Universidad Autónoma del Estado de México
Centro Universitario UAEM Valle de Teotihuacán

CUVT

Universidad Autónoma
del Estado de México

Licenciatura en Ingeniería en Computación

Unidad de aprendizaje: Programación orientada a objetos

Clave	Horas teoría	Horas práctica	Total de horas	Créditos
L41007	3	2	5	8

Unidad de competencia III
"Conceptos avanzados de la Programación Orientada a Objetos"

L.I.A. Cecilia Bibiana Ramírez Waldo

Septiembre 2019.

Tipo de Unidad de Aprendizaje	Carácter de la Unidad de Aprendizaje	Núcleo de formación	Modalidad
Curso Laboratorio	Obligatoria	Sustantivo	Presencial

Índice

I. Guion explicativo …………………………….4

II. Objetivo de la unidad de aprendizaje ..……5

III. Competencias genéricas ……………………..6

IV. Estructura de la unidad de aprendizaje …...7

V. Secuencia didáctica …………………………...8

1. Relaciones entre clases………………………10
- Asociación.
- Agregación.
- Composición.
- Generalización.
- Dependencia.

2. Manejo de excepciones y asserts…………..17
- Definición
- Manejo
- Declaración de nuevos tipos de excepciones.
- Aserciones (Assetrs).

Índice

3. Interfaces…………………………………………24
- Clases abstractas.
- Interfaces.
- Implementación de interfaces.
- Representación UML

4. Herencia……………………………………….35
- Herencia múltiple
- El problema de identificadores duplicados.
- El problema del ancestro común.
- Herencia selectiva.
- Herencia virtual
- Variables y métodos final

5. Genericidad……………………………………..41
- Clases genéricas.
- Declaración de clases genéricas.
- Uso de clases genéricas

6. Persistencias, colecciones e iteraciones....46

VI. Conclusiones ……………………………………53

VII. Bibliografía …………………………………...55

Guion explicativo

- El estudiante conocerá los conceptos de la programación orientado a objetos y su implementación en un lenguaje apropiado, los cuales servirán de base para unidades de aprendizaje encaminadas al análisis, el diseño y la elaboración de aplicaciones informáticas.

Objetivo de la unidad de aprendizaje

- Comprender los aspectos históricos y tecnológicos que dan importancia al desarrollo de software orientado a objetos.

Competencias genéricas

- Contar con bases para comprometerse con la calidad.
- Lograr una comunicación oral y escrita eficaz en su propia lengua y en una segunda lengua.
- Utilizar eficazmente al menos un lenguaje de programación orientado a objetos.
- Responder eficazmente a nuevas situaciones informáticas.
- Comunicarse con expertos de otras áreas.
- Aplicar los conocimientos en la práctica.
- Crear nuevas ideas para la solución de problemas.
- Dar mantenimiento a software.
- Evaluar sistemas.
- Desarrollar software.

Estructura de la unidad de aprendizaje

- 1. Comprender los aspectos históricos y tecnológicos que dan importancia al desarrollo de software orientado a objetos.
- 2. Implementar los conceptos básicos de la POO (encapsulamiento, herencia y polimorfismo).
- 3. Implementar los conceptos avanzados de la POO.
- 4. Utilizar un lenguaje de modelado estándar para lograr una documentación apropiada del software.
- 5. Instrumentar sistemas de software que mediante la aplicación de POO.

Secuencia didáctica

- Unidad 1. Paradigmas de la programación
- Unidad 2. Conceptos básicos de la POO
- Unidad 3. Conceptos avanzados de la POO
- Unidad 4. Lenguaje modelado
- Unidad 5. Interfaz gráfica de usuario

1. Relaciones entre clases

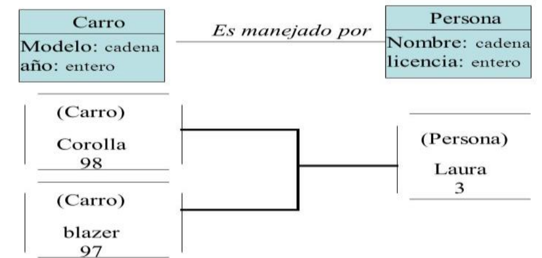
- Asociación.
- Agregación.
- Composición.
- Generalización.
- Dependencia.

Asociación

Es generalmente, una relación estructural entre clases. La multiplicidad de una asociación dice baste y que de eso dependerá si el atributo es una colección o simplemente una variable de referencia a un objeto. Especifica una relación semántica entre objetos no relacionados.

Este tipo de relaciones permiten crear asociaciones que capturen los participantes en una relación semántica.

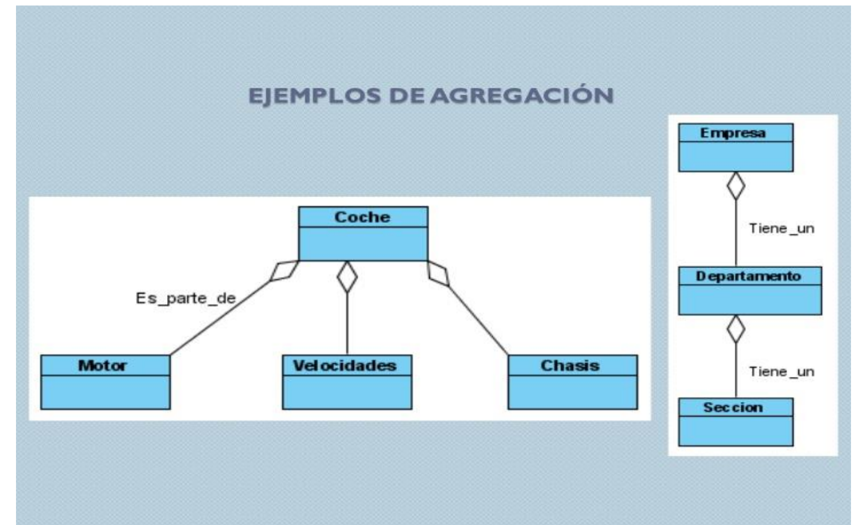
- Es una relación entre clases.
- Implica una dependencia semántica.
- Son relaciones del tipo “pertenece a” o “esta asociado con”.
- Se da cuenta una clase usa a otra clase para realizar algo.



Agregación

Es una relación que se derivo de la asociación, por ser igualmente estructural, es decir que contiene un atributo, que en todos los casos, será una colección, es decir un array , vector, collections, etc, y a demás de ello la clase que contiene la colección debe tener un método que agregue los elementos a la colección.

Tipos de agregación:



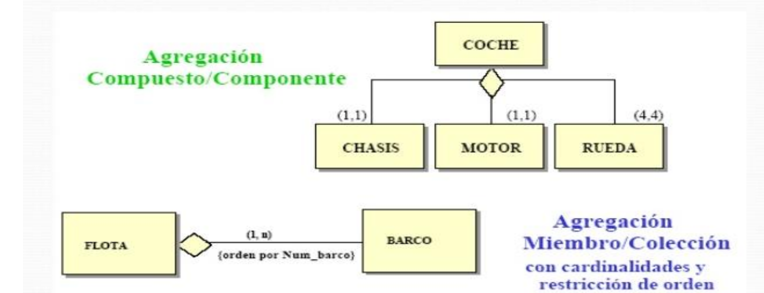
Agregación

Por contenido físico o valor:

- El contenedor contiene el objeto en si.
- Cuando creamos un objeto contenedor, se crean automáticamente los contenidos.

Agregación conceptual o por referencia:

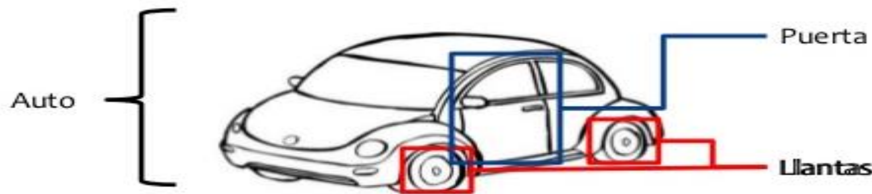
- Se tienen punteros a objetos.
- No hay un acoplamiento fuerte.
- Los objetos se crean y se destruyen dinámicamente.



Composición

Es un método cuando el objetos es construido.

En caso contrario, la composición es un tipo de relación **dependiente** en dónde un objeto más complejo es conformado por objetos más pequeños. En esta situación, la frase "*Tiene un*", debe tener sentido.



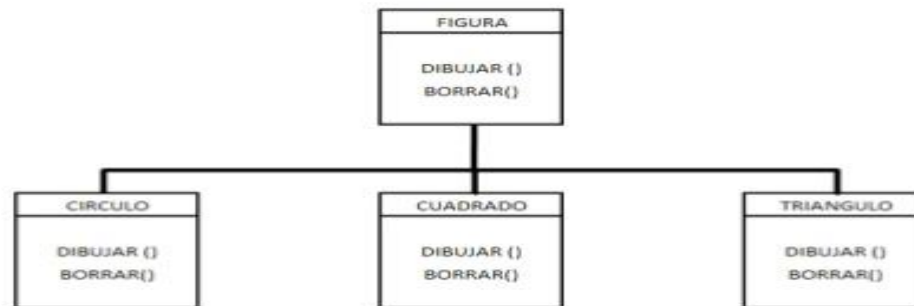
- Los objetos que componen a la clase contenedora, deben existir desde el principio. (También pueden ser creados en el constructor, no sólo al momento de declarar las variables como se muestra en el ejemplo).
- No hay momento (No debería) en que la **clase contenedora** pueda existir sin alguno de sus objetos componentes. Por lo que la existencia de estos objetos no debe ser abiertamente manipulada desde el exterior de la clase.

Generalización

Es una relación de herencia. Se puede decir que es un relación "es un tipo de"

La generalización consiste en crear una superclase más general dada una o varias clases existentes. Una forma común de descubrir la herencia es mediante la detección de miembros (atributos o funciones) repetidos en distintas clases.

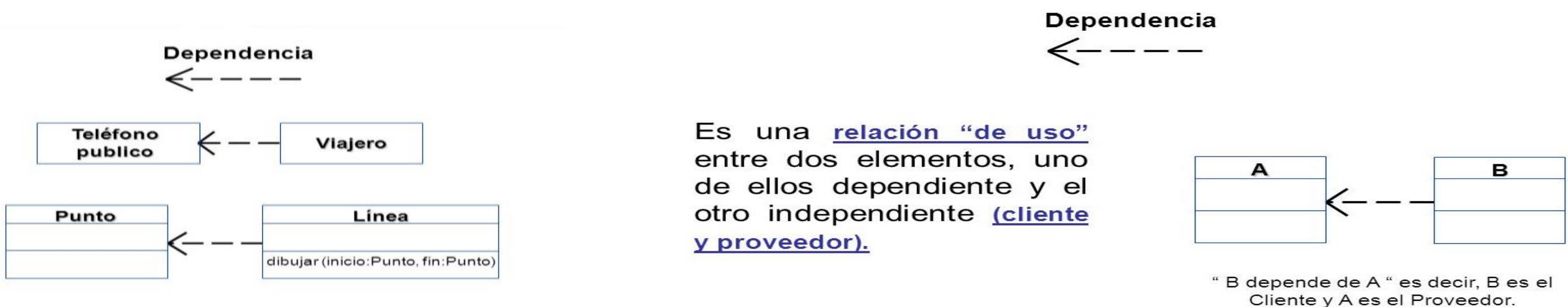
- Esto suele indicar que dichas clases comparten algo en común, o lo que es lo mismo que existe una superclase común a todas ellas, en la que deberían residir esos atributos repetidos, extraídos mediante un proceso de factorización.



Dependencia

La dependencia de clases es un concepto de la programación orientada a objetos que nos indica la relación existente entre dos clases. Como su nombre indica nos está diciendo que una clase depende de otra para realizar su funcionamiento.

La dependencia entre dos clases declara que una de ellas necesita conocer acerca de la clase a la que utilizará. Por lo general se denomina a esta relación como la más débil. El tiempo establecido para esta relación es corto.



2. Manejo de excepciones y asserts

- Definición
- Manejo
- Declaración de nuevos tipos de excepciones.
- Aserciones (Assetrs).

Manejo de excepciones y asserts

Las operaciones que se pueden realizar con excepciones son:

- Captura y tratamiento (manejo),
- Lanzamiento y declaración.

Captura y tratamiento de excepciones

- Para capturar una excepción, se escribe el código que se quiere controlar en un bloque t r y ; se especifica la excepción que se desea capturar en una cláusula catch y se coloca el tratamiento para la misma en el bloque catch. La cláusula catch consiste en la palabra reservada catch, seguida por el tipo de excepción que se desea capturar y un nombre de parámetro encerrados entre paréntesis. El conjunto try/catch no se puede separar con sentencias intermedias.

- No se debe colocar una estructura try/catch para cada una de las instrucciones de un programa que pueda lanzar una excepción, sino agrupar las que estén relacionadas en un único bloque try. En estos casos, es decir, cuando las sentencias encerradas en un bloque try pueden producir más de una excepción, es posible poner varias cláusulas catch consecutivas; permite efectuar un tratamiento distinto para cada una de las excepciones.

- La búsqueda de la excepción originada se efectuará secuencialmente en cada una de las clausulas catch, por tanto no se debe colocar una clausula catch que capture excepciones de una superclase antes de otra que capture las de una de sus subclases.

Declarar la excepción

- Las excepciones pueden no ser tratadas (manejadas), posponiendo su tratamiento para que sea efectuado por el método llamador, pero cuando no pertenecen a la clase Runt imeException, es necesario listar los tipos de excepciones que se pasan en la cabecera del método.

- Las excepciones RuntimeException deben capturarse en el método donde se han producido; en caso contrario serán tratadas por el bloque t r y del método invocador más próximo que capture dicha excepción y sin necesidad de ser listadas en la cabecera del método.

Creación de excepciones

- Para crear una excepción se define una subclase de Exception que implemente un constructor con un parámetro de tipo String y sobreescriba el método getMessage () de la clase Throwable. Hay que tener en cuenta que la clase Exception no define ningún método, sólo hereda los definidos por Throwabl.

Métodos de la clase: Throwable

- public java.lang.Throwable fillInCtackTrace():

Devuelve un objeto con los métodos en ejecución en el momento de lanzarse la excepción.

- public java.lang.String getMessage():

Devuelve el mensaje descriptivo almacenado en una excepción.

- public void printCtackTrace():

Muestra por consola un mensaje con la clase de excepción, el mensaje descriptivo de la misma y una lista con los métodos en ejecución en el momento de lanzarse la excepción.

- public void printStackTrace(java.io.PrintStream pl) :

Envía el mensaje anteriormente especificado al flujo que se le especifique como parámetro.

- public java.lang.String tostring():

Devuelve una cadena descriptora de la excepción

3. Interfaces

- Clases abstractas.
- Interfaces.
- Implementación de interfaces.
- Representación UML

Interfaces

- Java incorpora una construcción del lenguaje mediante la declaración de interfaces, que permite enunciar un conjunto de constantes y de cabeceras de métodos abstractos; estos deben implementarse en las clases y contribuyen la interfaz de clases.

- En cierto modo, es una forma de declarar que todo los métodos de una clase son públicos y abstractos, con ellos se especifican el comportamiento común de todas las clases que implementen la interfaz (Agilar & Martínez, 2011).

Clases abstractas

- Las clases abstractas representan conceptos generales, engloban las características comunes de un conjunto de objetos.

- Declaran métodos y variables instancia, y normalmente tienen métodos abstractos; si una clase tiene métodos abstractos debe declararse abstracta; una característica importante de esta clase es que de ella no se puede definir objetos, es decir, no se puede instanciar de una clase abstracta; el compilador devuelve un erro siempre que se desea crear un objeto de dicha clase. (Agilar & Martínez, 2011).

Clases abstractas

El propósito de una clase abstracta es proporcionar una superclase a partir de la cual otras clases pueden heredar interfaces e implementaciones. Las clases a partir de las cuales se pueden crear instancias (objetos), se denominan clases concretas. Todas las clases vistas hasta este momento son clases concretas, significando que es posible crear instancias de la clase.

- Una clase se declara abstracta con la palabra reservada abstract.

- Una jerarquía de clases no necesita contener clases abstractas, sin embargo, muchos sistemas orientados a objetos tienen jerarquías de clases encabezadas por superclases abstractas.

Métodos abstractos.

- Una clase abstracta es una clase definida con el modificador abstract que puede contener métodos abstractos (métodos sin implementación) y definir una interfaz completa de programación. Los métodos abstractos se implementan en las subclases (Agilar & Martínez, 2011).

Interfaces

En esencia, una interfaz es un sistema o dispositivo que utiliza entidades no relacionadas que interactúan. Ejemplos de interfaces son un mando a distancia para televisión, que es una interfaz entre el espectador y un aparato de televisión, un navegador de Internet, que es una interfaz entre el internauta y la red Internet.

Las interfaces en Java tienen la propiedad de poder obtener un efecto similar a la herencia múltiple. Si se utiliza la palabra reservada extends para definir una subclase, las subclases sólo pueden tener una clase padre.

- Definición:

Una interfaz (interface) en Java es una descripción de comportamiento.

Interfaces

En la mayoría de los casos, sin embargo, se puede utilizar una interfaz de un modo similar a como se utiliza una clase abstracta.

Una interfaz Java define un conjunto de métodos, pero no las implementaciones, así como datos. Los datos, sin embargo, deben ser constantes y los métodos, como se acaba de indicar, sólo pueden tener declaraciones sin implementación.

Interfaces

- Una interfaz se puede considerar una clase abstracta totalmente y en ella hay que tener en cuenta que:
  - ❑Todos los miembros son públicos (no hay necesidad de declararlos públicos).
  - ❑Todos los métodos son abstractos (se especifica el descriptor del método y no hay ninguna necesidad de declararlos abstract).
  - ❑Todos los campos son static y final (proporcionan valores constantes Útiles).

Implementación de una interfaz

- Java **no** permite que una clase derive de dos o mas clases, es decir, no permite la herencia múltiple; sin embargo una clase, una clase si puede implementar mas de una interfaz y tener el comportamiento común de varias de ellas.

Para eso, sencillamente se escriben las interfaces separadas por comas a comunicación de la palabra reservada implementada; así la clase tiene que implementar los métodos de todos (Agilar & Martínez, 2011).

Implementación de una interfaz

Una definición de interfaz consta de dos componentes:

La declaración de la interfaz y el cuerpo de la interfaz.

declaración de interfaz

```
public interface CompararObjetos
{
```

cuerpo de la interíaz

declaración de constantes

```
public static final int MENOR    1;
public static final int [GUAI    0;
public static final int MAYOR  - 1;
```

declaración de métodos

```
public int comparar (CompararObetos otroObjeto);
}
```

Interfaz UML
Ejemplo: Jerarquía de figuras geométricas

<<interface>>
Figura
+getPerimetro(): double
+getArea(): double
+getCentro(): Punto
+getTipoFigura(): String
+desplazar(x: double, y: double): void

<<extends>>

<<interface>>
Poligono
getNLados(): int

<<interface>>
Oval

<<class>>
Triangulo

<<class>>
Hexagono

<<class>>
Rectangulo

<<class>>
Pentagono

<<class>>
Circulo

<<metaclass>>
Elipse

4. Herencia

- Herencia múltiple
- El problema de identificadores duplicados.
- El problema del ancestro común.
- Herencia selectiva.
- Herencia virtual
- Variables y métodos final

Herencia

- Herencia. Una clase de alto nivel o superclase puede especializarse en clases de más bajo nivel llamada subclase, la subclase hereda los atributos y el comportamiento y puede agregar su propios atributos y especializar su comportamiento.

Herencia

- Cuando heredamos de una clase, nuestros objetos entiende todos los mensajes declarados en ellas y en toda la ascendencia de clases. A medida que vamos extendiendo la jerarquía, ampliamos el conocimiento, agregando nuevo comportamiento heredado (Vivona, 2011).
- En terminología Java, la clase existente se denomina superclase. La clase derivada de la superclase se denomina la subclase. También se conoce a la superclase como clase padre y una subclase se conoce como clase hija, clase extendida o clase derivada.

Herencia múltiple

Capacidad de definir una clase de tal maneta que esta tenga mas de una súper clase directa y, por ende, y puede así adquirir de manera automática todos los atributos y métodos definidos en toda su superclase.

- La diferencia entre este tipo de herencia es el numero de subclases directa que tiene la subclase. Java no permite la utilización de herencia múltiple para la definición de nuevas clases, sin embargo, si posibilita la utilización de un concepto extremadamente importante como las interfaces, que posibilita la simulación de ciertas características de la herencia múltiple

Ventajas de la herencia

- Facilidad en la modificación de clases. Evita la modificación del código existente al utilizar la herencia para añadir nuevas características o cambiar características existentes.
- Extracción de comunidad de clases diferentes. Evita la duplicación de estructuras/código idéntico o similar en clases diferentes. Sencillamente, se extraen las partes comunes para formar otra clase y se permite que ésta sea heredada por las demás.
- Organización de objetos en jerarquía. Se forman grupos de objetos que conservan entre sí una relación .
- Adaptación de programas para trabajar en situaciones similares pero diferentes. Evita la escritura de grandes programas, si la aplicación, sistema informático, formato de datos o modo de operación es sólo ligeramente diferente, pues se debe utilizar la herencia para modificar el código existente.

VENTAJAS DE LA HERENCIA

- Una clase extendida es una clase compuesta con miembros de la superclase (miembros heredados) y miembros adicionales definidos en las subclases (miem-, bros añadidos). Los miembros que se heredan por una subclase son:
  - Las subclases heredan de las superclases los miembros declarados como public o protected .
  - Las subclases heredan aquellos miembros declarados sin especificador de acceso mientras que la subclase está en el mismo paquete que la superclase.
  - Las subclases no heredan un miembro de la superclase si la subclase declara un miembro con el mismo nombre. En el caso de las variables miembros, la variable miembro de la subclase oculta (hides) la correspondiente de la superclase.

- En el caso de métodos, el método de la subclase anula el de la superclase.
- Las subclases no heredan los miembros privados de la superclase.

5. Genericidad

- Clases genéricas.
- Declaración de clases genéricas.
- Uso de clases genéricas.

Generosidad

La programación orientada a objetos tiene como principales objetivos favorecer la confiabilidad, reusabilidad y extensibilidad del software. Adoptar el enfoque propuesto por la programación orientada a objetos implica:

En la etapa de diseño reducir la complejidad en base a la descomposición del problema en piezas más simples a partir de la identificación de objetos y su organización en una estructura de clases.

En la etapa de implementación utilizar un lenguaje que permita retener la estructura de clases identificada en la etapa de diseño y encapsular la representación interna de modo que sea inaccesible desde el exterior.

Generosidad

- El encapsulamiento permite usar una clase considerando qué funcionalidad brinda, sin tener en cuenta cómo la implementa.
- La herencia permite aumentar el nivel de abstracción mediante un proceso de clasificación en niveles.
- El proceso consiste en abstraer lo que es común y esencial en un conjunto de entidades para formar un concepto general que comprenda a todas.
- Una clase derivada puede pensarse como una especialización de una clase más general.
- Alternativamente podemos pensar a una clase derivada como una extensión de la clase base

Valores y funciones de clase

- En los programas que generemos usualmente intervendrán constantes: valores matemáticos como el número Pi, o valores propios de programa que nunca cambian.

- Si nunca cambian, lo adecuado será declararlos como constantes en lugar de cómo variables. Supongamos que queremos usar una constante como el número Pi y que usamos esta declaración:

```
//
public class Calculadora {
    private double PI = 3.1416;
    public void mostrarConstantePi () { System.out.println (PI); }
    ... constructor, métodos, ... código de la clase ...   }
```

Valores y funciones de clase

- Si creas un objeto de tipo Calculadora, comprobarás que puedes invocar el método mostrarConstantePi para que te muestre el valor por pantalla. No obstante, una declaración de este tipo presenta varios problemas.

- En primer lugar, lo declarado no funciona realmente como constante, sino como variable con un valor inicial. Prueba a establecer this.PI = 22; dentro del método y verás que es posible, porque lo declarado es una variable, no una constante.

- En segundo lugar, cada vez que creamos un objeto de tipo calculadora estamos usando un espacio de memoria para almacenar el valor 3.1416. Así, si tuviéramos diez objetos calculadora, tendríamos diez espacios de memoria ocupados con la misma información, lo cual resulta ineficiente

6. Persistencia, colecciones e iteradores

Persistencia

- De forma general podemos comprender que la persistencia es la capacidad que tiene el programador para que sus datos se conserven bien al finalizar la ejecución de un proceso específico, permitiendo así la reutilización de otros procesos.

Persistencia

- Cuando se toma el concepto de persistencia puede considerarse dentro de cualquier ambiente de desarrollo y programación tanto como de ejecución de elementos, ya que éste trata de los archivos o elementos que se encuentran en un sistema y pueden seguir en el equipo a pesar de las modificaciones, realización de procesos o cualquier otra tarea o rutina realizada por el programador o el usuario final.

Persistencia

- Esta propiedad de los objetos trasciende en el tiempo y/o el espacio, puede que éste tenga modificaciones o cambios, tanto de ubicación como de contenido, sin embargo su propiedad siempre persistirá en un modo visible y disponible para los usuarios que deseen trabajar con éste.

Persistencia

La persistencia de los objetos se puede realizar o comprender de diferentes maneras, estos enfoques constan de:

- Por clases
- Poco flexible
- Por creación
- Por marcas
- Por referencias

Persistencia

Por clases: es el uso de nueva sintaxis que permite la declaración de que una clase es persistente. Todos los elementos de esta clase son creados como persistentes.

Por marcas: todos los objetos se crean igual. Los objetos persistentes se marcan como tales después de su creación.

Poco flexible: en ocasiones interesa que algunos objetos sean persistentes y otros no.
.

Por creación: se extiende la sintaxis de creación de objetos para permitir la creación de objetos persistentes.

Por referencia: aplicado cuando uno o varios objetos se declaran como persistentes de forma explícita. Estos objetos se les hace referencia desde uno de los objetos anteriores y se declaran también como persistentes.

Colecciones e iteradores

- Las colecciones en Java son un ejemplo destacado de implementación de código reutilizable utilizando un lenguaje orientado a objetos.
- Todas las colecciones son genéricas.
- Los tipos abstractos de datos se definen como interfaces.
- Se implementan clases abstractas que permiten factorizar el comportamiento común a varias implementaciones.

- Las colecciones pueden ser, según se almacenen los objetos, de 2 tipos:
- ❖ Ordenadas:
  1. Pueden ser recorridas siguiendo un orden, ya sea por un índice como por el orden en el que se han insertado.
- ❖ Clasificadas:
  1. Los objetos están clasificados siguiendo un orden natural definido por la clase del objeto.

Conclusiones

- Con la implementación de los conceptos avanzados de la programación orientada a objetos, desarrollará aplicaciones usando un lenguaje de programación, brindando soluciones a problemáticas planteadas en un entorno real.

Por su atención, gracias.

Referencias bibliográficas

- Joyanes Aguilar L. Programación orientada a objetos, 1ra edición, Ed:McGrawHill.
- Roger S. Pressman, Ingeniería del Software, un enfoque practico. 5ta edición, Ed: McGraw
- Ian Sommerville, Ingeniería del Software. 7ma edición, Ed: Pearson. Hill.
- Joyanes Aguilar L. & Zahonero Martínez Ignacio, Programación en Java 6: Algoritmos, programacion orientada a objetos e interfaz grafica de usuario, (2011) 1ra edición, Ed:McGrawHill,.
- Cohoon James & Davidson Jack, Programación en JAVA 5.0, (2006), 1ra edición, Ed:McGrawHill,.
- Vivona Ignacio, Java denominaste el lenguaje líder en aplicaciones cliente servidor, 1ra Edición, ISBN: 978-987-1773-97-8.