Contenido

Introducción

La programación orientada a objetos es el paradigma de programación imperante en la actualidad y ha reemplazado las técnicas de desarrollo estructurado vigentes.

La programación orientada a objetos (POO) es un enfoque conceptual específico para diseñar programas utilizando un lenguaje que se centra en los objetos, como C++ o Java, cuyas propiedades más importantes son:

- ✓ Abstracción
- ✓ Encapsulamiento y ocultación de datos
- ✓ Herencia
- ✓ Polimorfismo
- ✓ Reusabilidad o reutilización de código

En esencia, el presente trabajo se concentra en las propiedades más importantes de este paradigma que son la Herencia y el Polimorfismo.

La Herencia es la propiedad que permite definir nuevas clases usando como base las ya existentes; la nueva clase, también llamada *derivada*, hereda los atributos y comportamiento que son específicos de ella; la herencia es una herramienta poderosa que proporciona un marco adecuado para producir *software* fiable, comprensible, de bajo coste, adaptable y reutilizable.

El polimorfismo muestra cómo realizar operaciones polimórficas en el contexto de herencia de clases, ésta y la redefinición de los métodos heredados es el soporte del polimorfismo; este término significa muchas o múltiples formas, esto es la posibilidad que tiene una entidad para referirse a instancias diferentes en tiempo de ejecución. En la práctica, el polimorfismo permite hacer referencias a otros objetos de clases por medio del mismo elemento de programa y realizar la misma operación de formas distintas, de acuerdo con el Objeto al que hace referencia en cada momento.

1. Herencia

1.1 Clase base

A menudo, un objeto de una clase *es un* objeto de otra clase también. Al crear una clase, en vez de declarar miembros completamente nuevos, se puede designar que la nueva clase herede los miembros de una existente. La cual se conoce como **superclase** y la clase nueva como **subclase**. (El lenguaje de programación C++ se refieres a la superclase como la **clase base** y a la subclase como **clase derivada**).

Las superclases tienden a ser "más generales", y las subclases "más específicas". Por ejemplo, un PrestamoAuto *es un* Prestamo, así como PrestamoMejoraCasa y PrestamoHipotecario. Por ende, en Java se puede decir que la clase PrestamoAuto hereda de la clase Prestamo. En este contexto, dicha clase es una superclase y la clase PrestamoAuto es una subclase. Un PrestamoAuto es un tipo específico de Prestamo, pero es incorrecto afirmar que todo Prestamo *es un* PrestamoAuto; el Prestamo podría ser cualquier tipo de crédito.

1.2 Clases derivadas

En la herencia o relación *es-un*, es la relación existente entre dos clases: una es la derivada que se crea a partir de otra ya existente, denominada *base*, la nueva hereda de la ya existente; por ejemplo: si existe una clase Figura y se desea crear una clase Triangulo, esta última puede derivarse de la primera pues tendrá en común con ella un estado y un comportamiento, aunque tendrá sus características propias; Triangulo *es-un* tipo de Figura.

Tabla 1 Ejemplos de Herencia

Superclase	Subclases
Estudiante	EstudianteGraduado, EstudianteNoGraduado
Figura	Circulo, Triangulo, Rectángulo, Esfera, Cubo
Prestamo	PrestamoAutoMovil, PrestamoMejoraCasa, PrestamoHipotecario
Empleado	Docente, Administrativo
CuentaBancaria	CuentaDeCheques, CuentaDeAhorros

Como la clase base y la clase derivada tienen código y datos comunes, es evidente que si se crea la clase derivada de modo independiente, se duplicaría mucho de lo que ya se escribió para la clase base; entonces la herencia en la clase derivada es una ampliación de la base pues aquella dispone de sus atributos y métodos propios más los atributos y métodos heredados; por ello, Java utiliza la palabra reservada **extends** para crear clases derivada o clases que son extensión de otra, de modo que la nueva hereda todos los miembros datos y los métodos que pertenecen a la ya existente.

La declaración de derivación de clases debe incluir la palabra reservada **extends** y a continuación el nombre de la clase base de la que se deriva; su formato es el siguiente:

class nombre_clase **extends** nombre_clase_base

Ejemplo 1

Declaración de las clases Programador y Triangulo; la primera deriva o extiende a la clase Empleado; la segunda de Figura

```
public class Programador extends Empleado{

}

public class Triangulo extends Figura{

}
```

1.3 Herencia simple

La herencia simple es la más típica, la que se puede encontrar en cualquier lenguaje moderno como Java o C#.

La herencia simple es una relación entre una clase padre (clase base) y una clase hija (clase derivada) llamada "es un tipo de", que muchas veces se abrevia como *es-a*.

La herencia es simple cuando la clase derivada que estamos considerando sólo tiene una clase base.

Ejemplo 2

Considerar una clase Prestamo y una clase derivada de ella: Hipoteca

```
public class Prestamo {
    final int MAXTERM = 22;
    protected float capital;
    protected float tasaIntereres;

    public void prestamo(float p, float r)
    {
        // acciones que hara el metodo
    }


}
```

Las variables capital y tasaInteres no se repiten en la clase derivada, cuya declaración es:

```
public class Hipoteca extends Prestamo{
    private int numRecibos;
    private int recibosPorAnyo;
    private float pago;

    public Hipoteca()
    {}


}
```

1.4 Herencia Múltiple

El lenguaje C++ permite la herencia múltiple. Una clase puede derivar de varias clases base. La sintaxis es la misma que la utilizada para la herencia simple, separando por comas cada una de las clases base de las que hereda.

Aunque el lenguaje nos brinde la posibilidad de utilizar la herencia múltiple, existe un fuerte debate en lo referente a si tiene sentido o no usarla dentro de un diseño de software. Las principales razones que desaconsejan su uso es que introduce mayor

oscuridad en el código sin aportar ningún beneficio que no se pueda lograr usando solo herencia simple. Esta parece ser la postura más aceptada hoy en día y así lo reflejan los nuevos lenguajes, como Java, en los que ha desaparecido la herencia múltiple.

Ejemplo 3

Una familia desea registrar su plan vacacional. Si se sabe que el boleto contiene los siguientes atributos: precio, ciudad de origen, ciudad de destino y numero; posteriormente en el Hotel en que se van a registrar le proporcionan la siguiente información: precio de habitación, número de habitación, tipo de habitación. Desarrollar las clases Boleto, Hotel y Plan vacacional esta última debe heredar de las otras dos clases.

```
#include<iostream>

using namespace std;

class Boleto
{
    private:
        float precio;
        char Numero[64], CiudadOrigen[64], CiudadDestino[64];
    public:
        Boleto() {};
        Boleto(float, char*, char*, char*){};
        void imprimirDatos();
}
Boleto :: Boleto(float Prec, char Num[], char CO[], char CD[])
{
    Precio = Prec;
    strcpy(Numero, Num);
    strcpy(CiudadOrigen, CO);
    strcpy(CiudadDestino, CD);
}
void Boleto :: imprimirdatos()
{
    cout << endl << endl;
    cout << "----- Imprimir los Datos del Boleto -----" << endl;
    cout << "Numero del boleto : " << Numero << endl;
    cout << "Precio : $ " << Precio << endl;
    cout << "De la ciudad: " << CiudadOrigen << " a la ciudad: " << CiudadDestino <<
    endl;
}
class Hotel
{
    private:
        float PrecioHab;
        int NumHab;
        char TipoHab[10];
    public:
        Hotel(){}
        Hotel(float, int, char *);
        void imprimirDato();
}
```

```
Hotel::Hotel(float PrecH, int NH, char TH)
{
    PrecioHab = PrecH;
    NumHab = NH;
    strcpy(TipoHAb, TH);
}
void Hotel::imprimirDato()
{
    cout << endl << endl;
    cout << "----Imprimir Datos de la Habitacion del Hotel----" << endl;
    cout << "Numero de habitacion: " << NumHab << endl;
    cout << "Precio: $" << PrecioHab << endl;
    cout << "Tipo de Habitacion: " << TipoHab << endl;
}
class PlanVacac : public Boleto, public Hotel
{
    private:
        char Descripcion[80];
        int TotalDias;

    public:
        PlanVacac(){}
        PlanVacac(float, char*, char*, char*, float, int, char *, char *, int);
        void imprmirDatos();
}
PlanVacac::PlanVacac(float PB, char NB[], char CO[],
    char CD[], float PH, int NH, char TH[], char Desc[],
    int TD):Boleto(PB,NB,CO,CD):Hotel(PH, NH,TH)
{
    strcpy(Descripcion, Desc);
    TotalDias = TD;
}
void PlanVacac::imprmirDatos()
{
    cout << endl << endl;
    cout << "----Imprimir los Datos del Plan Vacacional----" << endl;
    cout << "Descripcion: " << Descripcion << endl;
    cout << "Total de Dias: "<< TotalDias << endl;
    Boleto::imprimirdatos();
    Hotel::imprimirDato();
}
```

```
PlanVacac Lee()
{
    char CO[64], CD[64],NumBol[64], TH[10],Desc[80];
    float prec, precHab;
    int numHab, td;

    cout << "De que ciudad sale?" ;
    gets(CO);
    cout << "A que ciudad llega?" ;
    gets(CD);
    cout << "Precio del Boleto: ";
    cin >> Prec;
    fflush(stdin);
    cout << "Numero de boleto: ";
    gets(NumBol);
    cout << "Tipo de Habitacion: ";
    gets(TH);
    cout << "Precio de Habitacion: ";
    cin >> PrecHab;
    cout << "Numero de habitacion asignada: ";
    cin >> NumHab;
    fflush(stdin);
    cout << "Tipo de paquete: ";
    gets(Desc);
    cout << "Total de Dias: ";
    cin >> TD;

    PlanVacac paquete(Prec, NumBol, CO, CD, PrecHab, NumHab, TH, Desc, TD);
    return paquete;
}

int main()
{
    PlanVacac viaje;
    viaje = Lee();
    viaje.imprimirDatos();
    System("pause>nul");
    return 0;
}
```

Como ya se había mencionado antes Java no soporta la herencia múltiple; pero en el diseño de una aplicación hay herencia múltiple y no puede transformarse a simple, entonces la primera puede simularse con el mecanismo **interface – implements**

Una clase puede derivar de otra e implementar una o más interfaces; por ejemplo, la clase Helicoptero deriva de Avion e implementa dos interfaces:

```
public class Helicoptero extends Avion implements Tranporte, Vertical{

}
```

Por otra parte, la herencia múltiple siempre se puede eliminar y convertir en simple si el lenguaje de implementación no la soporta o considera que tendrá dificultades en etapas posteriores a la implementación real.

1.5 Clase abstracta

Supongamos un esquema de herencia que consta de la clase Profesor de la que heredan ProfesorInterino y ProfesorTitular. Es posible que todo profesor haya de ser o bien ProfesorInterino o bien ProfesorTitular, es decir, que no vayan a existir instancias de la clase Profesor. Entonces, ¿qué sentido tendría tener una clase Profesor?

El sentido está en que una superclase permite unificar campos y métodos de las subclases, evitando la repetición de código y unificando procesos. Ahora bien, una clase de la que no se tiene intención de crear objetos, sino que únicamente sirve para unificar datos u operaciones de subclases, puede declararse de forma especial en Java: como clase abstracta. La declaración de que una clase es abstracta se hace con la sintaxis public abstract class NombreDeLaClase { … }. Por ejemplo public abstract class Profesor. Cuando utilizamos esta sintaxis, no resulta posible instanciar la clase, es decir, no resulta posible crear objetos de ese tipo. Sin embargo, sigue funcionando como superclase de forma similar a como lo haría una superclase “normal”. La diferencia principal radica en que no se pueden crear objetos de esta clase.

Declarar una clase abstracta es distinto a tener una clase de la que no se crean objetos. En una clase abstracta, no existe la posibilidad. En una clase normal, existe la posibilidad de crearlos aunque no lo hagamos. El hecho de que no creemos instancias de una clase no es suficiente para que Java considere que una clase es abstracta. Para lograr esto hemos de declarar explícitamente la clase como abstracta mediante la sintaxis que hemos indicado. Si una clase no se declara usando abstract se cataloga como “clase concreta”. En inglés abstract significa “resumen”, por eso en algunos textos en castellano a las clases abstractas se les llama resúmenes. Una clase abstracta para Java es una clase de la que nunca se van a crear instancias; simplemente va a servir como superclase a otras clases. No se puede usar la palabra clave new aplicada a clases

abstractas. En el menú contextual de la clase en BlueJ simplemente no aparece, y si intentamos crear objetos en el código nos saltará un error.

A su vez, las clases abstractas suelen contener métodos abstractos: la situación es la misma. Para que un método se considere abstracto ha de incluir en su signatura la palabra clave **abstract**. Además un método abstracto tiene estas peculiaridades:

a) No tiene cuerpo (llaves): sólo consta de signatura con paréntesis.
b) Su signatura termina con un punto y coma.
c) Sólo puede existir dentro de una clase abstracta. De esta forma se evita que haya métodos que no se puedan ejecutar dentro de clases concretas. Visto de otra manera, si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.
d) Los métodos abstractos forzosamente habrán de estar sobrescritos en las subclases. Si una subclase no implementa un método abstracto de la superclase tiene un método no ejecutable, lo que la fuerza a ser una subclase abstracta. Para que la subclase sea concreta habrá de implementar métodos sobrescritos para todos los métodos abstractos de sus superclases.

Un método abstracto para Java es un método que nunca va a ser ejecutado porque no tiene cuerpo. Simplemente, un método abstracto referencia a otros métodos de las subclases. ¿Qué utilidad tiene un método abstracto? Podemos ver un método abstracto como una palanca que fuerza dos cosas: la primera, que no se puedan crear objetos de una clase. La segunda, que todas las subclases sobrescriban el método declarado como abstracto.

Ejemplo 4

Desarrollar una clase abstracta llamada `Instrumento` y que de ellas se deriven otras clases y desarrollar una clase que haga uso de la clase abstracta.

```
abstract class Instrumento {
    public abstract void tocar();
    public String tipo() {
        return "Instrumento";
    }
    public abstract void afinar();
}
```

```java
class Guitarra extends Instrumento {
    public void tocar() {
        System.out.println("Guitarra.tocar()");
    }
    public String tipo() { return "Guitarra"; }
    public void afinar() {}
}
class Piano extends Instrumento {
    public void tocar() {
        System.out.println("Piano.tocar()");
    }
    public String tipo() { return "Piano"; }
    public void afinar() {}
}

class Saxofon extends Instrumento {
    public void tocar() {
        System.out.println("Saxofon.tocar()");
    }
    public String tipo() { return "Saxofon"; }
    public void afinar() {}
}
class Ukelele extends Guitarra {
    public void tocar() {
        System.out.println("Ukelele.tocar()");
    }
    public String tipo() { return "Ukelele"; }
}
```

```java
public class Musica2 {

    static void afinar(Instrumento i) {
        i.tocar();
    }

    static void afinarTodo(Instrumento[] e) {
        for(int i = 0; i < e.length; i++)
            afinar(e[i]);
    }
    public static void main(String[] args) {
        Instrumento[] orquesta = new Instrumento[5]
        int i = 0;
        orquesta[i++] = new Guitarra();
        orquesta[i++] = new Piano();
        orquesta[i++] = new Saxofon();
        orquesta[i++] = new Ukelele();
        afinarTodo(orquesta);
    }
}
```

Entonces podemos decir en otras palabras que las clases abstractas son útiles para realizar implementaciones parciales, es decir, realizar partes de sus funcionalidades postergando el resto de sus subclases.

1.6 Resumen

- La relación entre clases *es-un* tipo que indica relación de herencia.
- La relación *es-un* también se puede expresar como generalización-especialización ya que es una relación transitiva. Esta manera de relacionar las clases se expresa en Java con la derivación o extensión de clases.
- Una clase nueva que se crea a partir de una ya existente utilizando la propiedad de herencia, se denomina *clase derivada o subclase*. La clase de la cual se hereda se llama *clase base o superclase*.
- En java. La herencia siempre es imple y publica; la clase derivada hereda todos los miembros de la clase base excepto los miembros privados
- Un objeto de la clase derivada se crea siguiendo este orden: primero la parte del objeto correspondiente a la base y a continuación se crea la parte propia de la derivada; para llamar al constructor de la primera desde el constructor de la última se emplea la palabra reservada `super`; la primera sentencia del constructor de la derivada debe ser llamada al constructor de la base, es decir, `super(argumentos)`.

2. Polimorfismo

En POO, el polimorfismo permite que diferentes objetos respondan de modo distinto al mismo mensaje; adquiere su máxima potencia cuando se utiliza en unión de herencia; se establece con la ligadura dinámica de métodos, con la cual no es preciso decidir el tipo de objeto hasta el momento de la ejecución.

En Java las variable de referencia de una clase son polimórficas ya que pueden referirse a un objeto de su clase o a uno de alguna de sus subclases; por ejemplo; en la jerarquía de libros, una variable de tipo **Libro** se puede referir a un objeto de tipo **Libro Impreso** o a un objeto de tipo **Libro Electrónico**.

```
Libro xx;

xx = new LibroElectronico("Biologia creativa",500);

xx = new LibroImpreso("La botica del palacio",144,67.0);
```

Solo durante la ejecución del programa, la máquina virtual Java conoce el objeto concreto al que se refiere la variable **xx**.

El polimorfismo se usa a través de referencias a la clase base; si, por ejemplo, se dispone de una colección de objetos **Archivo** en un arreglo, éste se almacena referencias a dichos objetos que apuntan a cualquier tipo de archivo; cuando se actúa sobre los archivos mencionados, simplemente basta recorrer el arreglo e invocar el método apropiado mediante la referencia de instancia; naturalmente, para realzar esta tarea los métodos deben ser declarados como abstractos en la clase **Archivo**, que es la clase base, y redefinirse en las derivadas.

Para utilizar el polimorfismo en Java se deben seguir las siguientes reglas:

1. Crear una jerarquía de clases con las operaciones importantes definidas por los métodos miembro declaradas como abstractos en la base.
2. Las implementaciones específicas de los métodos abstractos se deben hacer en las clases derivadas, cada una de ellas puede tener su propia versión del método; por ejemplo, la implementación del método **annadir()** varía de un tipo de fichero a otro
3. Las instancias de estas clases se manejan a través de una referencia a la base mediante la ligadura dinámica, la cual es la esencia del polimorfismo en Java.

Realmente no es necesario declarar abstractos los métodos en la base, si después se redefinen con la misma signatura en la derivada.

El polimorfismo hace su sistema más flexible, sin perder ninguna de las ventajas de la compilación estática de tipos que tienen lugar en tiempo de compilación; tal es el caso de Java.

Las aplicaciones más frecuentes del polimorfismo son:

- **Especialización de clases derivadas**. Es decir, especializar clases que han sido definidas; por ejemplo: `Cuadrado` es una especialización de la clase `Rectángulo` porque cualquier cuadrado es un tipo de rectángulo; esta clase de polimorfismo aumenta la eficiencia de la subclase, mientras conserva un alto grado de flexibilidad y permite un medio uniforme de manejar rectángulos y cuadrados.
- **Estructuras de datos heterogéneos**. A veces es útil poder manipular conjuntos similares de objetos con el polimorfismo se pueden crear y manejar fácilmente estructuras de datos heterogéneos que son fáciles de diseñar y dibujar; sin perder la comprobación de tipos de los elementos utilizados.
- **Gestión de una jerarquía de clases**. Son colecciones de clases altamente estructuradas con relaciones de herencia que se pueden extender fácilmente.

2.1 Variables polimórficas

En Java, las variables que contienen objetos son variables polimórficas. El término polimórfico (literalmente: muchas formas) se refiere al hecho de que una misma variable puede contener objetos de diferentes tipos (del tipo declarado o de cualquier subtipo del tipo declarado). El polimorfismo aparece en los lenguajes orientados a objetos en numerosos contextos, las variables polimórficas constituyen justamente un primer ejemplo.

Algunos ejemplos de variables polimórficas pueden ser:

- ArrayList
- LinkedList

Ejemplo 5

Se declaran las clases correspondientes a la jerarquía Barco, DeVapor, Carguero y Velero; cada una de las clases descendientes de Barco disponen del método alarma() que escribe un mensaje en pantalla; en el programa se define un arreglo de referencias a Barco, se crean objetos de las clases derivadas DeVapor, Carguero y Velero, asigna esos objetos al arreglo y por ultimo se llama al método alarma().

```
public class Barco {

    public Barco()
    {
        System.out.println("\tSe crea parte de un barco");
    }
    public void alarma()
    {
        System.out.println("\tS.O.S desde un barco");
    }
}

public class DeVapor extends Barco{

    public DeVapor()
    {
        System.out.print("Se crea la parte de un barco de vapor");
    }
    public void alarma()
    {
        System.out.println("\tS.O.S desde un barco de vapor");
    }
    public void alarma(String msg)
    {
        System.out.print("\tMensaje: " + msg + "enviado desde un barco de vapor");
    }
}


public class Velero extends Barco{

    public Velero()
    {
        System.out.println("Se crea la parte del barco velero");
    }
    public void alarma()
    {
        System.out.print("\tS.O.S desde un velero");
    }
}
```

```java
public class Carguero extends DeVapor{
    public Carguero()
    {
        System.out.println("\tSe crea la parte del barco carguero");
    }
    public void alarma()
    {
        System.out.println("\tS.O.S desde un carguero");
    }
    public void alarma(String msg)
    {
        System.out.println("\tMensaje: " + msg + "enviado desde un carguero");
    }
}


public class AlarmaDeBarcos {
    public static void main(String[] args) {
        Barco[] bs = new Barco[3];
        DeVapor mss = new DeVapor();
        System.out.println();
        Velero vss = new Velero();
        Carguero css = new Carguero();
        bs[0] = mss;
        bs[1] = vss;
        bs[2] = css;
        for(int i = 0; i < 3 ;)
        {
            bs[i++].alarma();
        }
        mss = css;
        mss.alarma("A 3 horas del puerto");
    }
}
```

2.2 Reutilización de código

Otra propiedad fundamental de la programación orientada a objetos es la reutilización o *reusabilidad*; este concepto significa que una vez que se ha creado, escrito y depurado una clase, se puede poner a disposición de otros programadores; de manera similar, al uso de las bibliotecas de funciones en un lenguaje de programación procedimental como C.

El concepto de herencia en Java proporciona una ampliación o extensión importante a la idea de *reusabilidad*; una clase existente se puede ampliar añadiéndole nuevas características, atributos y operaciones.

3. Conclusión

En este trabajo se aprendió el concepto de herencia: la habilidad de crear clases mediante la absorción de los miembros de una clase existente, mejorándolos con nuevas capacidades. Aprendí las nociones de las superclases y subclases, y utilizamos la palabra clave `extends` para crear una subclase que hereda miembros de una superclase.

También vimos el polimorfismo: la habilidad de procesar objetos que comparten la misma superclase en una jerarquía de clases, como si todos fueran objetos de la superclase. Se habló sobre como el polimorfismo facilita la extensibilidad y el mantenimiento de los sistemas.

Se presentó la noción de clases abstractas, las cuales nos permiten proporcionar una superclase apropiada, a partir de la cual otras clases pueden heredar. Aprendí que las clases abstractas se pueden declarar métodos abstractos en cada una de sus subclases debe implementar para convertirse en una clase concreta, y que un programa puede utilizar variables de una clase abstracta para invocar implementaciones en las subclases de los métodos abstractos en forma polimórfica.

Por último se mencionó sobre la declaración e implementación de una interfaz, como otra manera de obtener el comportamiento polimórfico.

Ahora ya estoy familiarizado con las clases, los objetos, el encapsulamiento, la herencia, las interfaces y el polimorfismo: los aspectos más esenciales de la programación orientada a objetos.