientes de los valores de los atributos de los demás objetos. Por ejemplo:

```

algoritmo ejemplo
  definir alum1, alum2 como Alumno
  alum1= nuevo Alumno()
  alum2= nuevo Alumno()

  alum1.nota = 20
  alum2.nota = 12
fin algoritmo
```

En este ejemplo el objeto alum1 almacena en el atributo "nota" el valor 20, pero el objeto "alum2" almacena en el mismo atributo el valor 12. Cada objeto (aunque sean de la misma clase) posee un valor para el atributo "nota".

Un atributo puede tomar el valor de una variable siempre y cuando sea del mismo tipo. Por ejemplo:

```

algoritmo ejemplo
  definir alum1 como Alumno
  definir n como numero
  alum1= nuevo Alumno()
  n = 12
  alum1.nota = n
  mostrar "La nota del alumno es " alum1.nota
fin algoritmo
```

O entre valores de atributos de objetos:

```
algoritmo ejemplo
    definir alum1, alum2 como Alumno
    alum1= nuevo Alumno()
    alum2= nuevo Alumno()

    alum1.nota = 20
    alum2.nota = alum1.nota
    mostrar "La nota del alumno es " alum2.nota // mostrará 20
fin algoritmo
```

En el ejemplo, el objeto "alum2" está recibiendo en su atributo "nota" el mismo valor de ese atributo en el objeto "alum1". Esto puede hacerse entre cualquier atributo de cualquier objeto, siempre y cuando sean de tipos de datos compatibles.

2.3.- El Valor Null

En los lenguajes de programación donde se declaran las variables, un objeto no puede ser usado hasta que no sea instanciado. Como la declaración y la instancia se pueden hacer en momentos diferentes del algoritmo, un objeto mantiene un valor mientras que no sea instanciado. Ese valor es "null".

Si se intenta usar un objeto sin haberse instanciado se obtendría un error de ejecución en el programa. Por ejemplo:

```
algoritmo ejemplo
    definir alum1 como Alumno()

    alum1.nombre = "maría" // producirá un error
    alum1 = nuevo Alumno
fin algoritmo
```

Un objeto puede ser asignado a otro objeto:

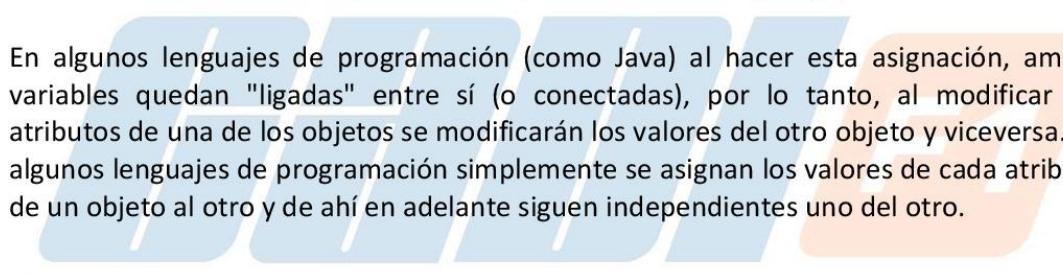
```
algoritmo ejemplo
    definir alum1, alum2 como Alumno
    alum1= nuevo Alumno()

    alum1.nota = 20
    alum1.nombre = "jenny"

    alum2 = alum1
    mostrar "el nombre del alumno es " alum2.nombre
fin algoritmo
```

En el ejemplo anterior, se está asignando al objeto "alum2" todo el objeto "alum1" (con todos sus atributos), por lo tanto, al mostrar el valor del atributo "nombre" mostrará el mismo valor que tiene el objeto "alum1" (mostrará el valor "jenny").

En algunos lenguajes de programación (como Java) al hacer esta asignación, ambas variables quedan "ligadas" entre sí (o conectadas), por lo tanto, al modificar los atributos de una de los objetos se modificarán los valores del otro objeto y viceversa. En algunos lenguajes de programación simplemente se asignan los valores de cada atributo de un objeto al otro y de ahí en adelante siguen independientes uno del otro.



Capítulo 3. LOS MÉTODOS. PARTE 1

3.1.- Definición de Métodos

Los métodos son las acciones que pueden realizar los objetos de una clase. Como se explicó en el primer capítulo, al definir una clase se definen los atributos y los métodos. Cada lenguaje de programación tiene sus propias reglas al momento de definir métodos. En este curso, se usarán las palabras reservadas "metodo" y "fin metodo". Entre estas 2 palabras puede haber cualquier tipo de instrucciones: entradas, salidas, asignación, comparación, etc. Por ejemplo:

```
clase Alumno
    definir nota como numero

    metodo ejemplo()
        mostrar "Hola mundo"
    fin metodo
fin clase
```

Una clase puede tener cualquier cantidad de métodos definida en su interior. El orden en que se definen los métodos es irrelevante. Por ejemplo:

```
clase Alumno
    definir nota como numero

    metodo saludar()
        mostrar "Hola que tal"
    fin metodo
    metodo despedir()
        mostrar "hasta la vista"
    fin metodo
fin clase
```

El nombre de un método puede ser cualquier identificador válido (según el lenguaje de programación que se esté usando), aunque debería ser coherente con la acción que éste

realizará. Es una buena práctica usar en el nombre de un método un verbo en infinitivo, por ejemplo: calcularNota, leerEntradas, determinarSueldo, etc.

3.2.- Manipulación de Atributos

Los métodos pueden acceder y manipular los atributos de la clase. Por ejemplo:

```
clase Alumno
    definir nota como numero

    metodo leerNota()
        mostrar "Introduzca la nota: "
        leer nota
    fin metodo
    metodo mostrarNota()
        mostrar "la nota es: " nota
    fin metodo
fin clase
```

3.3.- Ejecución de Métodos

Los métodos se definen en la clase pero no se ejecutarán a menos que sean llamados en el algún lugar. Normalmente es en el cuerpo principal del programa, aunque un método puede llamar a otro método. El llamado de un método se hace colocando el nombre del objeto, el operador "." y luego el nombre del método. Por ejemplo:

```
algoritmo ejemplo
    definir alum1 como Alumno
    alum1 = nuevo Alumno()

    alum1.leerNota()
    alum1.mostrarNota()
fin algoritmo
```

Capítulo 4. ATRIBUTOS COMPUESTOS

4.1.- Definición de Atributos Compuestos

Como se explicó en el capítulo 2, una clase puede tener atributos de tipos de datos simples (numérico, lógico, carácter, etc.) o atributos de tipos de datos complejos (otros objetos). Cuando una clase posee un atributo que sea un objeto de otra clase, la clase debe estar definida previamente. Por ejemplo:

```
clase Examen
    definir nota como entero
    definir duracion como entero
fin clase
clase Alumno
    definir nombre como caracter
    definir eval1 como Examen
fin clase
```

En este ejemplo, el atributo "eval1" es un objeto de la clase "Examen". También puede un atributo ser un objeto que a su vez tenga algún atributo que sea objeto de otra clase:

```
clase Fecha
    definir dia como entero
    definir mes como entero
    definir ano como entero
fin clase
clase Examen
    definir nota como entero
    definir duracion como entero
    definir fecha como Fecha
fin clase
definir Alumno
    definir nota como numero
    definir eval1 como Examen
fin clase
```

En el ejemplo anterior, la clase "Examen" tiene el atributo "fecha" que es un objeto de la clase "Fecha" y la clase "Alumno" tiene el atributo "eval1" que es un objeto de la clase "Examen".

4.2.- Acceso Interno

Internamente en un método de una clase se puede acceder a los atributos de algún atributo que sea un objeto:

```
clase Fecha
    definir dia como entero
    metodo mostrarDia()
        mostrar "El dia es " dia
    fin metodo
fin clase
clase Examen
    definir fecha como Fecha

    metodo mostrarFecha()
        mostrar "El dia de la evaluación es " fecha.dia
        // o se llama el método de la clase Fecha
        fecha.mostrarDia()
    fin metodo
fin clase
```

Si una clase tiene un atributo que es un objeto, que a su vez tiene una atributo del tipo de otra clase, se hace de la siguiente forma:

```

clase Alumno
  definir eval1 como Examen
  definir nombre como caracter

  metodo mostrarEvaluacion()
    mostrar "El alumno " nombre
    mostrar "Su dia del examen es " eval1.fecha.dia
    // a se puede ejecutar un método de la clase Examen
    eval1.mostrarFecha()
  fin metodo
fin clase

```

4.3.- Acceso Externo

En el cuerpo principal si se necesita acceder a algún atributo de un objeto interno, se realiza de la siguiente forma:

```

algoritmo ejemplo
  definir alum1 como Alumno
  definir f1 como Fecha
  definir exa como Examen

  // luego de instanciar cada objeto
  f1.dia = 10
  exa.fecha.dia = 20
  alum1.nombre = "jenny"
  alum1.eval1.fecha.dia = 15
fin algoritmo

```

También se pueden ejecutar los métodos de cualquiera de las clases a la que pertenece el objeto:

```
algoritmo ejemplo
  definir alum1 como Alumno
  definir f1 como Fecha
  definir exa como Examen

  // luego de instanciar cada objeto
  f1.mostrarDia()
  exa.fecha.mostrarDia()
  exa.mostrarFecha()
  alum1.eval1.fecha.mostrarDia()
  alum1.eval1.mostrarFecha()
  alum1.mostrarEvaluacion()

fin algoritmo
```



Capítulo 5. LOS MÉTODOS. PARTE 2

5.1.- Los Parámetros

Como se estudió en capítulos anteriores, los métodos son las acciones o funciones que pueden realizar los objetos de una clase. Un método es un conjunto de instrucciones identificadas bajo un nombre, que realiza una operación sobre el objeto que lo ejecuta. Es posible que para que un método realice su trabajo necesite recibir información del mundo exterior. Para lograr esto, el método debe recibir "parámetros". Los parámetros se colocan entre los paréntesis al lado del nombre del método. Por ejemplo, si un método de la clase Teléfono envía un mensaje de texto debe recibir del mundo exterior el número de teléfono y el mensaje de texto. El código sería como el siguiente:

```
clase Telefono  
  
    metodo enviarMensaje( numero, mensaje)  
        // se envia el mensaje de texto usando el numero y el mensaje  
    fin metodo  
fin clase
```

Un método puede recibir tantos parámetros como haga falta. El orden en que se colocan los parámetros es irrelevante. El nombre de los parámetros debe ser un identificador válido.

5.2.- Paso de Parámetros

Cuando se ejecuta un método que tiene parámetros, los parámetros deben ser enviados al método cuando se solicita su ejecución. Siguiendo con el ejemplo de la clase Teléfono, cuando un teléfono va a enviar un mensaje de texto, se debe pasar el número de teléfono y el mensaje de texto:

```

algoritmo ejemplo
  definir tel como Telefono
  tel = nuevo Telefono()

  tel.enviarMensaje("04245006266","Hola mundo")
fin algoritmo

```

Los parámetros pueden ser pasados usando variables, por ejemplo:

```

algoritmo ejemplo
  definir numero, mensaje como caracter
  definir tel como Telefono
  tel = nuevo Telefono()

  numero = "04245006266"
  mensaje = "Hola mundo"
  tel.enviarMensaje(numero, mensaje)
fin algoritmo

```

5.3.- Valor de Retorno

Bajo ciertas circunstancias, un método debe informar al mundo exterior el resultado de la operación realizada. Para esto, el método debe retornar un valor, por ejemplo, si la clase Teléfono tiene un método para enviar un mensaje de texto, es posible que al ejecutar ese método se necesite verificar si el mensaje de texto se envió o no. Cuando un método, necesita retornar un valor, debe contener entre sus instrucciones la instrucción "retornar" en conjunto con el valor que retornará. La forma general de un método que retorna valor es la siguiente:

```

metodo ejemplo() como TipoDatos
  retornar valor
fin metodo

```

Donde "TipoDatos" es el tipo de dato del valor que va a retornar la función. Un método que retorna valor también puede recibir parámetros del mundo exterior.

Por ejemplo, si una clase producto tiene un método que calcula el iva del precio, debe retornar un valor de tipo real:

```
clase Producto
    definir precio como real

    metodo calcularIva() como real
        retornar precio*0.14
    fin metodo
fin clase
```

Cuando un método retorna valor se le denomina función. Al ejecutar un método que retorna valor, se pueden realizar algunas acciones que antes no se podían hacer, por ejemplo:

```
algoritmo ejemplo
    definir prod como Producto
    prod = nuevo Producto()

    prod.precio = 1000
    mostrar "El Iva del producto es " , prod.calcularIva()

    // tambien se puede guardar en un valor
    iva = prod.calcularIva()
    mostrar "El Iva del producto es " , iva
fin algoritmo
```

Capítulo 6. MODIFICADORES DE ACCESO

6.1.- Modificador Público

Como se explicó anteriormente, los métodos de una clase, por defecto, pueden acceder a los atributos que están definidos adentro de ésta. También se pudo acceder a los valores de los atributos declarando e instanciando un objeto de la clase. Pero, ¿cómo se puede controlar esta característica inherente en la programación orientada a objetos? para esto existen los modificadores de acceso.

Los modificadores de acceso permiten controlar a cuales elementos de un objeto se pueden acceder desde el mundo exterior. Los modificadores de acceso se especifican al definir los elementos de la clase, porque aplican tanto a los atributos como a los métodos.

En la mayoría de los lenguajes de programación existen 3 modificadores de acceso: público, privado y protegido. Algunos lenguajes agregan otros más. En este curso se van a usar sólo estos 3 modificadores. En este capítulo se trabajará con los modificadores público y privado, y en los capítulos de herencia se trabajará con el modificador "protegido".

Normalmente en los lenguajes de programación cuando no se establece explícitamente el modificador de acceso, se asume alguno por defecto. Esto varía de un lenguaje de programación a otro. En el pseudo lenguaje que se está utilizando en el curso, cuando no se establece el modificador de acceso se asume que es público.

Un atributo o método declarado público puede ser accedido desde un método de la misma clase y también desde el exterior del objeto. En la mayoría de los lenguajes de programación el modificador de acceso se coloca antes de la declaración del atributo o el método.

Por ejemplo:

```
clase Producto
    publico definir precio como real
    definir nombre como caracter
fin clase

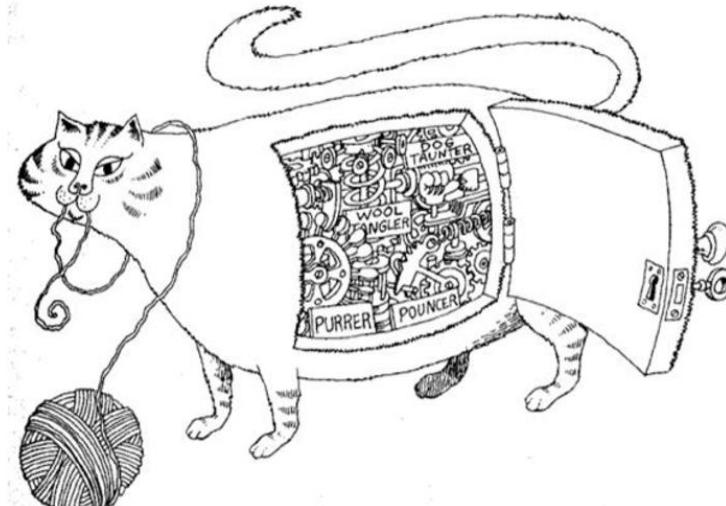
algoritmo ejemplo
    definir prod como Producto
    prod = nuevo Producto()

    // la siguiente asignación es válida porque es público
    prod.precio = 15000
    // la siguiente también es válida, porque se asume que es público
    prod.nombre = "monitor"
fin algoritmo
```

6.2.- El Ocultamiento

El ocultamiento se refiere a la capacidad de hacer que los atributos estén ocultos o escondidos, asegurando que éstos no se puedan modificar accidental o intencionalmente por otros objetos. La intención es proteger elementos sensibles del objeto, que pudieran dañarlo o impedir que se cumplan algunas reglas de negocio.

Los métodos también pueden ocultarse, aunque no es lo más común. Este principio permite que los objetos sean tratados como "cajas negras", es decir, que sean usados sin necesidad de conocer internamente como logran hacer lo que se solicita que hagan.



Por ejemplo, cuando se envía un correo electrónico, la fecha y hora en que el correo fue enviado no puede ser modificado por el usuario que lo visualiza. Esto asegura que un usuario pueda intentar engañar a otro usuario asegurando que envió un correo electrónico a cierta hora o día distinto al real.

También, en el contexto de un sistema bancario, una cuenta bancaria tiene un número, pertenece a un cliente y posee un saldo. El saldo de una cuenta bancaria no puede ser modificado arbitrariamente externamente. El saldo sólo puede ser modificado por los métodos "depositar" y "retirar".

6.3.- Modificador Privado

Un atributo o método al cual se le establezca el modificador de acceso "privado", solamente puede ser accedido por los métodos de la misma clase. No puede ser accedido por miembros de una clase heredada ni de clases externas, es decir, el acceso a los datos está restringido a los métodos de esa clase en particular. Este es el nivel más alto de protección de datos y es el que permite el principio de ocultación u ocultamiento en los objetos.

En la programación orientada a objetos los atributos privados de un objeto solo pueden ser modificados u obtenidos desde el exterior haciendo de métodos públicos que se crean para tal fin.

Por ejemplo:

```

clase Producto
    privado definir precio como real
    publico definir nombre como caracter
fin clase

algoritmo ejemplo
    definir prod como Producto
    prod = nuevo Producto()

    // la siguiente asignación es inválida porque precio es privado
    prod.precio = 15000
    // la siguiente asignación es válida porque el nombre es público
    prod.nombre = "monitor"
fin algoritmo
```

En el ejemplo de la cuenta bancaria, el saldo debería ser un atributo privado que sólo puede ser modificado con los métodos depositar y retirar:

```

clase Cuenta
    privado definir saldo como real

    publico metodo depositar( monto )
        saldo = saldo + monto
    fin metodo
    publico metodo retirar ( monto )
        saldo = saldo - monto
    fin metodo
fin clase
```

Por lo tanto, se deberían usar estos métodos para modificar el saldo y no acceder directamente al atributo saldo:

```
algoritmo banco
    definir c como Cuenta
    c = nuevo Cuenta()

    // es invalido porque saldo es privado
    c.saldo = 1000
    c.depositar ( 1000 )
    c.retirar ( 500 )
fin algoritmo
```



Capítulo 7. GETTERS Y SETTERS

7.1.- Getter

Como se vió en el capítulo anterior, el ocultamiento se logra utilizando el modificador de acceso privado. Un atributo privado sólo puede ser usado por los métodos de la clase. La pregunta es: cómo se puede acceder a este atributo para consultar su valor ?, la respuesta es: con los métodos getter.

Un getter (de la palabra en inglés "get" , cuya traducción al español es "obtener") es un método que tiene como objetivo retornar el valor de un atributo privado. Un método getter debe ser público y el tipo de dato de retorno debe coincidir con el tipo de dato del atributo cuyo valor va a retornar. De forma general los métodos getter tienen la siguiente estructura:

```
publico metodo getAtributo() como tipoDatos
    retornar atributo
fin metodo
```

Cada atributo privado debería tener su método getter (aunque no es obligatorio). Por ejemplo, si el saldo de una cuenta bancaria es privado se debe hacer un método getter:

```
clase Cuenta
    privado definir saldo como real

    publico metodo getSaldo() como real
        retornar saldo
    fin metodo
fin clase
algoritmo ejemplo
    definir c como Cuenta
    c = nuevo Cuenta()

    c.depositar ( 1500 )
    mostrar "el saldo de la cuenta es " c.getSaldo()
fin algoritmo
```

7.2.- Setter

Un setter (de la palabra en inglés "set" que traducido es "establecer" o "asignar") es un método que se utiliza para asignar un valor a un atributo privado. El setter hace el proceso contrario al getter. El método setter debe recibir por parámetro del mundo exterior el valor que se va a asignar al atributo. El formato general de un método setter es el siguiente:

```
publico metodo setAtributo ( nuevoValor )
    atributo = nuevoValor
fin metodo
```

Los métodos setter deben ser públicos. Lo usual es que todos los atributos privados tengan un método setter, pero no siempre será así. Existen atributos cuyo valor no puede ser asignado sino que es el resultado de la ejecución de otros métodos.

Por ejemplo, el precio de un producto puede ser cambiado directamente desde el mundo exterior:

```
clase Producto
    privado definir precio como real

    publico metodo setPrecio( nuevoPrecio )
        precio = nuevoPrecio
    fin metodo
fin clase

algoritmo ejemplo
    definir p como Producto
    p = nuevo Producto()
    p.setPrecio ( 1500 )
fin algoritmo
```

7.3.- Uso de Getters y Setters

Generalmente cada atributo privado debe tener su método Get y su método Set. Ambos deben tener acceso público. El método get debe ser un método que retorne el valor del atributo (no recibe parámetros y retorna un valor) y el método set un método para asignar el valor de un atributo (recibe un parámetro y no retorna nada).

Cuando se ocultan todos los atributos de una clase, la única forma de manipular los atributos y conocer sus valores es usando los getters y los setters.

Completando el ejemplo de la clase Producto:

```
clase Producto
    privado definir precio como real

        publico metodo getPrecio() como real
            retornar precio
        fin metodo
        publico metodo setPrecio( nuevoPrecio )
            precio = nuevoPrecio
        fin metodo
    fin clase

    algoritmo ejemplo
        definir p como Producto
        p = nuevo Producto()
        p.setPrecio ( 1500 )
        mostrar "el precio es " p.getPrecio()
    fin algoritmo
```

Capítulo 8. INSTRUCCIONES CONDICIONALES

8.1.- Operadores Relacionales

Una expresión lógica retorna 2 posibles valores: verdadero o falso. Para crear expresiones lógicas se deben usar operadores relacionales:

<code>==</code>	igual
<code><></code>	diferente
<code>></code>	mayor que
<code><</code>	menor que
<code><=</code>	menor o igual que
<code>>=</code>	mayor o igual que

Las expresiones lógicas se utilizan para crear instrucciones condicionales, ya sea en un método de una clase o en el programa principal.

`edad >= 18`
`precio < 1000`
`respuesta == "si"`

8.2.- Condiciones

Las condiciones permiten que el flujo del programa sea alterado. Las condiciones pueden estar internamente en los métodos de una clase o pueden estar en el cuerpo principal del programa. El siguiente ejemplo muestra cómo hacer una condición con el resultado de un método getter de una clase:

```

algoritmo ejemplo
  definir p1 como Producto

  p1 = nuevo Producto()
  si (p1.getPrecio() <= 1000)
    mostrar "El producto cuesta menos de 1000"
  sino
    mostrar "Los producto cuesta más de 1000"
  fin si
fin algoritmo
```

El siguiente es un ejemplo de una condición en un método de la clase Producto:

```

clase Producto
  privado definir precio como real
  privado definir exento como logico

  publico metodo calcularIva()
    si exento == falso entonces
      retornar precio*0.12
    sino
      retornar 0
    fin si
  fin metodo
fin clase
```

8.3.- Comparando Objetos

Generalmente se comparan los valores de los atributos de los objetos, ya sean de la misma clase o de clases diferentes, siempre y cuando sean valores de tipos de datos comparables. Por ejemplo, se puede comparar el precio de 2 productos instanciados:

```
algoritmo ejemplo
    definir p1, p2 como Producto

        p1 = nuevo Producto()
        p2 = nuevo Producto()

        si (p1.getPrecio() == p2.getPrecio() )
            mostrar "los productos tienen el mismo precio"
        sino
            mostrar "Los producto no tienen el mismo precio"
        fin si
    fin algoritmo
```

En ocasiones hace falta comparar 2 objetos completos, para determinar si son iguales atributo por atributo. En algunos lenguajes de programación hace falta crear un método especial para comparar los objetos completos, en cambio en otros lenguajes es permitido hacer la comparación directamente. Por ejemplo:

```
algoritmo ejemplo
    definir p1, p2 como Producto

        p1 = nuevo Producto()
        p2 = nuevo Producto()

        si (p1 == p2)
            mostrar "son los mismos productos"
        sino
            mostrar "no son los mismos productos"
        fin si
    fin algoritmo
```

Capítulo 9. INSTRUCCIONES ITERATIVAS

9.1.- Ciclos

Muchos de los algoritmos que se ejecutan en los programas necesitan que se repitan parte de sus instrucciones. Para lograr esto se tienen 2 opciones: copiar y pegar las instrucciones para que se repitan en el algoritmo o utilizar instrucciones iterativas.

Una instrucción iterativa o ciclo es una instrucción que se coloca en el algoritmo, que le indica que ciertas instrucciones debe repetirse. Aunque las instrucciones iterativas son parte fundamental de la programación estructurada, también forman parte de los algoritmos en la programación orientada objetos.

Los ciclos pueden estar adentro de los métodos o pueden estar en el cuerpo principal o en ambos lugares.

Igualmente hay 3 tipos de ciclos: para, mientras y repetir hasta (en la mayoría de los lenguajes de programación son llamados for, while y do while). En la siguiente imagen se muestra un ejemplo de un flujo de datos con ciclos:

9.2.- Iterando en Los Métodos

Como se explicó anteriormente, los ciclos pueden estar adentro de los métodos (en cualquier método de la clase), por ejemplo:

```
clase Alumno
    publico metodo leerNotas()
        para i = 1 hasta 3 hacer
            mostrar "Introduzca una nota:"
            leer nota
        fin para
    fin metodo
fin clase
```

En los métodos puede haber cualquier tipo de ciclo (para, mientras o repetir) y puede haber varios métodos con ciclos:

```
clase Alumno
    publico metodo saludar()
        repetir
            mostrar "hola..."
            hasta (condicion cualquiera)
        fin metodo
    publico metodo leerNotas()
        para i = 1 hasta 3 hacer
            mostrar "Introduzca una nota:"
            leer nota
        fin para
    fin metodo
fin clase
```

Al ejecutar en el cuerpo principal el método que tenga un ciclo, se ejecutaran las instrucciones que están adentro del éste:

```
algoritmo practica
    definir alum como Alumno

    alum.leerNotas() // se leeran 3 notas
fin algoritmo
```

9.3.- Iterando al Llamar Métodos

Los ciclos pueden estar en el cuerpo principal, lo que implicará que cierto(s) método(s) se ejecutaran varias veces, por ejemplo:

```

clase Alumno
    publico metodo leerNota()
        mostrar "Introduzca una nota:"
        leer nota
    fin metodo
fin clase
algoritmo ejemploCiclos
    definir alum como Alumno
    // se ejecuta el método leerNota 3 veces
    para i=1 hasta 3 hacer
        alum.leerNota()
    fin para
fin algoritmo

```

En el ejemplo anterior el método "leerNota" no tiene un ciclo, porque el ciclo está en el cuerpo principal, por lo tanto se conseguirá el mismo resultado: se leerán 3 notas de un alumno. Es posible que existe un ciclo adentro de un método y también en el cuerpo principal, por ejemplo:

```

clase Alumno
    publico metodo leerNota()
        para i=1 hasta 3
            mostrar "saludar..."
        fin para
    fin metodo
fin clase
algoritmo ejemploCiclos
    definir alum como Alumno
    para i=1 hasta 3 hacer
        alum.saludar()
    fin para
fin algoritmo

```

En el ejemplo anterior, el método "saludar" de la clase Alumno tiene un ciclo para mostrar 3 mensajes. Además, en el cuerpo principal del programa hay un ciclo que ejecuta 3 veces el método "saludar", por lo tanto, el mensaje de saludo aparecerá 9 veces (3 veces por cada llamado al método).

Capítulo 10. CONSTRUCTORES

10.1.- Concepto

En programación orientada a objetos, un constructor es un método que se ejecuta automáticamente al crear un objeto (instanciar). El uso típico que se le da a los constructores es inicializar el objeto que nace luego de ejecutar la instrucción "nuevo". Por ejemplo:

```
algoritmo constructores
  definir obj como Clase

  obj = nuevo Clase() // aqui se ejecuta el constructor de la clase
fin algoritmo
```

Si una clase no tiene un método constructor, igualmente los atributos del objeto que está naciendo serán inicializados a los valores por defecto según su tipo: los numéricos en cero, los lógicos en falso, los atributos de tipo carácter en una cadena vacía, los objetos en null. Por ejemplo:

```
clase Fecha
  privado definir dia,mes,año como numero
    // se asume que la clase tiene los métodos getter
  fin clase
  algoritmo ejemplo
    definir f como Fecha

    f = nuevo Fecha() // se ejecuta el constructor
    // se mostrará dia: 0, mes: 0, año 0
    mostrar "Dia: " , f.getDia()
    mostrar "Mes: " , f.getMes()
    mostrar "Año: " , f.getAño()
  fin algoritmo
```

En la mayoría de los lenguajes de programación el método constructor se identifica porque tiene el mismo nombre de la clase. En otros lenguajes se utiliza una palabra

reservada para identificarlo, por ejemplo: construct, constructor, init, create, entre otros.

Los constructores no deben tener tipo de retorno, es decir, no pueden ser funciones. Una clase puede tener más de un constructor siempre y cuando el lenguaje de programación permita hacer sobre carga de métodos (que se verá más adelante).

10.2.- Constructor Por Defecto

Un constructor por defecto es un constructor que no tiene parámetros. El uso típico que se le da a este constructor es inicializar el objeto con valores predeterminados o valores por defecto, generalmente distintos a los valores que por defecto serían asignados según el tipo de dato de cada atributo si la clase no tiene constructor. Los valores por defecto son arbitrarios, según sea necesario, por ejemplo:

```
clase Fecha
    privado definir dia,mes,año como numero

    // se asume que la clase tiene los métodos getter
    publico metodo Fecha()
        dia = 11
        mes = 1
        año = 1980
    fin metodo
fin clase
```

En el ejemplo anterior, al instanciar un objeto de la clase "Fecha" se ejecutará el constructor. A diferencia de cuando no hay un constructor, los atributos estarán inicializados con los valores asignados en el constructor:

```

algoritmo ejemplo
  definir f como Fecha
    // se ejecuta el constructor por defecto
    f = nuevo Fecha()
    // se mostrará dia: 11, mes: 1, año 1980
    mostrar "Dia: ", f.getDia()
    mostrar "Mes: ", f.getMes()
    mostrar "Año: ", f.getAño()
fin algoritmo

```

Si en el algoritmo se instancian varios objetos de una clase que tiene un constructor por defecto, todos los objetos serán inicializados con los mismo valores. Por ejemplo:

```

algoritmo ejemplo
  definir f1,f2 como Fecha

    // se ejecuta el constructor por defecto
    f1= nuevo Fecha()
    // se ejecuta el constructor por defecto
    f2 = nuevo Fecha()
    // al mostrar el dia, mes y año ambos objetos mostraran los mismos valores
fin algoritmo

```

IMPORTANTE: el hecho de que los atributos del objeto sean inicializados en el constructor no significa que los valores de los atributos no puedan cambiar posteriormente. Lo que se está logrando con el constructor es que los atributos tengan valores iniciales, pero estos valores pueden cambiar durante la vida del objeto.

10.3.- Constructor Con Parámetros

Los constructores por defecto no reciben parámetros y por tal razón deben inicializar los atributos con valores por defecto. Pero cuando se necesita que los atributos de un objeto sean inicializados con valores específicos, éstos valores deben ser pasados por parámetros al constructor. Por ejemplo:

```
clase Fecha
    privado definir dia,mes,año como numero

        // se asume que la clase tiene los métodos getter
    publico metodo Fecha(d, m, a)
        dia = d
        mes = m
        año = a
    fin metodo
fin clase
```

Cuando una clase tiene un constructor con parámetros, al instanciar la clase se deben pasar los parámetros que éste espera, por ejemplo:

```
algoritmo ejemplo
    definir f como Fecha

    f = nuevo Fecha(15,5,2004)

    // se mostrará dia: 15, mes: 5, año 2004
    mostrar "Dia: ", f.getDia()
    mostrar "Mes: ", f.getMes()
    mostrar "Año: ", f.getAño()
fin algoritmo
```

Los parámetros enviados al constructor pueden ser variables, por ejemplo:

```
algoritmo ejemplo
    definir f como Fecha
    definir x,y,z como numero

    x = 115
    y = 5
    z = 2004
    f = nuevo Fecha(x,y,z)
    // igual se mostrará dia: 15, mes: 5, año 2004
    mostrar "Dia: " , f.getDia()
    mostrar "Mes: " , f.getMes()
    mostrar "Año: " , f.getAño()
fin algoritmo
```

A diferencia de los constructores por defecto, si en el algoritmo se instancian varios objetos de una clase, cada objeto podrá ser inicializado con valores particulares. Por ejemplo:

```
algoritmo ejemplo
    definir f1,f2,f3 como Fecha
    definir x,y,z como numero

    x = 115
    y = 5
    z = 2004

    // cada nuevo objeto nacerá con valores particulares
    f1 = nuevo Fecha(15,5,2004)
    f2 = nuevo Fecha(20,8,1980)
    f3 = nuevo Fecha(x,y,z)
fin algoritmo
```

Capítulo 11. MÉTODOS. PARTE 3

11.1.- Variables Locales

Al implementar el algoritmo de un método es posible necesitar manejar algunos datos que sólo serán usados en el método, por lo tanto, no es necesario que sean atributos de la clase. Para esto se utilizan variables locales. Una variable local es aquella variable que es declarada internamente en una clase y que sólo puede ser usada adentro del método donde se declara. Por ejemplo:

```
clase Producto
    privado definir precio como numero

    publico metodo aumentarPrecio()
        definir aumento como numero

        aumento = precio*0.3
        precio = precio + aumento
    fin metodo
fin clase
```

En el ejemplo anterior, en el método "aumentarPrecio" se declara una variable local con el nombre "aumento", que almacenará el monto que se va a aumentar al precio de un producto. Esta variable sólo puede ser usada en el método "aumentarPrecio", a diferencia del atributo "precio" que se puede usar en cualquier método de la clase.

Un método puede tener una variable local con el mismo nombre de otra variable local de otro método, por ejemplo:

```

clase Producto
    privado definir precio como numero
    publico metodo mostrar()
        definir iva como numero
        // no se puede usar la variable "aumento"
        mostrar "El precio es " precio " y el aumento es " aumento
    fin metodo
    publico metodo aumentarPrecio()
        definir aumento como numero // variable local
        aumento = precio*0.3
        // no se puede usar la variable iva
        precio = precio + aumento + iva
    fin metodo
fin clase

```

11.2.- Sobrecarga de Métodos

En programación orientada a objetos la sobrecarga se refiere a la posibilidad de tener en una misma clase dos o más métodos con el mismo nombre pero funcionalidad distinta. Dicho de otro modo, dos o más métodos con el mismo nombre pero que realizan acciones con ciertas diferencias.

En tiempo de ejecución se usará una u otra versión del método, dependiendo de los parámetros usados. Es decir, cada método de igual nombre se diferenciará de los demás por el número y/o tipo de parámetros que éste tiene. Por ejemplo:

```

clase Alumno
    publico metodo calcularPromedio(notas1,notas2)
    ...
    fin metodo
    publico metodo calcularPromedio(nombre,notas1,notas2,notas3)
    ...
    fin metodo
    publico metodo calcularPromedio()
    ...
    fin metodo
fin clase

```

En el ejemplo anterior la clase Alumno posee 2 métodos con el nombre "calcularPromedio". La diferencia entre éstos es la cantidad de parámetros (uno de los métodos tiene 2 parámetros y el otro tiene 3 parámetros). En la ejecución, la forma de diferenciar cuál de los 2 métodos se ejecutará dependerá de la cantidad de parámetros que se le pase al método. Por ejemplo:

```
algoritmo ejemplo
definir alum1 como Alumno

    // se ejecutará la primera versión del método
    alum1.calcularPromedio(20,15)
    // se ejecutará la segunda versión del método
    alum1.calcularPromedio("jenny",12,16,18)
    // es un error porque no hay ninguna versión del método con 3 parámetros
    alum1.calcularPromedio(15,12,16)
fin algoritmo
```

11.3.- Sobre Carga de Constructores

Uno de los métodos que más comúnmente se sobre cargan en las clases son los constructores. La razón por la cual sucede esto es porque generalmente lo que identifica a un constructor es que debe tener el nombre de la clase, por lo tanto, si se necesita tener un constructor por defecto y también un constructor con parámetros, será obligatorio tener 2 métodos con el mismo nombre y por consecuencia se deben sobre cargar. Por ejemplo:

```

clase Alumno
    privado definir nombre como caracter
    privado definir nota como numero

    publico metodo Alumno() //constructor por defecto
        nombre = "sin nombre"
        nota = 0
    fin metodo
    publico metodo Alumno(nom, not) // constructor con parámetros
        nombre = nom
        nota = not
    fin metodo
fin clase

```

En el ejemplo anterior, la clase Alumno tiene 2 constructores, uno sin parámetros y otro con parámetros. Igualmente, la forma de diferenciar cuál constructor usar es en el llamado, es decir, en la instanciación de la clase. Por ejemplo:

```

algoritmo ejemplo
    definir alum1 como Alumno
    definir alum2 como Alumno

    // ejecuta al constructor por defecto
    alum1 = nuevo Alumno()
    // ejecuta al constructor con parámetros
    alum2 = nuevo Alumno("jenny",20)
fin algoritmo

```

En el ejemplo anterior, el objeto "alum1" es inicializado usando el constructor por defecto, por lo tanto, su nombre inicial será "sin nombre" y su nota inicial será "0", a diferencia del objeto "alum2" que es inicializado usando el constructor con parámetros, por lo tanto su nombre inicial será "jenny" y su nota inicial será "20".

Es importante destacar que los valores asignados en el constructor pueden ser modificados durante la vida del objeto. Lo que se está logrando es crear el objeto con valores indicados durante la instanciación, de modo que se ahorra el llamado de los métodos setter inmediatamente después de instanciar.

Capítulo 12. COLECCIONES. PARTE 1

12.1.- Definición de Una Colección

Una colección es una variable que permite almacenar varios objetos del mismo tipo con el fin de no declarar una variable para cada objeto. Esto es muy útil cuando se necesitan muchos objetos o cuando no se conoce de antemano cuantos objetos de cierta clase deben ser creados. La forma de manejar las colecciones varía de un lenguaje de programación a otro. Generalmente las colecciones son clases que son parte del lenguaje de programación y poseen algunas operaciones predeterminadas. En el pseudolenguaje utilizado en el curso se utilizará el tipo de dato "Coleccion" para indicar que una variable almacenará en su interior una serie de objetos de cierta clase. La forma general es:

```
definir col como Coleccion<Clase>
```

La variable "col" almacenará la colección de objetos de la clase "Clase". En los lenguajes de programación un arreglo o vector es también denominado como una colección.

Por ejemplo:

```
clase Alumno
    definir nombre como caracter
    definir nota como numero
fin clase

algoritmo ejemplo
    // alumnos es una colección de objetos Alumno
    definir alumnos como Coleccion<Alumno>
    // alum1 es un objeto de la clase Alumno
    definir alum1 como Alumno
fin algoritmo
```

12.2.- Agregar Objetos

Las colecciones por si mismo son objetos y como tal poseen algunas operaciones que permiten manipularlos. Para agregar objetos a la colección se utilizará el método "agregar", al cual se le pasa por parámetro el objeto que se desea agregar a la colección. Al agregar un objeto a una colección, éste es agregado al final (como si fuera una cola). Por ejemplo:

```
algoritmo ejemplo
    // alumnos es una colección de objetos Alumno
    definir alumnos como Coleccion<Alumno>
    // alum1 es un objeto de la clase Alumno
    definir alum1 como Alumno
    alum1 = nuevo Alumno()

    // se agrega el objeto alum1 a la colección
    alumnos.agregar(alum1)
    // se agrega un objeto nuevo a la colección
    alumnos.agregar( nuevo Alumno() )
fin algoritmo
```

Si es necesario agregar varios objetos a la colección se puede utilizar un ciclo repetitivo, por ejemplo:

```
algoritmo ejemplo
    // alumnos es una colección de objetos Alumno
    definir alumnos como Coleccion<Alumno>
    // alum1 es un objeto de la clase Alumno
    definir alum1 como Alumno
    para i = 1 hasta 5 hacer
        // se instancia un nuevo objeto
        alum1 = nuevo Alumno()
        // se agrega a la colección
        alumnos.agregar( alum1 )
    fin para
fin algoritmo
```

12.3.- Acceso a Atributos y Métodos

Cada vez que un objeto es agregado a una colección ocupa un lugar o posición adentro de ésta. A esta ubicación de cada objeto adentro de la colección se le denomina "la posición del objeto". El primer objeto que es agregado ocupa la primera posición, el segundo la segunda y así sucesivamente.

Luego de que un objeto está en una colección, para acceder a sus atributos y métodos se debe conocer primero en qué lugar de la colección está. También debe haber una forma de acceder al objeto. Para acceder a los objetos que están en una colección se utilizarán los corchetes y un número que indica la posición del objeto al cual se desea acceder, asumiendo que el primer objeto de la colección está en la ubicación 0 "cero". Por ejemplo:

```
algoritmo ejemplo
    definir alumnos como Coleccion<Alumno>
    definir alum1 como Alumno

    alumnos.agregar(nuevo Alumno())
    alum1 = alumnos[0]
fin algoritmo
```

En el ejemplo anterior, la variable "alum1" toma al objeto que está en la primera posición de la colección. Al acceder al objeto en una posición se puede acceder a sus métodos o atributos, por ejemplo:

```
algoritmo ejemplo
    definir alumnos como Coleccion<Alumno>

    alumnos.agregar(nuevo Alumno())

    alumnos[0].setNombre("jenny")
    mostrar "el primer alumno se llama ", alumnos[0].getNombre()
fin algoritmo
```

Las colecciones tienen una propiedad para conocer cuántos elementos contiene. El nombre de la propiedad es "tamaño". Si el primer elemento de la colección está en la posición cero, el último estará en la posición "tamaño-1".

```
algoritmo ejemplo
    definir alumnos como Coleccion<Alumno>
        alumnos.agregar(nuevo Alumno())
        alumnos[alumnos.tamano-1].setNombre("jenny")
        mostrar "el primer alumno se llama ", alumnos[0].getNombre()
    fin algoritmo
```

Otra forma de acceder al último elemento guardando en una variable la posición y luego acceder a la colección usando esa variable:

```
algoritmo ejemplo
    definir alumnos como Coleccion<Alumno>
    definir ultimo como numero

    alumnos.agregar(nuevo Alumno())
    ultimo = alumnos.tamano-1
    // se asigna el nombre al último alumno
    alumnos[ultimo].setNombre("jenny")
    // se muestra el nombre del último alumno
    mostrar "el primer alumno se llama ", alumnos[ultimo].getNombre()
fin algoritmo
```

Se puede acceder a cada objeto de la colección usando un ciclo, por ejemplo:

```
algoritmo ejemplo
```

```
    mostrar "Se han inscrito " alumnos.tamano " alumnos"  
    mostrar "Los nombres de los alumnos son:"  
    para i=0 hasta alumnos.tamano-1 hacer  
        mostrar alumnos[i].getNombre()  
    fin para  
fin algoritmo
```



Capítulo 13. COLECCIONES. PARTE 2

13.1.- Atributos de Tipo Colección

En los ejemplos vistos anteriormente, las colecciones estan declaradas como variables en el programa principal. Sin embargo, tambien puede haber colecciones en la definición de una clase. Para lograrlo, al definir un atributo de una clase éste se define como una colección de objetos de otra clase.

La clase de la cual se define la colección (clase contenida) debe estar definida previamente a la clase que contendrá la colección de objetos (clase contenedora). En el programa principal se puede declarar un objeto de la clase contenedora o incluso una colección de objetos de la clase contenedora.

Por ejemplo:

```
// clase contenida
clase Alumno
    definir nombre como caracter
fin clase
// clase contenedora
clase Seccion
    definir alumnos como Coleccion<Alumno>
fin clase
algoritmo ejemplo
    // se define un objeto de la clase Seccion
    definir secc como Seccion
    // colección de objetos de la clase Seccion
    definir secciones como Coleccion<Seccion>
fin algoritmo
```

En la clase contenedora, los métodos pueden manipular la colección como cualquier otro atributo de la clase, tomando en cuenta las consideraciones explicadas anteriormente al tratar las colecciones. Por ejemplo:

```

clase Alumno
    definir nombre como caracter
fin clase
clase Seccion
    definir alumnos como Coleccion<Alumno>

    metodo mostrar_alumnos()
        para i=1 hasta alumnos.tamano-1 hacer
            mostrar alumnos[ i ].nombre
        fin para
    fin metodo
fin clase

```

Si los atributos de la clase contenida son privadas, se deben usar los getter y setter. Por ejemplo:

```

clase Alumno
    privado definir nombre como caracter
        // se hacen los getters y setters
fin clase
clase Seccion
    definir alumnos como Coleccion<Alumno>

    metodo mostrar_alumnos()
        para i=1 hasta alumnos.tamano-1 hacer
            mostrar alumnos[ i ].getNombre ()
        fin para
    fin metodo
fin clase

```

Para el ejemplo anterior, en el cuerpo principal de un programa, al tener un objeto de la clase Seccion y ejecutar el método "mostrar_alumnos", se ejecutará el ciclo que mostrará el nombre de todos los alumnos:

```

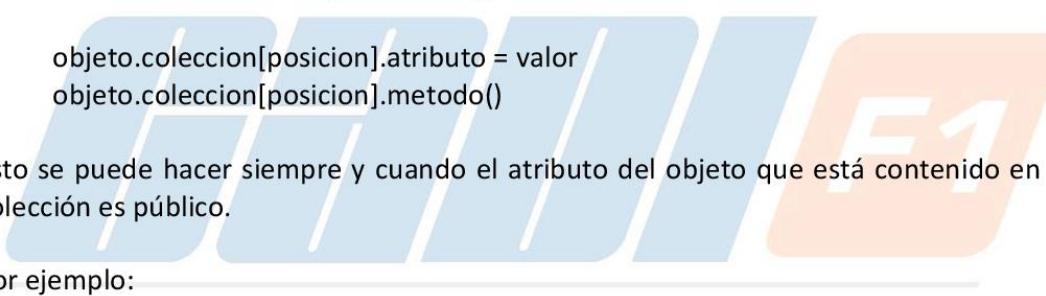
algoritmo ejemplo
    // objeto de la clase Seccion
    definir secc como Seccion
    secc = nuevo Seccion()

    secc.mostrar_alumnos()
fin algoritmo

```

13.2.- Acceso Externo a Atributos y Métodos

Cuando existe una colección definida internamente en una clase, en los métodos de la clase se puede acceder a cada uno de los elementos de la colección. Sin embargo, cuando se necesita acceder a los elementos de una colección que está adentro de una clase se debe usar de forma general la siguiente estructura:



```

objeto.coleccion[posicion].atributo = valor
objeto.coleccion[posicion].metodo()

```

Esto se puede hacer siempre y cuando el atributo del objeto que está contenido en la colección es público.

Por ejemplo:

```

clase Alumno
    publico definir nombre como caracter
fin clase
clase Seccion
    publico definir alumnos como Coleccion<Alumno>
fin clase

algoritmo ejemplo
    definir secc como Seccion

    secc.alumnos[0].nombre="jose"
    mostrar "El nombre es " secc.alumnos[0].nombre
fin algoritmo

```

Si los atributos de la clase de la cual hay una colección son privados, se deben usar los getter y setter de la clase. Por ejemplo:

```

clase Alumno
    privado definir nombre como caracter
        // se definen los getters y setters
    fin clase
clase Seccion
    publico definir alumnos como Coleccion<Alumno>
    fin clase
algoritmo ejemplo
    definir secc como Seccion

    secc.alumnos[0].setNombre("jose")
    mostrar "El nombre es " secc.alumnos[0].getNombre()
fin algoritmo
```

Si la colección es un atributo privado, deben existir métodos públicos que permitan acceder a los elementos de la colección. Por ejemplo:

```

clase Seccion
    privado definir alumnos como Coleccion<Alumno>

        // retorna el nombre del alumno que está en la posición i
    publico metodo getNombreAlumno( i ) como caracter
        retornar alumnos[ i ].getNombre()
    fin metodo

        // asigna el nombre al alumno que esta en la posición i
    publico metodo setNombreAlumno( i , nombre )
        alumnos[ i ].setNombre(nombre)
    fin metodo
fin clase
```

Para el ejemplo anterior, el cuerpo principal debe ejecutar los métodos públicos para acceder a los valores de los objetos que estan en la colección:

```

algoritmo ejemplo
  definir secc como Seccion

    // se le asigna el nombre "jose" al alumno que está en
    // la posición 0 de la colección
    secc.setNombreAlumno( 0, "jose" )

    // se muestra el nombre del alumno que está en
    // la posición 0 de la colección
    mostrar "El nombre es " secc.getNombreAlumno(0)
  fin algoritmo

```

13.3.- Objetos Por Parámetro

A un método se le puede enviar por parámetro un objeto. Así como también un método puede retornar un objeto. Esta práctica se hace imperativa en los casos en que las colecciones son atributos privados de una clase. Por ejemplo:

```

clase Seccion
  privado alumnos como Coleccion<Alumno>

    // este método agrega un objeto a la colección
    publico metodo agregarAlumno( alum )
      alumnos.agregar( alum )
    fin publico
    // este método retorna el objeto en la posición de la colección
    public metodo getAlumno( i ) como Alumno
      retornar alumnos[ i ]
    fin metodo
  fin clase

```

El siguiente ejemplo muestra como utilizar los métodos definidos anteriormente:

```
algoritmo ejemplo
    definir secc como Seccion
    definir alum como Alumno

    alum.setNombre("jose")
    // se agrega un objeto a la colección
    secc.agregarAlumno( alum )

    // se muestra el nombre del primer objeto agregado a la colección
    mostrar "El nombre del alumno es " secc.getAlumno( 0 ).getNombre()
fin algoritmo
```



Capítulo 14. HERENCIA. PARTE 1

14.1.- Definición de Sub Clases

La herencia se refiere a la capacidad de definir una clase en base a una clase ya existente, permitiendo así que la nueva clase ya posea todo lo que está definido en la anterior. A la nueva clase que se crea se le da el nombre de "subclase" o "clase derivada" y la clase que se utiliza como base se le denomina "clase base" o "super clase". Una clase hereda los métodos y atributos especificados por una Super Clase, por ende la subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos de la Super Clase.

La forma de definir una clase en base a otra (herencia) es muy particular en cada lenguaje de programación. Típicamente se utiliza la palabra reservada "extends" (que significa "extiende", que literalmente sería que una nueva clase extiende la funcionalidad de otra clase ya existente).

En el siguiente ejemplo se define una clase con el nombre "Persona" y se crea una sub clase con el nombre "Alumno":

```
// clase base Persona
clase Persona

fin clase
// sub clase de Persona
clase Alumno extiende Persona

fin clase
// otra sub clase de Persona
clase Profesor extiende Persona

fin clase
```

Una clase puede heredar de otra que ya hereda también de una super clase:

```

clase Persona

fin clase

clase Alumno extiende Persona

fin clase

// Universitario hereda de Alumno y transitivamente hereda de Persona
clase Universitario extiende Alumno

fin clase

```

14.2.- Acceso a Atributos y Métodos

Cuando una clase hereda de otra puede utilizar sus atributos y métodos como si fueran propios (siempre y cuando sean públicos). Por ejemplo:

```

clase Persona
    publico definir nombre como caracter
    privado definir edad como numero
    publico metodo saludar()
        mostrar "Hola a todos. Mi edad es " edad
    fin metodo
fin clase

clase Alumno extiende Persona
    definir codigo como caracter
    metodo mostrar_datos()
        saludar() // de la clase Persona
        mostrar "El alumno " nombre " tiene codigo " codigo
        edad = 20 // error porque edad es privado en la super clase
    fin metodo
fin clase

```

Al crear un objeto de la clase Persona, también se pueden ejecutar los métodos de la clase Alumno como los públicos de la clase Persona. Por ejemplo:

```
algoritmo ejemplo
    definir alum como Alumno
    definir per como Persona

        alum.nombre = "jose" // atributo heredado de la clase Persona
        alum.codigo = "123" // atributo propio de la clase Alumno
        alum.mostrar_datos()
        alum.saludar() // metodo heredado

        // atributo heredado de Persona pero es un error porque es privado
        alum.edad = 20
        // metodo propio
        per.saludar()
        // es un error, la clase Persona no tiene el método
        per.mostrar_datos()
        // error porque es privado
        per.edad =10
    fin algoritmo
```

14.3.- Modificador Protected

En ocasiones es necesario que una clase permita que ciertos atributos sean accesibles para las sub clases pero que sean tratados como atributos privados para el resto de las clases. Hasta ahora se conocen los modificadores de acceso "público" y "privado". Si un atributo es privado, sólo se puede acceder desde la misma clase, pero si es público todas las clases pueden acceder a éstos. Para lograr esta accesibilidad intermedia es necesario utilizar otro modificador de acceso denominado "protegido" (protected).

Por ejemplo:

```

clase Persona
    privado definir cedula como caracter
    publico definir nombre como caracter
    protegido definir edad como numero

    publico metodo saludar()
        mostrar "Hola a todos. Mi edad es " edad
    fin metodo
fin clase

clase Alumno extiende Persona
    publico definir codigo como caracter
    publico metodo mostrar_datos()
        saludar() // de la clase Persona
        mostrar "El alumno " nombre " tiene codigo " codigo
        edad = 20 // es válido porque es protegido
        cedula = "123456" // es invalido porque es privado
    fin metodo
fin clase

```

Igualmente al tener un objeto de la clase no se puede acceder a los atributos protegidos porque son tratados como si fueran privados:

```

algoritmo ejemplo
    definir alum como Alumno
    definir per como Persona

    alum.nombre = "jose"
    alum.codigo = "123" // atributo heredado
    alum.mostrar_datos()
    alum.saludar() // metodo heredado
    alum.edad = 20 // sigue siendo un error porque es protegido

    per.saludar() // metodo propio
    per.mostrar_datos() // es un error, la clase Persona no tiene el método
    per.edad = 10 // error porque es protegido
fin algoritmo

```

Capítulo 15. HERENCIA. PARTE 2

15.1.- Sobre Escritura de Métodos

La sobre escritura es la forma por la cual una clase que hereda de otra puede re-definir los métodos de su clase Padre, de esta manera puede crear nuevos métodos con el mismo nombre de su súper clase, es decir, escribir nuevamente el método, exactamente igual en nombre y parámetros, pero con distinta funcionalidad.

Se debe tener en cuenta que así como en la sobrecarga hay que fijarse en el orden, tipo y cantidad de parámetros, en la sobre escritura hay que fijarse en que la estructura del método de la subclase sea igual a la de su súper clase, no solo el mismo nombre sino el mismo número de parámetros y tipo de retorno.

Por ejemplo:

```
clase Persona
    publico definir nombre como caracter
    privado definir edad como numero

    publico metodo saludar()
        mostrar "Hola a todos. Mi edad es " edad
    fin metodo
fin clase

clase Alumno extiende Persona
    definir codigo como caracter

    publico metodo saludar() // sobre escribe el método en la súper clase
        mostrar "Soy un alumno"
    fin metodo
fin clase
```

Si se define un objeto de la clase Alumno y se ejecuta el método "saludar", se ejecutará el método de la sub clase. Por ejemplo:

```

algoritmo ejemplo
  definir alum como Alumno
  definir per como Persona

  alum.saludar() // se ejecuta el método de la clase Alumno
  per.saludar() // se ejecuta el método de la clase Persona
fin algoritmo

```

15.2.- Clases Abstractas

Una clase abstracta es una clase de la que no se pueden crear objetos. Su utilidad es permitir que otras clases deriven de ella, proporcionándoles un marco o modelo que deben seguir y algunos métodos de utilidad general.

Por ejemplo:

```

abstracta clase Persona
  publico definir nombre como caracter
  privado definir edad como numero

  publico metodo saludar()
    mostrar "Hola a todos. Mi edad es " edad
  fin metodo
fin clase
clase Alumno extiende Persona
  definir codigo como caracter

  publico metodo saludar()
    mostrar "Soy un alumno"
  fin metodo
fin clase

```

En el ejemplo anterior, la clase Persona fue definida como abstracta, por lo tanto, no se podrá instanciar. Por ejemplo:

```

algoritmo ejemplo
  definir alum como Alumno
  definir per como Persona

  alum = nuevo Alumno() // es válido
  per = nuevo Persona() // es un error
  alum.saludar() // se ejecuta el método de la clase Alumno
fin algoritmo

```

15.3.- Métodos Abstractos

Un método abstracto es un método que no está programado en la clase, sino que se encuentra declarado ahí para ser implementado en una subclase (sobre escrito). Por ejemplo:

```

abstracta clase Persona
  publico definir nombre como caracter
  privado definir edad como numero

  asbtracto publico metodo saludar()
fin clase
clase Alumno extiende Persona
  definir codigo como caracter

  publico metodo saludar()
    mostrar "Soy un alumno"
  fin metodo
fin clase

```

En cualquier sub-clase, que herede de una clase que contenga un método abstracto, este debe ser sobre escrito o debe declararse la clase también como abstracta, en cuyo caso la siguiente subclase es la que debe sobrescribir el método obligatoriamente. Si una clase tiene al menos un método abstracto es obligatorio que la clase sea abstracta, pero no necesariamente las clases abstractas deben tener métodos abstractos. En otras palabras, este tipo de clases permiten crear “método generales”, que recrean un comportamiento común entre las clase que se deriven; entonces su único fin es ser heredadas o extendidas por otras clases.