UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CONTADURÍA Y ADMINISTRACIÓN

Licenciatura En Informática

Informática VI

(Programación Orientada a Objetos)

Autor: L.I. María de Lourdes Isabel Ponce Vásquez

SEPTIEMBRE 2020 - ENERO 2021

Contenido

Unidad 3. Herencia y Polimorfismo

Objetivos Específicos

- Comprender cómo pueden interrelacionarse diferentes objetos
- Reconocer la diferencia entre la asociación, composición y herencia
- Aprender a reutilizar código mediante herencia
- Definir qué es la herencia
- Definir clases abstractas para crear herencia
- Comprender la diferencia entre herencia simple y múltiple
- Utilizar interfaces para simular herencia múltiple
- Definir qué es el polimorfismo
- Utilizar métodos virtuales

3.1. Introducción

En la unidad anterior se dio el primer paso para la comprensión del paradigma orientado a objetos trabajando con clases y objetos y revisando la filosofía básica de organización de un programa como la interacción de componentes de software acoplados. El siguiente paso para aprender programación orientada a objetos es comprender la forma en que se relacionan las clases.

3.2. Relaciones entre Clases

Las clases y los objetos no existen de manera aislada, de hecho, los objetos por sí mismos no son muy interesantes. Los objetos contribuyen al comportamiento de un sistema colaborando con otros, por lo que, para comprender un problema, se debe comprender la forma en que se relacionan formando un diseño de clases.

Las relaciones se establecen por dos razones:

- porque comparten algo, a y b tienen los mismos atributos u operaciones;
- o porque tienen algún tipo de conexión semántica, a y b son independientes, pero colaboran en un contexto específico.

Las relaciones (asociación/agregación) poseen una cardinalidad o multiplicidad, que define el número de clases u objetos que participan en la relación.

La cardinalidad puede ser:

1	Exactamente uno
0..*	Cero a muchos
1..*	Uno a muchos
0..1	Cero o uno
5..8	Especifica rango (5, 6, 7 u 8)
4..7,9	Combinación (4, 5, 6, 7 ó 9)

Existen tres tipos principales de relaciones entre clases:

- La primera es la generalización/especialización (herencia), que denota una relación “es un” (is a). Por ejemplo, un obrero es un empleado, por lo que un obrero es una subclase especializada de una clase más general, la de empleado.

- La segunda es una agregación, composición o relación de todo/parte (whole/part), que denota una relación "parte de" o "tiene un". Así, un motor no es un tipo auto, es parte de un auto.
- La tercera es la asociación, que denota alguna dependencia semántica entre clases que de otro modo permanecerían independientes, como los alumnos y las inscripciones.

En los lenguajes de programación han evolucionado varios enfoques comunes para plasmar estas relaciones. Específicamente, la mayoría de los LOO ofrecen soporte directo para algunas combinaciones de las siguientes relaciones:

- Asociación
- Agregación
- Uso
- Tipos genéricos
- Metaclase
- Herencia

De estos seis tipos de relaciones entre clases, la asociación es la más general y con mayor debilidad semántica, es común que estas asociaciones se refinen hacia otras relaciones más concretas y la herencia es la más interesante.

3.3. Asociación

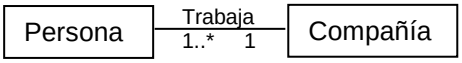
La ***asociación*** es una "*conexión física o conceptual entre objetos"*. Un objeto colabora con otros mediante asociaciones con éstos, donde un objeto (cliente) utiliza los servicios de otro (servidor).

Una asociación sólo denota una dependencia semántica y puede o no establecer la dirección de esta dependencia (es una relación bidireccional principalmente y en ocasiones unidireccional), y establece la forma en que una clase se relaciona con otra (denotando esta semántica nombrando el papel -rol- que desempeña cada clase en relación con la otra). Esta semántica es suficiente durante el análisis de un problema, momento en el que sólo se requiere identificar las dependencias. Mediante asociaciones se llega a plasmar quiénes participan en la relación, sus papeles y su cardinalidad o multiplicidad (uno a uno, uno a muchos, muchos a muchos).

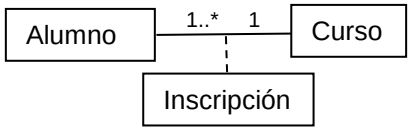
Las asociaciones aparecen como verbos en la declaración de un problema. Generalmente definen una relación de pertenencia. Se puede identificar con frases como "pertenece a", "es miembro de", "está asociado con", "trabaja para", etc.

Una asociación puede implementarse frecuentemente por agregación cíclica o relaciones "de uso" cíclicas; sin embargo, una asociación (que por definición implique bidireccionabilidad) se refina durante el diseño para ser una sola relación de agregación o "de uso", denotando una restricción acerca de la dirección de la asociación.

Por ejemplo, una persona *trabaja para* una compañía. Todas las ligas en una asociación conectan objetos de las mismas clases.



Las Clases de Asociación se agregan al modelo para representar relaciones muchos a muchos. Cuando una relación tiene atributos que no pertenecen a ninguno de los objetos de la asociación se usa una clase de asociación. Estas clases tienen sus propios atributos y/o métodos.

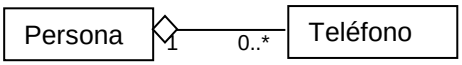


3.4. Agregación

La agregación es una relación que representa a los objetos compuestos de otros objetos. Las relaciones de agregación entre clases tienen un paralelismo directo con las relaciones de agregación entre los objetos correspondientes de esas clases.

La agregación también se conoce como una relación "tiene un" o "parte de", ya que los componentes son *"parte de"* un agregado, o un agregado "tiene un" (o varios) componentes. La ***agregación*** *es semánticamente un objeto extendido que se trata como una unidad en muchas operaciones*. La agregación cíclica es común.

En la implementación, un objeto no contiene a otro, en realidad hace referencia a él. El objeto "contenido" puede modificarse sin que esto afecte al contenedor. Los objetos contenidos pueden pertenecer a varios objetos a la vez.



En una agregación de ***contención por valor***, es un tipo de contención física que significa que el objeto contenido no existe independientemente de la instancia del objeto de la clase contenedora que lo encierra. Por el contrario, el tiempo de vida de ambos objetos es el mismo: cuando se crea una instancia del contendor, se crea también una de la contenida en. Cuando se destruye la contenedora, también se destruye la contenida.

También es posible otro tipo de agregación menos directo llamado ***contención por referencia***. En ella, la clase contenedora sigue denotando al todo, y una de sus partes sigue siendo instancia de otra clase, aunque se accede a esa parte indirectamente. En este caso, los tiempos de vida de ambos objetos ya no están tan ligados, ya que se pueden crear y destruir instancias de cada clase independientemente, en ese caso se debe definir una política para crear y destruir el objeto sólo por uno de los agentes que comparten la referencia a ese objeto.

La agregación establece una dirección en la relación todo/parte. Por ejemplo, un motor es parte de un auto y no al revés. La contención por valor no puede ser cíclica, es decir que ambos objetos no pueden ser físicamente partes de otro, aunque la contención por referencia si lo permite, teniendo cada objeto una referencia al otro.

La propiedad más significativa de la agregación es la ***transitividad***, que implica que, si A es parte de B y B es parte de C, entonces A es también parte de C. También se puede especificar la multiplicidad en caso de que la relación no sea de uno a uno.

Otra propiedad de la agregación es la propagación, que es la aplicación automática de una operación a sus partes, por ejemplo, al mover un objeto ventana se mueven todas sus partes. La propagación no se da en sentido contrario, de sus partes al todo.

La agregación es una forma especial de asociación, no un concepto independiente, si dos objetos se consideran generalmente por separado, aún cuando estén unidos es una asociación y no agregación. Para identificar esta relación se pueden hacer algunas pruebas:

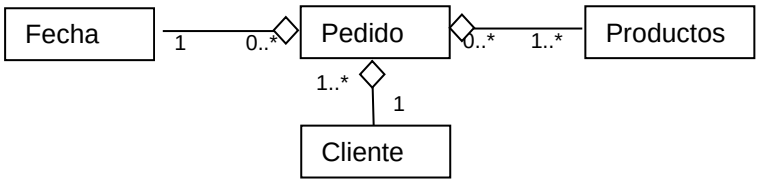
- ¿Se puede usar la frase "parte de" o "tiene un (o muchos)"?
- ¿Existen operaciones en el todo que se apliquen automáticamente a sus partes (propagación)?
- ¿Existen algunos valores de atributos que se propaguen desde el todo a todas o algunas partes?
- ¿Existe una subordinación de una clase hacia otra?

La decisión de usar agregación es una cuestión de juicio y normalmente arbitraria. En ocasiones no es obvio si una asociación debe modelarse como una agregación. Sin embargo, en la práctica generalmente no existen problemas en la forma en que se represente.

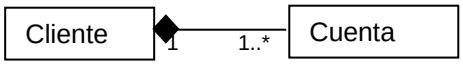
La herencia múltiple se confunde a menudo con la agregación. Cuando se considere la herencia contra la agregación, se debe aplicar la prueba correspondiente. Si no se puede afirmar que existe una relación "es un", se debe emplear agregación o alguna otra relación en vez de herencia. La agregación no es lo mismo que la herencia. La agregación relaciona dos objetos y uno es parte del otro. La generalización relaciona clases y es

una forma de estructurar la descripción de un solo objeto. Tanto las subclases como las superclases se refieren a propiedades de un mismo objeto.

Un ejemplo de agregación es el caso de un pedido de un cliente, el registro del pedido debe contener una fecha (que es un objeto), un cliente y uno o varios productos. Todos estos datos son objetos por sí mismos, ya que tienen atributos y comportamientos particulares.



Una **composición** es un tipo especial de agregación donde cada componente dentro de una composición puede pertenecer sólo a un todo. Cliente-Cuenta.



3.4.1. Tipos de agregaciones

La agregación puede ser:

- **Fija**. Aquella que tiene una estructura fija; el número y tipo de subpartes son predefinidas. Un auto tiene un motor.
- **Variable**. Tiene un número finito de niveles, pero el número de partes puede variar. Las compañías pueden tener un número variable de departamentos.
- **Cíclica o recursiva**. La que contiene directa o indirectamente, una instancia del mismo tipo del todo. Por ejemplo, un bloque de código tiene instrucciones que pueden ser instrucciones simples o compuestas y las instrucciones compuestas son a su vez bloques de código.

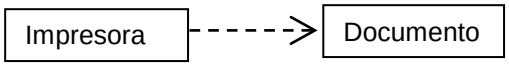
3.5. Uso

En este caso la relación de ***uso*** se da en algún método, donde un objeto de una clase *es parte de la implementación de algún método* (parámetro o variable local) y por tanto puede decirse que la clase definida usa los servicios de la otra clase.

Las relaciones “de uso” entre clases corren paralelas a los enlaces “hermano a hermano” entre las instancias correspondientes de esas clases. Mientras que una asociación denota una conexión semántica bidireccional, una relación “de uso” es un posible refinamiento de una asociación, por el que se establece qué abstracción es el cliente y qué abstracción es el servidor que proporciona ciertos servicios.

Una relación “de uso” cíclica es equivalente a una asociación, aunque la afirmación inversa no es necesariamente cierta.

En las relaciones “de uso” los objetos no son parte de otra clase, sólo es usada por ella. Estas relaciones ocasionalmente son demasiado restringidas porque sólo permiten acceder a la interfaz pública del proveedor. En ocasiones, por razones tácticas, hay que romper el encapsulamiento de estas abstracciones y para ello se emplea el concepto friendly o amiga.



3.6. Metaclases

Un metadato es un dato que describe otro dato. Sabemos que un objeto es una instancia de alguna clase, sin embargo, algunos lenguajes permiten manejar una clase como un objeto. Esta clase de clase es una *metaclase*. Dicho de otra forma, una metaclase *es una clase cuyas instancias son, ellas mismas*. Smalltalk y CLOS soportan este concepto y C++ no. La idea de metaclases lleva a la idea del modelo de objetos a su conclusión natural en los LOO puros.

El propósito principal de una metaclase es proporcionar variables de clase (compartidas por todas las instancias de la clase) y operaciones para inicializar variables de clase y crear la instancia simple de la metaclase.

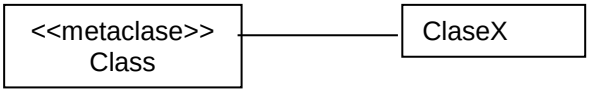
Java proporciona la metaclase Class y la reflexión (Reflection) que permite obtener al momento de ejecución la clase de un objeto, sus atributos y métodos.

Class se encuentra en el paquete java.lang y como todas las clases, hereda de la clase Object. Class no tiene constructores, las instancias de esta clase se crean automáticamente por la máquina virtual cada vez que se carga una clase.

Java permite el acceso a las instancias de la clase Class y la ejecución de los métodos definidos en esa clase. Este acceso permite la Reflexión, capacidad de obtener al momento de ejecución la clase a la que pertenece un objeto, sus atributos y métodos.

Class proporciona el método estático forName, que obtiene el objeto Class de una clase a partir de su nombre.

La representación de metaclases en el diagrama de clases usa estereotipos << >> para indicar la metaclase y las relaciones pueden ser asociaciones o uso.



3.7. Tipos Genéricos (Instanciación)

Este tipo de relación se logra mediante clases paramétrizadas declaradas como plantillas o modelos (template), también llamadas clases genéricas o tipos genéricos, las cuales se pasan como parámetros. Una clase parametrizada es aquella que sirve como modelo para otras clases, debe ser instanciada (sus parámetros deben llenarse) antes de crear los objetos. En este caso las clases instanciadas pueden ser diferentes y no estar unidas por ninguna superclase común, aunque ambas se derivan de la misma clase parametrizada.

Estas instanciaciones son seguras respecto a tipos. Las reglas de tipos rechazarán tipos impropios.

Las relaciones de instanciación casi siempre requieren alguna relación “de uso”, que hace visible a las clases actuales usadas para rellenar el modelo o plantilla.

La parametrización de tipos permite parametrizar las funciones aritméticas respecto al tipo numérico básico, de forma que los programadores puedan obtener un modo uniforme de tratar con enteros, flotantes o dobles.

Desde una perspectiva de diseño, las clases parametrizadas también son útiles para capturar ciertas decisiones de diseño sobre el protocolo de una clase. Mientras que una definición de clase exporta las operaciones que pueden realizarse sobre instancias de esa clase, los argumentos de un modelo sirven para importar clases (y valores) que suministran un protocolo específico. En C++ y Java, esta congruencia de tipos se realiza en tiempo de compilación, cuando se expande la instanciación. Por ejemplo, se podría declarar una clase fila ordenada que representase colecciones de objetos ordenados según algún criterio. Esta clase parametrizada debe contar con alguna clase Elemento, pero debe esperar también que Elemento proporcione alguna operación de ordenación. Parametrizando la clase de esta forma, se logra esto de forma más débilmente acoplada: se puede

sustituir el argumento formal Elemento con cualquier clase que ofrezca esta función de ordenación. En este sentido, se puede definir una clase parametrizada como una que denota una familia de clases cuya estructura y comportamiento están definidos independientemente de los parámetros formales de la clase.

Los ***genéricos*** son un convenio sintáctico que permite utilizar una plantilla de una clase que se parece al polimorfismo. Cuando se particulariza, se genera una nueva clase. La potencia de los genéricos es que permiten definir funcionalidad para distintos tipos de valores.

Por ejemplo, se puede definir una clase genérica llamada Envoltorio que permite guardar y mostrar un solo valor. El parámetro formal del tipo genérico Tipo es un sustituto del tipo de valor que se desea representar.

```
public class Envoltorio<Tipo> {

        private Tipo valor;

        public Envoltorio (Tipo parametro) {
                valor = parametro;
        }

        public Tipo get () {
                return valor;
        }

        public void set(Tipo parametro) {
                valor = parametro;
        }
}
```

En la definición de una clase genérica, los parámetros formales se escriben entre ángulos tras el nombre de la clase. Estos parámetros formales se pueden utilizar en la definición de la clase como sustitutos del tipo real que se desee posteriormente.

Para generar una clase a partir de una plantilla de clase, se proporcionan los parámetros actuales en sustitución de los parámetros formales. Los parámetros actuales se indican tras el nombre de la clase entre ángulos, donde el parámetro actual de tipo es un tipo no primitivo. Por ejemplo:

```
MiFecha oFecha = new MiFecha (4, 3, 2020);

Envoltorio<MiFecha> x1 = new Envoltorio<MiFecha>(fecha);

Envoltorio<String> x2 = new Envoltorio<String>(“nuevo texto”);
```

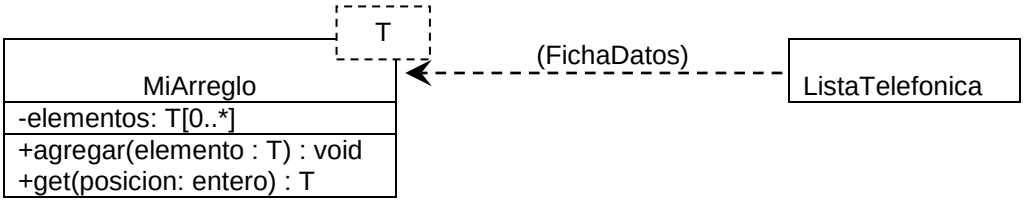
La variable x1 hace referencia a un objeto Envoltorio<MiFecha> que tiene una variable de instancia valor del tipo MiFecha, un método de acceso get() que devuelve un objeto MiFecha y un método modificador set() que espera recibir un objeto MiFecha como parámetro. La variable x2 hace referencia a un objeto Envoltorio>String> que tiene una variable de instancia valor del tipo String, un método de acceso get () que retorna un String y método de modificación set () que espera recibir un String como parámetro. Tanto x1 como x2 hacen referencia a objetos que tienen un método toString() que retorna un String. Por lo tanto, lo siguiente es correcto:

```
String sCadena = x2.get ();
x2.set (“otro texto”);
x1.set (new MiFecha (10, 3, 2020);
MiFecha oFecha = x1.get ();
System.out.println(x1 + “ “ + x2);
```

y las siguientes sentencias son ilegales:

```
MiFecha oFecha2 = x2.get ();
```

```
x2.set (new MiFecha (1,1, 2020);
x1.set ("fecha asignada");
String sCadena = x1.get ();
```



3.8. Herencia

Uno de los objetivos principales de la POO es la reutilización y la herencia es el mecanismo más empleado para lograrlo. Reutilizar código no significa copiar y pegar sobre el código que se escribe, sino usar componentes especialmente diseñados para ello. La reutilización reduce costos y tiempos de los proyectos.

La herencia es una relación entre clases en la que una clase comparte la estructura y/o comportamiento definidos en una o más clases. La herencia o generalización es la relación entre una clase y una o más versiones refinadas de ella. La clase de la que otras heredan se denomina ***superclase*** y la clase que hereda de otra o más clases se denomina ***subclase***. La herencia define una jerarquía de tipos entre clases, en la que una subclase hereda de una o más superclases. La capacidad de un lenguaje para soportar o no este tipo de herencia distingue a los lenguajes de programación orientados a objetos de los lenguajes basados en objetos.

Los términos herencia, generalización y especialización se refieren a aspectos de la misma idea y es común usarlos como sinónimos. ***Generalización*** se usa para referirse a la relación entre clases, mientras que herencia se refiere al mecanismo de compartir atributos y métodos usando la relación de generalización. La generalización y especialización son dos puntos de vista diferentes de la misma relación, vista desde la superclase o desde las subclases. La palabra generalización se deriva del hecho que la superclase generaliza a las subclases. La ***especialización*** se refiere al hecho de que las subclases refinan o especializan a la superclase. La herencia también se dice que es una relación "es un", ya que cada instancia de una subclase también "es una" instancia de la superclase.

En los lenguajes de programación, herencia significa que el comportamiento y datos asociados con las clases hijas son siempre una *extensión* (esto es, un conjunto más grande) de las propiedades asociadas con las clases padre. Una clase hija obtendrá todas las propiedades de la clase padre, y puede adicionalmente definir nuevas propiedades. Por otro lado, ya que la clase hija es más especializada (o restrictiva) que la clase padre, también es, en cierto sentido, una contracción del tipo padre. Esta tensión entre herencia como expansión y herencia como contracción es una fuente de mucho del poder inherente a la técnica, pero al mismo tiempo es causa de mucha confusión en su empleo.

La herencia es siempre ***transitiva***, de modo que una clase puede heredar propiedades de muchas superclases varios niveles hacia arriba. Por ejemplo, la clase Perro es una subclase de la clase Mamífero, y la clase Mamífero es una subclase de la clase Animal, por tanto, Perro heredará atributos de Mamífero y de Animal.

Un factor de complicación en una descripción intuitiva de la herencia es el hecho de que las subclases puedan sobrescribir el comportamiento heredado de las clases padre, para ello, definen operaciones con el mismo nombre. La operación sobrescrita refina y remplaza la operación sobrescrita, sin embargo, la firma es la misma.

Existe cierta tensión entre la herencia y el encapsulamiento. En un alto grado, el uso de la herencia expone algunos de los secretos de una clase heredada. En la práctica, esto implica que, para comprender el significado de una clase particular, muchas veces hay que estudiar todas sus superclases, en ocasiones incluyendo sus vistas internas.

La mayoría de los LOO permiten a la implantación de un método de una subclase invocar directamente un método definido por alguna superclase. También es común que la implantación de un método redefinido invoque el método del mismo nombre definido por una clase padre. Por ejemplo, en Java y Smalltalk se puede invocar un método que provenga de la clase inmediatamente superior usando la palabra reservada ***super***.

En la práctica, un método redefinido suele invocar un método de una superclase ya sea antes o después de llevar a cabo alguna otra acción. De este modo, los métodos de la subclase desempeñan el papel de aumentar el comportamiento definido en la superclase.

****

****

EXAMEN PARCIAL

3.8.1. La clase base Object

La clase más generalizada en una estructura de clases se llama *clase base*. La mayoría de los LOO tienen muchas clases bases, que representan las categorías más generalizadas de abstracciones en el dominio que se trata. Otros lenguajes requieren una clase base en la raíz, que sirve como la clase base de todas las clases, tal es el caso de Java y Smalltalk cuya clase raíz es Object.

En Java todas las clases usan herencia. A menos que se especifique lo contrario, todas las clases se derivan de la clase raíz llamada ***Object***. Si no se especifica una clase padre explícitamente, la clase Object se asume implícitamente. Así, la declaración de la clase HolaMundo es la misma que se muestra a continuación:

```
public class HolaMundo extends Object {…}
```

La clase Object proporciona la funcionalidad mínima que es común a todos los objetos. Esta incluye los siguientes métodos:

- **equals** (Object obj). Determina si el argumento objeto es el mismo que el receptor. Este método generalmente se sobrescribe para cambiar la prueba de igualdad para diferentes clases.
- **getClass** (). Retorna la clase del receptor, un objeto de tipo Class.
- **hashCode** (). Retorna un valor hash para cada objeto. Este método debe sobrescribirse cuando se modifica el método equals.
- **toString** (). Convierte el objeto en una cadena. Este método también se sobrescribe generalmente.

La sintaxis en Java para especificar la herencia es:

```
<declaración_de_clase> ::= <modificador> class <nombreClase> extends <nombreSuperclase> {
        <declaración_de_atributos>*
        <declaración_de_constructor>*
        <declaración_de_método>*
}
```

3.8.2. El principio de Sustitución

Aunque la herencia por sí misma es importante, su verdadera potencia se obtiene cuando se uso conjuntamente con el enlazado dinámico (dinamic binding) y el encapsulamiento.

El concepto de sustitución es fundamental para muchas de las técnicas más poderosas de desarrollo de software en la POO. La idea de ***sustitución*** es que el *tipo dado en una declaración de una variable no tiene que coincidir con el tipo asociado con el valor de la variable almacenada*. Nótese que esto no puede ocurrir en

los lenguajes de programación convencionales, pero es común en programas orientados a objetos y se puede implementar gracias a los mecanismos de herencia.

Ya que Object es la clase padre de todos los objetos, una variable declarada usando este tipo puede contener cualquier valor no primitivo.

Cuando una clase se construye usando herencia de una clase existente, el argumento usado para justificar la validez de la sustitución es:

- Las instancias de la subclase poseen todos los atributos asociados a la clase padre.
- Las instancias de la subclase deben implementar, mediante la herencia al menos (si no se sobrescriben explícitamente) toda la funcionalidad definida para la clase padre. (Pueden también definir nueva funcionalidad, pero no es importante para este argumento).
- Así, una instancia de la clase hija puede realizar el comportamiento de la clase padre y debería ser indistinguible de una instancia de la clase padre si se substituye en una situación similar.

Dependiendo de la forma de herencia, no todas las clases son candidatas a la sustitución.

El término ***subtipo*** se usa para describir la relación entre tipos que explícitamente reconocen el principio de substitución. Esto es, el tipo B se considera como un subtipo de A si se cumplen las siguientes condiciones: Una instancia de B puede asignarse legalmente a una variable declarada como de tipo A; y este valor puede ser usado por la variable sin observar cambios de comportamiento.

El término ***subclase*** se refiere solamente al mecanismo de construcción de una nueva clase usando herencia. La relación subtipo es más abstracta. Los subtipos pueden formarse usando interfaces, uniendo tipos que no tienen relación de herencia.

La asignación de un objeto x a un objeto c es posible si el tipo de x es el mismo que el tipo de y o un subtipo de él. Este tipo de asignación es peligrosa ya que cualquier estado adicional definido por una instancia de la subclase se ve recortado en la asignación a una instancia de la superclase.

Los lenguajes con comprobación estricta de tipos permiten la conversión (explícita) del valor de un objeto de un tipo a otro, pero normalmente sólo si hay alguna relación superclase/subclase entre los dos.

El ***enlace dinámico*** consiste en que no se enlaza la llamada hasta el momento de la ejecución; en ese momento se puede despachar ésta al código que sea adecuado. El runtime system que controla la ejecución del programa “une” en ese momento el método correcto a la llamada.

3.8.3. Formas de herencia

La herencia se emplea en diversas formas. A continuación, se muestran las más comunes sin intentar ser exhaustiva; incluso algunas veces dos o más descripciones se aplican a una misma situación, ya que algunos métodos en una clase usan herencia de un modo y otros métodos en otro.

3.8.3.1. Herencia por Especialización, Extensión u Optimización

Probablemente la herencia y subclasificación más común es para especialización. En esta forma, la clase nueva es una variedad de especialización de la clase padre, pero satisface las especificaciones de la clase padre en todos los aspectos relevantes. Así, esta forma siempre crea un subtipo, y el principio de sustitución es válido. Junto con la siguiente categoría (especificación) esta es la forma ideal de herencia, y algunas veces el mejor diseño.

La subclasificación por extensión *ocurre cuando una clase hija sólo agrega nuevo comportamiento a la clase padre (generalmente afectando nuevos atributos de la subclase) y no modifica o altera ninguno de los comportamientos heredados*.

Como la funcionalidad del padre permanece disponible e intocable, la subclasificación por extensión no contraviene el principio de sustitución, y por tanto las subclases son siempre subtipos.

La creación de ventanas en Java usando herencia de la clase JFrame es un ejemplo de subclasificación por extensión. En este caso, la nueva clase agrega uno o varios métodos, extendiendo el comportamiento y haciéndolo específico de la aplicación.

Un ejemplo es La clase Auto es una especialización de la clase Vehículo. El Auto hereda atributos y métodos de la clase Vehículo, Auto es Subclase de Vehículo, Vehículo es Superclase de Auto.

3.8.3.2. Herencia por Especificación (clases abstractas)

Otro uso frecuente de la herencia es garantizar que las clases mantengan una cierta interfaz común –esto es, implementan métodos que tienen el mismo encabezado. La clase padre puede ser una combinación de operaciones implementadas y operaciones que son dejadas a las clases hijas para su implementación. Generalmente, no existe intercambio entre la clase padre e hija, la hija implementa los métodos descritos, pero no implementados en la clase padre.

Este *es un caso especial de subclasificación por especialización, excepto que las subclases no son refinamientos de tipos existentes, sino realizaciones de una especificación abstracta incompleta*. Así, la clase padre define la operación, pero no la implementa. Sólo la clase hija proporciona una implementación. En tales casos la clase padre se conoce como una *especificación de clase abstracta*; las clases especializadas son llamadas *clases concretas o clases hoja*.

Existen dos mecanismos diferentes proporcionados por Java para soportar la especificación. La más obvia es el uso de interfaces, donde una interfaz se usa para describir sólo los requerimientos necesarios, y no se hereda comportamiento a las subclases, quienes los implementan.

El otro mecanismo es la herencia de clases formadas usando extensión y la forma de garantizar que las subclases implementarán las definiciones es usar la palabra reservada *abstract*. Una clase declarada como abstracta, debe subclasificarse; no es posible crear una instancia de esta clase, sólo las clases concretas pueden instanciarse.

Una ***clase abstracta*** es una clase que no se puede instanciar y que organiza características comunes a varias clases. Puede aparecer naturalmente en la aplicación o ser introducida artificialmente para promover la reutilización de código y compartir atributos y métodos.

Cuando una superclase se divide en subclases y las instancias pertenecen siempre a alguna de las subclases, entonces la superclase se considera abstracta, ésta puede definir métodos para ser usados por la subclase, o definir el protocolo de la operación, sin proporcionar la implementación (método abstracto) y cada clase concreta debe proporcionar su propia implementación y no debe tener métodos abstractos.

Los métodos individuales pueden también ser declarados como abstractos, y por tanto deberán sobrescribirse antes de instanciar objetos. Los métodos abstractos sólo tienen firma, pero no implementación. Cualquier clase con un método abstracto, automáticamente se vuelve una clase abstracta, la cual no puede instanciarse. Si algún método es abstracto, es obligatorio que la clase se defina como abstracta. Lo opuesto no es obligatorio.

En general, la subclasificación por especificación puede reconocerse cuando la clase padre no implementa el comportamiento, sino que sólo proporciona los encabezados de los métodos que deben implementarse en las clases hijas.

Una clase concreta puede tener subclases abstractas, las cuales deberán tener descendientes concretas.

La sintaxis en Java para crear clases abstractas es:

<declaración_clase_abstracta>::=<modificador_acceso> **abstract class** <identificador> **extends** <Superclase>
{
<declaración_de_atributos>*
<declaración_de_constructor>*
<declaración_de_método>* | <declaración_de_métodos_abstractos>*
}

La declaración de métodos abstractos es:

<declaración_método_abstracto> ::= **public abstract** <tipo> <identificador> ([<parámetros>*]);

Nótese que, en la declaración de métodos abstractos, siempre deben ser públicos, ya que se está definiendo una interfaz, siempre llevan la palabra reservada abstract y nunca tienen cuerpo o instrucciones.

Ejemplo:

```
public abstract class FormaGrafica {
    …
    public abstract void desplegar (int x, int y);
    …
}
```

En este caso, la clase FormaGrafica desconoce cómo desplegar la forma ya que puede ser de cualquier tipo, por lo que el método se deja para ser implementado por alguna subclase concreta de un tipo específico como una Ventana.

3.8.3.3. Herencia por Construcción o Conveniencia

Una clase puede heredar casi toda la funcionalidad deseada de una clase padre, tal vez cambiando sólo los nombres de los métodos usados o modificando los argumentos. Esto puede ser cierto aún si la nueva clase y la clase padre comparten una relación como conceptos abstractos.

La subclasificación por construcción *ocurre cuando no existe una relación lógica entre los conceptos de dos clases, pero desde un punto de vista práctico, mucho del comportamiento necesario para la abstracción de una clase corresponde con el comportamiento de otra*. Así, usar la herencia en esta situación reduce la cantidad de trabajo necesario para desarrollar la otra clase.

La herencia por construcción algunas veces se evita, ya que rompe el principio de sustitución (ni se pretende) directamente (formando subclases que no son subtipos). Por otro lado, ya que es una forma fácil y rápida de desarrollar nuevas abstracciones, es ampliamente usado. Este tipo de herencia puede traer problemas de mantenimiento, por lo que es mejor generalizar los aspectos comunes de ambas clases en una tercera clase, de la cual hereden.

Por ejemplo, una pared y un edificio tienen en común que se construyen, pudiendo aplicar herencia de edificio a pared, sin embargo, una pared no es un edificio, aunque se usa pared para construir un edificio. Otro ejemplo es un punto y una línea, ambas se dibujan, pero un punto no es una línea y el punto se usa para crear líneas.

3.8.3.4. Herencia por Limitación o Restricción

La subclasificación por limitación *ocurre cuando el comportamiento de la subclase es menor o más restrictivo que el comportamiento de la clase padre*. Como en la subclasificación por extensión, la subclasificación por limitación ocurre frecuentemente cuando un programador construye en base a una clase existente que no debe o no puede ser modificada.

En teoría una alternativa para implementarla podría ser, tener métodos no deseados como excepciones. Sin embargo, Java no permite que las subclases sobrescriban un método e introduzcan nuevas excepciones que no

se declaran en la clase padre. De modo que las restricciones podrían hacerse sobrescribiendo un método que no realice las operaciones restringidas.

En general, la implementación de las restricciones se realiza mediante algoritmos que verifican que se cumplan ciertas normas y de no ser así rechazan las operaciones.

La subclasificación por limitación se caracteriza por la presencia de técnicas que toman un método permitido previamente y lo hacen ilegal. Ya que la subclasificación por limitación va en contra del principio de sustitución directamente, y ya que construye clases que *no son subtipos*, es mejor evitarla.

Por ejemplo, un Cuadrado puede ser una clase que restrinja los valores de los atributos de la clase Rectángulo, ya que el ancho y largo deben ser iguales.

3.8.3.5. Herencia por Combinación o Herencia Múltiple (Interfaces)

Cuando se discuten conceptos abstractos, es común formar nuevas abstracciones combinando propiedades de dos o más abstracciones. *La habilidad de una clase de heredar de más de una clase padre* se conoce como herencia múltiple.

Java no permite herencia múltiple. Sin embargo, permite que una nueva clase extienda una clase existente e implemente una interfaz. También es posible implementar más de una interfaz, y esto se ve como una combinación de dos categorías.

El término interfaz es conocido también como "herencia de especificación", en contraste con la "herencia de código" proporcionada por la subclasificación. En la subclasificación se comparte una estructura o código, mientras que una interfaz es una técnica más apropiada cuando dos conceptos comparten la especificación del comportamiento, pero no el comportamiento actual.

Las ***interfaces*** no son clases, aunque son similares a las clases abstractas excepto que se utiliza la palabra *interface* en lugar de abstract y class y *son plantillas parciales de lo que debe aparecer en una clase*. La definición de un interfaz difiere de la de una clase abstracta en tres puntos:

- Una interfaz no puede especificar la implementación de ningún método.
- Todos los métodos de una interfaz son públicos.
- Todos los atributos definidos en una interfaz son public static final, ya sea explícita o implícitamente.

La sintaxis en Java para declarar una interfaz es:

```
<declaración_de_interfaz> ::= <modificador_acceso> interface <identificador> [<superInterfaz>] {
        <declaración_de_constante>*
        <declaración_de_método_abstracto>*
}
```

Ejemplo:

```
public interface FormaGrafica {
        …
        public void desplegar (int x, int y);
}
```

Los métodos definidos en una interfaz son implícitamente abstractos y no pueden contener implementación. Todos los atributos declarados dentro de una interfaz deben ser *static final* (constantes estáticas).

Ya que las interfaces no son clases, no se pueden heredar, sólo se deben implementar, por lo que cuando se utilizan se usa la palabra ***implements***:

```
public class Linea implements FormaGrafica {
        …
        public void desplegar (int x, int y) {
                //implementación
        }
}
```

En Java sólo se puede heredar (extends) de una clase, sin embargo, se pueden implementar (implements) múltiples interfaces dentro de una clase.

De modo similar a las clases, las interfaces se pueden extender por jerarquías en subinterfaces. Una subinterfaz hereda los métodos abstractos y constantes estáticas de la superinterfaz, y puede definir nuevos métodos abstractos y constantes estáticas. Una interfaz es capaz de extender más de una interfaz a la vez.

Por ejemplo, la clase JFrame, de la cual heredan la mayoría de las ventanas Java, proporciona una gran variedad de código en forma de métodos que se heredan y usan sin modificaciones. Así, la herencia es el mejor mecanismo para usar en esta situación:

```
public class GUI extends JFrame {
        …
}
```

Por otro lado, las características necesarias para un ActionListener, que es un tipo de objeto que responde a la selección de un botón puede ser descrito por un solo método, y la implementación de ese método no puede ser preestablecida, ya que difiere de una aplicación a otra. Así, se usa *interfaz* para describir sólo los requerimientos necesarios y no se hereda ningún comportamiento a la subclase que implementa este comportamiento:

```
public class GUI extends JFrame {
        …
        private void actionPerformed (ActionEvent e) {
                …
        }
}
```

En general, la relación clase-subclase debe usarse cuando una subclase herede código, atributos o comportamiento de la clase padre. El mecanismo de interfaces debe usarse cuando la clase hija herede sólo la especificación del comportamiento esperado, pero no el código actual.

La sustitución puede ocurrir también en interfaces. Un ejemplo son las instancias escuchas de eventos. La clase en la que se declara un valor Escucha implementa la interfaz ActionListener. Ya que implementa esa interfaz, se puede usar el Escucha como parámetro a un método (en este caso addActionListener) que espera un valor ActionListener.

```
public class Prueba extends JFrame {
        …
        private class EscuchaBoton implements ActionListener {
                public void actionPerformed (ActionEvent e) {
                        …
                }
        }

        public Prueba () {
                …
                boton.addActionListener(new EscuchaBoton ());
        }
}
```

A partir de Java 8, las interfaces pueden contener métodos con implementación por omisión. Estos métodos pueden sobrescribirse en las clases que implementan la interfaz o usar la versión original sin modificaciones.

Si existe más de una interfaz con métodos con el mismo nombre, el método debe ser sobrescrito.

```
public interface Interfaz{
        default void metodo(){
                System.out.println(“Omisión”);
        }
}
```

También, a partir de Java 8, las interfaces pueden contener métodos **estáticos** con implementación por omisión. Estos métodos NO pueden sobrescribirse en las clases que implementan la interfaz. Esto ayuda a evitar implementaciones no deseadas y deben usarse con el nombre de la interfaz.

```
public interface Interfaz{
        static void metodo(){
                System.out.println(“Omisión”);
        }
}
```

Las modificaciones a las interfaces en Java 8, hacen semejantes a las clases abstractas e interfaces. La diferencia es que las interfaces no heredan los atributos, a diferencia de las clases abstractas.

Su objetivo es simular la herencia múltiple y disminuir la duplicidad de código. También evita modificar varias clases que implementen una interfaz después de agregar nuevos métodos a la interfaz.

Si una clase tiene un método privado con el mismo nombre de un método agregado, el compilador marcará error.

3.8.4. Modificadores y la herencia

Mientras que los modificadores de acceso private y public no se ven afectados por la herencia, el modificador de acceso ***protected*** está pensado para evitar romper del todo el encapsulamiento al momento de crear una jerarquía de clases, ya que un atributo o método declarado como protegido (protected) sólo puede ser usado por la clase misma y las subclases que la hereden. Para tener acceso a estos miembros protegidos pueden llamarse por su nombre directamente <miembro>, siempre que no exista ambigüedad; o se puede usar super.<miembro> en caso de ambigüedad.

Por otro lado, el modificador ***final***, es lo opuesto a abstract. Cuando se aplica a una clase, indica que la clase no puede subclasificarse. Similarmente, aplicada a un método, indica que el método no puede sobrescribirse. Por lo tanto, se garantiza que el comportamiento de la clase siempre será el definido en la clase original y no se modificará en las subclases.

Otro aspecto importante es que cualquier atributo declarado dentro de una interfaz, es automáticamente público y final, ya que no se puede modificar, pero se debe poder tener acceso a él desde cualquier clase que la implemente.

Los compiladores optimizados pueden hacer uso del hecho de que los atributos, métodos o clases se han declarado como finales generando mejor código.

3.8.5. Reglas para crear herencia

Las siguientes reglas ayudan a hacer el software fácil de entender, fácil de mejorar y menos propenso a errores:

- Las operaciones *selectoras*, que acceden a los datos, pero no los modifican, deben *heredarse* a todas las subclases.
- Las operaciones *modificadoras*, que modifican los valores de los atributos, se *heredan* por *extensión*.
- Las operaciones *modificadoras* que cambian atributos o asociaciones restringidas se deben bloquear con una herencia por *restricción*.
- Las operaciones *no* deben *sobrescribirse* para hacerlas diferentes de su operación heredada, en su manifestación externa visible. Todos los métodos que implementan una operación deben tener el *mismo protocolo*.
- Las operaciones heredadas pueden tener un *comportamiento adicional* refinado.

La implementación y uso de muchos LOO existentes violan estos principios.

3.8.6. Beneficios de la herencia

- **Reutilización de código**. Cuando el comportamiento se hereda de otra clase, el código que proporciona ese comportamiento no tiene que reescribirse. Esto puede ser obvio, pero las implicaciones son importantes. Muchos programadores gastan mucho tiempo reescribiendo código que ya han escrito muchas veces. Con las técnicas orientadas a objetos, estas funciones pueden escribirse una vez y reutilizarse muchas veces.
- **Incremento de confianza**. El código que se ejecuta frecuentemente tiende a tener menos errores que el código que se ejecuta pocas veces. Cuando los mismos componentes se usan en dos o más aplicaciones, el código será ejercitado más que el código que se desarrolla para una sola aplicación. De este modo, los errores en ese código tienden a descubrirse más rápido y las siguientes aplicaciones se beneficiarán del uso de componentes que tienen menos errores. Similarmente, el costo de mantenimiento de componentes compartidos puede dividirse entre muchos proyectos.
- **Compartir código**. El código compartido se puede dar en varios niveles con las técnicas orientadas a objetos. En un nivel, muchos usuarios o proyectos pueden usar las mismas clases. Otra forma de compartir ocurre cuando dos o más clases desarrolladas por el mismo programador como parte de un proyecto hereda de una misma clase padre. Cuando esto ocurre, dos o más tipos de objetos compartirán el código que heredaron. Este código necesita escribirse sólo una vez y contribuirá en tamaño sólo una vez en el resultado del programa.
- **Consistencia de interfaz**. Cuando dos o más clases heredan de la misma superclase, se está asegurando que el comportamiento que heredan será el mismo en todos los casos. Así, es más fácil garantizar que las interfaces de objetos similares son similares y que los usuarios no se confundirán con objetos que son casi iguales pero su interacción es muy diferente.
- **Componentes de software**. La herencia permite que los programadores construyan componentes de software reutilizables. El objetivo es permitir el desarrollo de aplicaciones nuevas sin requerir mucho código. La mayoría de los lenguajes proporcionan una biblioteca amplia de componentes de software para usar en el desarrollo de software.
- **Desarrollo rápido de prototipos**. Cuando un sistema de software se construye a partir de componentes reutilizables, los desarrolladores se pueden concentrar en comprender las porciones nuevas e inusuales del sistema. De modo que los sistemas pueden generarse más rápido y fácil; desarrollando un prototipo con el que se experimenta y se puede ir mejorando. Esto es particularmente importante cuando los requerimientos del sistema son vagos al principio del proyecto.
- **Polimorfismo**. El software construido convencionalmente generalmente se escribe de abajo hacia arriba, y puede diseñarse de arriba abajo. Esto es, las rutinas de bajo nivel se escriben y se van generando nuevas abstracciones sobre ellas; siendo como el proceso de construcción de una pared, donde un bloque de arriba debe colocarse sobre otro bloque que este abajo. Normalmente, la portabilidad del código decrece mientras se llega a los niveles superiores de abstracción. El polimorfismo permite que los programadores generen componentes altamente reutilizables que pueden servir para diferentes aplicaciones cambiando partes de los niveles inferiores.
- **Ocultamiento de información**. Un programador que reutiliza un componente de software sólo necesita comprender la naturaleza del componente y su interfaz. No es necesario conocer el detalle de las técnicas de implementación, reduciéndose el acoplamiento entre componentes.

3.8.7. Costos de la herencia

Los beneficios de la herencia en la POO son maravillosos, sin embargo, casi nada viene sin algún costo, he aquí una lista de ellos:

- **Rapidez de ejecución**. Es casi imposible que las herramientas de software general sean tan rápidas como las creadas de modo particular. La herencia de métodos, que debe lidiar con subclases arbitrarias, es generalmente más lenta que el código especializado.
- **Tamaño del programa**. El uso de bibliotecas de software frecuentemente impone un aumento de tamaño que la construcción de software especializado no lo hace.
- **Sobrecarga en el paso de mensajes**. El paso de mensajes es por naturaleza una operación más costosa que la simple invocación de procedimientos.
- **Complejidad de programa.** Aunque la POO es considerada una solución a la complejidad de software, el abuso de herencia puede simplemente remplazar una forma de complejidad con otra.

3.8.8. Tipos de herencia

La herencia se puede dar desde un solo padre o de más de uno a la vez, dependiendo de esto existen dos tipos de herencia.

3.8.8.1. Herencia Simple

En la ***herencia simple***, la relación entre las clases es de un padre a una o varias clases hijas, de modo que una subclase no puede tener más que padre.

3.8.8.2. Herencia Múltiple

La herencia simple es útil, pero obliga al programador a derivar de una sola clase de entre dos igualmente útiles. Esto limita la aplicabilidad de las clases predefinidas, haciendo necesario en ocasiones duplicar código. Por ejemplo, no se puede derivar un gráfico que es al mismo tiempo círculo e imagen; hay que derivar de uno o del otro y reimplementar la funcionalidad de la clase excluida. La herencia simple no es suficiente para expresar algunos tipos de relaciones complejas.

La ***herencia múltiple*** permite a una clase tener más de una superclase y heredar características de todos los padres. Esto permite mezclar información desde varias fuentes. Esta es una forma más complicada de generalización que la herencia simple. La ventaja de la herencia múltiple es el gran poder de especificar clases e incrementar la oportunidad de reutilización. Esto trae el modelo de objetos más cercano a la forma en que piensa la gente. La desventaja es la pérdida de una concepción e implementación simple.

El diseño de una estructura de clases adecuada que implique herencia es difícil, especialmente si se trata de herencia múltiple. Frecuentemente es un proceso incremental e iterativo y pueden aparecer dos problemas:

- Tratar las **colisiones de nombres** de diferentes superclases.
- Manejar **herencia repetida**.

Las ***colisiones de nombres*** *son el principal problema de la herencia múltiple y pueden aparecer cuando dos o más superclases diferentes emplean el mismo nombre para algún elemento de sus interfaces, como los atributos o los métodos*. Puesto que la subclase hereda de ambas clases, heredaría dos operaciones con el mismo nombre. Las colisiones pueden introducir ambigüedad en el comportamiento de la subclase que hereda de forma múltiple.

Existen tres tipos de soluciones básicas a este problema:

- Primero, la semántica del lenguaje podría contemplar ese choque como algo incorrecto y rehusar la compilación de la clase. Éste es el enfoque adoptado por Smalltalk y Eiffel; sin embargo, Eiffel permite renombrar elementos de forma que no haya ambigüedad.
- Segundo, la semántica del lenguaje podría contemplar el mismo nombre introducido por varias clases como referente al mismo atributo, que es el enfoque adoptado por CLOS.
- Tercero, la semántica del lenguaje podría permitir el desacuerdo, pero requerir que todas las referencias al nombre califiquen de forma completa la fuente de su declaración. Este es el enfoque de C++.

El segundo problema es la ***herencia repetida****; esta sucede cuando una clase es antecesora de otra por más de una vía*. Si se permite herencia múltiple en un lenguaje, en algún momento se escribirá una clase D con dos padres B y C, donde cada uno tiene como padre a una clase A; o alguna otra situación donde D herede dos (o más veces) de A. Esta situación se llama ***herencia repetida*** y debe tratarse de forma correcta.

Existen tres enfoques para tratar este problema:

- Se puede tratar la presencia de herencia repetida como incorrecta. Es el enfoque de Smalltalk y Eiffel.
- Se puede permitir la duplicación de superclases, pero requerir el uso de nombres plenamente calificados para referirse a los miembros de una copia específica. Es el enfoque de C++.
- Se pueden tratar las referencias múltiples a la misma clase como denotadas a la misma clase. Éste es el enfoque de C++ cuando la superclase repetida se introduce como una clase base virtual. También en CLOS, las clases repetidas son compartidas, usando un mecanismo llamado de *lista de precedencia de clases,* que es una lista donde cada vez que se introducen una nueva clase, se incluye la propia clase y todas sus superclases, sin duplicación.

La existencia de herencia múltiple plantea un estilo de clases, llamadas *aditivas*, que son pequeñas clases combinadas para construir clases con un comportamiento más sofisticado. Su función es simplemente añadir funciones a otras clases y nunca se crean objetos de esas clases, ya que no pueden existir por sí mismas. Una clase que se construye principalmente heredando de las clases aditivas y no agrega su propia estructura o comportamiento se llama ***clase agregada****.*

En general, como muchos LOO, como Java, no soportan la herencia múltiple, es necesario en estos casos implementar la herencia múltiple mediante herencia sencilla y posiblemente agregación. A continuación, se describen tres enfoques generales:

- **Implementación de herencia múltiple usando agregación**. Una superclase con múltiples generalizaciones individuales se redefine como un agregado, en el cual cada uno de sus componentes reemplaza una de las ramas de la generalización. Se reemplazan las posibles instancias de la herencia múltiple por un grupo de instancias que componen el agregado. La herencia de las operaciones a través del agregado no es automática, deben delegarse a los componentes apropiados. Si una subclase tiene varias superclases, todas de igual importancia, es mejor usar agregación y preservar la simetría.
- **Implementación de la herencia múltiple heredando de la clase más importante y delegando el resto**. Se toma una como subclase de la superclase más importante y se combina con un agregado correspondiendo a las generalizaciones restantes. Si se tiene una superclase principal, se implementa la herencia múltiple a través de herencia sencilla y agregación. Si el número de combinaciones es pequeño, se puede usar la generalización anidada. Si el número de combinaciones, o el tamaño del código, es grande, se debe evitar este tipo de implementación.
- **Implementación de herencia múltiple usando generalización anidada**. Se crean varios niveles de generalización, se termina la jerarquía con subclases para todas las posibles combinaciones de clases unidas. En este caso no se utiliza agregación. Se preserva la herencia, pero se duplican las declaraciones, rompiendo con el espíritu de la orientación a objetos. Primero se factoriza según el criterio de herencia más importante, y luego el resto. Si una superclase tiene más características que las otras superclases, o si es el cuello de botella en el rendimiento, se debe preservar la herencia en relación a esa clase.

La herencia múltiple es un tema de debate en los LOO, algunos a favor y otros en contra de ella.

3.8.9. Implicaciones de la herencia

La relación entre la herencia y otras características de los LOO puede resumirse:

- Con el fin de hacer más eficiente el uso de las técnicas OO, el lenguaje debe soportar las variables polimórficas. Una variable polimórfica es una variable que se declara de un tipo, pero puede contener un valor de ese tipo o un valor derivado de un subtipo del tipo declarado.
- Ya que la compilación no puede determinar la cantidad de memoria que se requiere para alojar el valor de una variable polimórfica, todos los objetos deben residir en el montículo de la memoria y no en la pila.
- Ya que los valores residen en el montículo, la interpretación más natural de asignación y paso de parámetros es usar una referencia semántica en vez de una copia semántica.
- Similarmente, la interpretación más natural de la prueba de igualdad es verificar la identidad del objeto. Sin embargo, ya que el programador requiere diferentes mecanismos de igualdad, se requieren dos operadores (equals y ==).
- Debido a que los valores residen en el montículo, debe existir algún mecanismo de manejo de memoria. Ya que la asignación es por referencia, es difícil para el programador determinar cuando un valor ya no será usado, por tanto, se necesita un sistema recolector de basura para recuperar la memoria que ya no se usa.

3.9. Polimorfismo

El término polimorfismo tiene raíces griegas y significa “muchas formas” (poly – muchos, morphos – forma). En los LOO, el polimorfismo es un resultado natural de las relaciones “es un” y los mecanismos de invocación de mensajes, herencia y el principio de sustitución. Una de las grandes ventajas de la POO es que estas características se pueden combinar en diversas formas, permitiendo formas de compartir y reutilizar código.

El polimorfismo puro ocurre cuando un método simple puede ser aplicado a diferentes tipos de argumentos. En el polimorfismo puro, existe un método (el cuerpo del código) y diversas interpretaciones (diversos significados). El otro extremo ocurre cuando existen diferentes métodos (cuerpos de código) todos denotados bajo el mismo nombre (sobrecarga). Entre estos dos extremos existe la sobreescritura de métodos y los métodos abstractos.

Un comportamiento es una acción o transformación que un objeto realiza a la cual está sujeto. Un comportamiento de un objeto se activa por la recepción de un mensaje o por la entrada en un estado particular. En ocasiones el mismo comportamiento se manifiesta de manera diferente en diferentes clases o subclases, propiedad que se denomina polimorfismo.

El **polimorfismo** es la *capacidad de tener métodos con el mismo nombre, pero con una implementación diferente o que un método sea aplicable a diferentes tipos de argumento*.

Por ejemplo, al tratar de frenar un auto siempre se debe oprimir el pedal del freno y el vehículo se detendrá sin importar si los frenos son de tambor o de disco.

Una *implementación específica de un comportamiento para una cierta clase* se denomina **método**. En un sistema polimórfico, un comportamiento puede tener más de un método que lo implemente. Por ejemplo, existirá un método frenar cuando se trata de frenos de disco y otro cuando son de tambor, pero el método sigue siendo frenar.

Un lenguaje de programación OO se diseña para seleccionar automáticamente el método correcto para implementar la operación, a partir de los datos asociados con la operación como parámetros, y el nombre de la clase de objeto, esto es llamado **enlace dinámico** (dinamic binding). El enlace dinámico, permite que las entidades del programa puedan referenciar en tiempo de ejecución a objetos de diferentes clases y está íntimamente relacionado con el concepto de herencia. En el ejemplo del frenado, el objeto seleccionará el método de frenado apropiado, basado en los parámetros que describen el tipo de frenos.

El polimorfismo permite agregar nuevas clases sin cambiar el código existente. Así, un nuevo tipo de transporte, como un camión puede agregarse con facilidad a la jerarquía. Sin embargo, hay que señalar que será necesario revisar los métodos.

- **Tipo estático:**
  - Tipo asociado en la declaración
- **Tipo dinámico:**
  - Tipo correspondiente a la clase del objeto conectado a la entidad en tiempo de ejecución
- **Conjunto de tipos dinámicos:**
  - Conjunto de posibles tipos dinámicos de una entidad

Los **tipos** de polimorfismos son:

- Cuando el tipo del objeto origen y el tipo del objeto destino son diferentes:
  - **Variables polimórficas**:
    Empleado obEmpleado = new Administrador()
  - **Métodos polimórficos** (Polimorfismo Puro)
- Cuando sólo se permiten objetos del mismo tipo de la referencia:
  - **Sobrecarga**
    - De operadores
    - Paramétrica (De constructores y métodos)
  - **Sobreescritura**
    - Invocación de métodos virtuales con variables polimórficas
    - Por refinamiento en los constructores
    - Clases abstractas e Interfaces

3.9.1. Variables Polimórficas

Con excepción de la sobrecarga, el polimorfismo en los LOO es posible sólo por la existencia de las *variables polimórficas* y la idea de sustitución. Una ***variable polimórfica*** es una variable que se declara de un tipo, pero puede contener un valor de ese tipo o un valor derivado de un subtipo del tipo declarado. Las variables polimórficas encierran el principio de sustitución. En otras palabras, aunque se espera un tipo para una variable, el tipo actual puede venir de cualquier valor que es un subtipo del tipo esperado.

En lenguajes como Smalltalk, todas las variables son potencialmente polimórficas. En estos lenguajes el tipo deseado se define por un conjunto de comportamientos esperados. Por ejemplo, un algoritmo puede usar un arreglo de valores, esperando que las operaciones sean definidas por cierta variable; cualquier tipo que defina el comportamiento apropiado es viable. Así, el usuario puede definir su propio tipo de arreglo y si las operaciones del arreglo se implementan usando el mismo nombre, se usa este nuevo tipo con el algoritmo existente.

En lenguajes con tipos estáticos, como Java, la situación es un poco más compleja. El polimorfismo ocurre en Java mediante las diferencias entre la clase declarada de una variable y la clase actual del valor de la variable contenida.

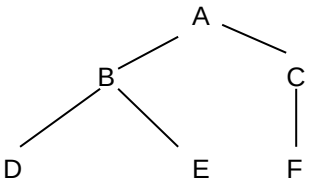
3.9.1.1. Tipos de Conversión entre clases

La conversión implícita y explícita de clases puede hacerse gracias al polimorfismo de variables, de este modo se puede tener:

- **Widening**: cuando una subclase se emplea como instancia de la superclase. (Ejemplo se invoca al método de la superclase que no ha sido sobrescrito). Es una conversión implícita.
- **Narrowing**: en este caso, la superclase se utiliza como una instancia de la subclase. Conversión explícita.

La conversión explícita sólo se puede realizar entre clases padre e hijas, no entre clases hermanas. Ejemplo:

Dadas las clases A, B, C, D, E y F en la siguiente jerarquía:



Si se declara	Se pueden recibir objetos
método (A objeto)	A, B, C, D, E y F
método (B objeto)	B, D y E
método (C objeto)	C y F

3.9.2. Sobrecarga

Una forma de polimorfismo en OO se da al usar un operador para aplicarlo a elementos de diferente tipo. Por ejemplo, al pretender sumar enteros, reales o complejos, se emplea el mismo símbolo +, esto se conoce como **sobrecarga de operadores**. En este caso, el compilador se encarga de determinar cuál es el método que se está invocando de acuerdo a los objetos involucrados en la operación.

La definición de dos o más métodos con el mismo nombre se conoce como **sobrecarga**. Esta facilidad de los LOO permite que cualquier función en una clase sea sobrecargada. En otras palabras, es una redefinición de una función con otro proceso con el mismo nombre, pero con diferente número y/o tipo de parámetros.

En algunas circunstancias, se puede requerir escribir varios métodos en la misma clase que hacen básicamente el mismo trabajo con diferentes argumentos.

Por ejemplo, un método simple que se requiere para mostrar un texto representando sus argumentos. Estos métodos pueden llamarse imprimir (). Ahora supongamos que se necesita un método diferente de impresión para imprimir cada tipo de dato int, float y String. Esto es razonable, porque tipos de datos diferentes requieren formatos diferentes y probablemente varíe el manejo. Se pueden crear tres métodos, llamados imprimirEntero (), imprimirFlotante () e imprimirCadena () respectivamente, sin embargo, esto puede ser tedioso, al permitir reutilizar un nombre de método para más de un método, se facilita la escritura.

Esto funciona, sólo si existe algo en las circunstancias bajo las cuales la llamada es realizada de modo que se distingue el método que se requiere usar. En el caso de los tres métodos de impresión, esta distinción se basa en el número y tipo de argumentos.

Reutilizando el nombre de método, se pueden escribir:

```
public void imprimir(int i)
public void imprimir(float f)
public void imprimir ()
```

Cuando se escribe el código para llamar uno de estos métodos, el método apropiado se selecciona dependiendo del tipo del argumento o argumentos proporcionados. Existen dos reglas para métodos sobrecargados:

- La lista de argumentos de la instrucción de llamada debe diferir suficiente para evitar la ambigüedad del método apropiado a llamar. Las promociones (por ejemplo, float a double) pueden ser aplicadas; esto puede causar confusión bajo ciertas circunstancias.
- El tipo regresado por el método puede ser diferente, pero no es suficiente que el valor de retorno sea diferente. La lista de argumentos debe ser diferente.

A esta característica se suele llamar ***polimorfismo en la sobrecarga***.

- Es el *nombre* de la función lo que es polimórfico
- Se resuelve en tiempo de compilación, según la signatura de la rutina.

- No es necesario que exista similitud semántica.
- En los lenguajes OO puede existir sobrecarga
  - dentro de una clase
  - entre clases no relacionadas (es fundamental)

3.9.3. Sobrecarga paramétrica

Otro estilo de sobrecarga en la cual, las operaciones en el mismo contexto permiten compartir el nombre y no son ambiguas gracias el número y tipo de argumentos proporcionados, se llama ***sobrecarga paramétrica*****.** La sobrecarga paramétrica es común encontrarla en los constructores. Un Empleado puede crearse sólo pasándole el nombre, o el nombre y la fecha de nacimiento o pasándole la fecha, nombre y sueldo. Hay varios constructores en esta clase, todos con el mismo nombre, el compilador decide cuál ejecutar basado en el número y tipo de argumentos.

La sobrecarga es un prerrequisito para otras formas de polimorfismo: sobreescritura, métodos abstractos y polimorfismo puro.

3.9.4. Sobrecarga de Constructores

Aunque cada clase tiene al menos un constructor, algunas clases pueden tener múltiples constructores (constructores sobrecargados), esto permite mayor flexibilidad al momento de crear objetos de esa clase. Existen tres tipos de constructores:

- **Constructor por omisión.** Es aquel que se proporciona automáticamente, sin necesidad de declarar alguno, no tiene parámetros y su cuerpo no contiene instrucciones. Cuando se crea el objeto inicializa los atributos con los valores por omisión. Permite crear objetos instanciados con *new Xxx( )* sin tener que escribir un constructor.
- **Constructor con parámetros**. Es el que tiene una lista de argumentos, a los cuales se les asigna algún valor al momento de crear el objeto. Estos valores sirven para dar valor a los atributos del objeto creado.
- **Constructor con valores por omisión**. Es el que asigna valores por omisión a los atributos en caso de que no se asignen parámetros.

La existencia, número y tipo de parámetros determina a qué tipo de constructor se llama. Es importante mencionar que los constructores por omisión no existen al momento de declarar algún otro tipo de constructor dentro de una clase.

Es una buena práctica de programación definir al menos un constructor con valores por omisión. La sobrecarga de constructores puede tener una desventaja, si existen dos constructores con el mismo tipo y número de parámetros, no se puede distinguir a cuál invocar, por ejemplo, si se desea crear un círculo con el valor de su radio o con el valor de su diámetro, y ambos valores son de tipo real, no se podría distinguir cuál es el constructor que se desea ejecutar.

En Java, para crear objetos con un constructor determinado, se emplea el operador **new** seguido del nombre de la clase que obviamente corresponde al nombre de un constructor y entre paréntesis se agregan los argumentos.

3.9.5. Sobreescritura

En este caso en una clase, generalmente una superclase abstracta, se define un método general para un mensaje particular que se hereda y es usado por las subclases. En al menos una subclase, sin embargo, un método con el mismo nombre se define y esconde el acceso al método general para instancias de esa clase o en el caso de refinamiento, retoma el acceso al método general. Se dice entonces que el segundo ***método sobrescribe*** al primero.

Los constructores siempre usan ***sobreescritura por refinamiento***, ya que un constructor de una clase hija siempre invoca al constructor de la clase padre. Esta invocación tiene lugar antes de que el código del constructor se ejecute. Si el constructor de la clase padre requiere argumentos, se usa la referencia super como si fuera un método.

```
public Cuadrado (int x, int y, int lado) {
        super(x, y);
        this.lado = lado;
}
```

Cuando se usa ***super***, ésta debe ser la primera instrucción ejecutada. Si no se especifica super, automáticamente se invoca al constructor por omisión (sin argumentos).

3.9.5.1. Invocación de Métodos Virtuales

La invocación a ***métodos virtuales*** se realiza cuando se tienen métodos sobrescritos y se maneja una variable polimórfica, en este caso, el runtime, determina cuál es el método que debe ejecutar, ya sea el de la superclase o el de la subclase, dependiendo del tipo del objeto al que se esté haciendo referencia la variable polimórfica.

3.9.6. Métodos abstractos

Un ***método abstracto*** se declara como *abstract* y se puede ver como un *método que se especifica en la clase padre, pero debe implementarse en la clase hija*. Las interfaces pueden también ser vistas como un método para definir métodos abstractos. Ambos pueden considerarse como *generalizaciones de la sobreescritura*. En ambos casos, el comportamiento descrito en una clase padre se modifica en la clase hija. En un método abstracto, sin embargo, el comportamiento en la clase padre es esencialmente nulo y la actividad útil se describe como parte del código proporcionado por la clase hija.

Una ventaja de los métodos abstractos es conceptual, ya que permite a los programadores pensar en una actividad asociada con una abstracción a un nivel alto. Otra razón para usar métodos abstractos es que permite enviar un mensaje a un objeto sólo si el compilador puede determinar que existe un método que corresponda con el mensaje seleccionado.

El modificador abstract no puede emplearse en constructores, en métodos estáticos (static) o en métodos privados. Todos los métodos declarados en una interfaz deben ser públicos.

3.9.7. Polimorfismo Puro

Muchos autores reservan el término polimorfismo (o ***polimorfismo puro***) para situaciones donde un método puede usarse con una variedad de argumentos, y el término sobrecarga para situaciones donde múltiples métodos son denotados con un mismo nombre. La habilidad de crear métodos polimórficos es una de las técnicas más poderosas en POO. Permite que el código se escriba sólo una vez, a un nivel alto de abstracción, y ajustarlo si es necesario para cubrir varias situaciones.

Un ejemplo de polimorfismo puro es el método *valueOf*, que se encuentra en la clase String. Este método se usa para generar una descripción textual de un objeto.

El método toString se define en Object y se redefine en muchas clases. Ya que las diversas versiones de toString producen diferentes efectos, el método valueOf también produce diferentes resultados.

```
public class String {
        …
```

```
        public static String valueOf (Object obj) {
            if (obj == null)
                return “null”;
            return obj.toString();
        }
}
```

La característica importante de la definición del polimorfismo puro, en relación a la sobrecarga y sobreescritura, es que existe un solo método con un nombre, usado con una variedad de diferentes argumentos (sólo un método y puede recibir “cualquier” objeto de la jerarquía que se defina en el parámetro especificado). En la mayoría de los casos, el cuerpo de este algoritmo hace uso de otras formas de polimorfismo, como las variables polimórficas usadas como argumentos, que invocan a métodos virtuales (sobrescritos).

3.9.7.1. Enlace Dinámico

El enlazado dinámico está estrechamente relacionado con la herencia y consiste en llamadas a métodos con parámetros cuyo tipo no se conoce hasta el momento de la ejecución. El runtime observa el parámetro y determina el tipo adecuado del parámetro. El enlazado dinámico sólo puede darse entre superclases y subclases.

En este caso, no es el tipo de la variable el que determina la invocación, sino el tipo del objeto al que hace referencia la variable; gracias a esto, es posible tener arreglos heterogéneos, ya que se depende de cómo se asignen los elementos.

Se puede recibir un parámetro de tipo Object y posteriormente acceder al valor del parámetro de dos modos:

- Por medio de la prueba de pertenencia, mediante el operador *instanceof* se puede determinar el tipo del parámetro y tratarlo como tal.
- La segunda es realizando una conversión explícita y luego usarlo como el tipo adecuado.

La herencia sin polimorfismo es posible, pero ciertamente no es muy útil. El polimorfismo y el enlace dinámico van de la mano. En presencia del polimorfismo, el enlace de un método se determina hasta la ejecución, no en la compilación.

3.9.8. Polimorfismo Simple

Básicamente, el polimorfismo es un concepto de teoría de tipos en el que un nombre puede denotar instancias de muchas clases diferentes en tanto estén relacionadas por alguna superclase común. Cualquier objeto denotado por este nombre es, por tanto, capaz de responder a algún conjunto de operaciones de diversas formas.

Los lenguajes convencionales con tipos, como Pascal, se basan en la idea de que las funciones y los procedimientos, y, por tanto, los operandos, tienen un único tipo. Tales lenguajes se dice que son ***monomórficos***, en el sentido de que todo valor y variable puede interpretarse que tiene un tipo y sólo uno. Los lenguajes monomórficos pueden contrastarse con los polimórficos en los que algunos valores y variables pueden tener más de un tipo, a este tipo de polimorfismo se le denomina ***polimorfismo paramétrico***. El concepto de polimorfismo donde un operador o método puede significar cosas distintas es denominado también *sobrecarga*.

Sin polimorfismo, el desarrollador tiene que escribir código que consiste en grandes sentencias de selección múltiple, con el polimorfismo, cada objeto conoce implícitamente su propio tipo. El polimorfismo es más útil cuando existen muchas clases con los mismos protocolos.

En el ***polimorfismo simple***, los métodos están especializados (son polimórficos) respecto a un factor, que es el objeto para el que se invoca la operación.

3.9.9. Polimorfismo Múltiple

En lenguajes como CLOS es posible escribir operaciones llamadas ***multimétodos*** que son polimórficas respecto a más de un factor. En lenguajes que sólo soportan polimorfismo simple, como Java, se puede simular este comportamiento polimórfico múltiple usando un recurso llamado ***selección doble (double dispatching)***. La invocación de operaciones es polimórfica según la clase concreta del objeto actual, y a continuación exhibe comportamiento polimórfico según la subclase exacta del argumento recibido como parámetro.

Este recurso puede extenderse hasta cualquier grado de selección polimórfica.

Tarea:

Leer en otras fuentes sobre los temas vistos en esta unidad, hacer un resumen de la unidad (máximo 1 cuartilla), no olvidar conclusiones y bibliografía).

Comparar y explicar las similitudes y diferencias entre:

1. Herencia y composición
2. Clase abstracta e interfaz
3. Tipos genéricos y polimorfismo

Investigar pará qué sirven las siguientes “Colecciones” Java: ArrayList<T>, List, Queue<T>, Set<T>, Map<K, V> y HashMap<K, V>, y cuáles son las operaciones habituales que se pueden realizar con ellas.

Crear una jerarquía de clases que representen los diferentes productos que se pueden encontrar en un supermercado (alimentación, ropa, electrónica, **etc**.). Se tendrá un conjunto de características comunes (precio, nombre, código de barras, etc.) y un conjunto de características específicas de cada tipo de producto.