PROGRAMACIÓN ORIENTADA A OBJETO

SEMANA 5

Características de la POO: El modelo de objeto en Java

iacc

APRENDIZAJES ESPERADOS

El estudiante será capaz de:

- El estudiante será capaz de: Programar utilizando abstracción, encapsulamiento, herencia y polimorfismo en sus atributos y métodos, de igual manera realizará sobre carga de métodos.

ÍNDICE

INTRODUCCIÓN

El lenguaje de Programación Java, usa y extiende muchos conceptos de programación orienta a objetos (POO).

No hay un acuerdo aceptado por todo el mundo respecto a cuáles son las características que definen la POO, pero al menos todos concuerdan en estas tres:

Abstracción.

Encapsulación.

Herencia.

Sin embargo, para entender estas características aplicadas a JAVA es necesario comprender que un objeto se define como una entidad que tiene un estado, un método (comportamiento) y una identidad. Un objeto es una abstracción de algún hecho o ente del mundo real, con atributos que representan sus características o propiedades, y métodos que emulan su comportamiento o actividad. Todas las propiedades y métodos comunes a los objetos se encapsulan o agrupan en clases.

A continuación, se estudiará cómo se implementan las relaciones entre objetos, la herencia, las abstracciones la implementación de polimorfismos y se repasarán los métodos y constructores.

Además, se verá cómo se controla el nivel de visibilidad, fundamentos del encapsulamiento y el modularidad con la aplicación de métodos abstractos, su implementación en herencia de clases.

1. CARACTERÍSTICAS DE LA POO

IMPORTANTE

Comencemos con una reflexión:

¿Cuáles son las características de la POO y como las podemos identificar en nuestros programas informáticos?

1.1. ABSTRACCIÓN

¿Qué ventajas le da al programador poder abstraer una clase?

Con anterioridad se ha mencionado el concepto de abstracción y herencia, en este caso las clases abstractas son aquellas que no permiten instanciación y tampoco generan un objeto, lo que quiere decir que su principal función es ser heredada por otra clase.

Esto es útil en el desarrollo de programas en Java, dado que hay objetos que se requieren constantemente, y redefinirlos no representa utilidad alguna, al contrario, heredarlos y hacer referencia a estos es más conveniente y eficiente, a diferencia de las interfaces, la abstracción permite al desarrollador el control sobre cuánto de la clase abstracta tiene que implementar.

1.1.1. GENERACIÓN DE CLASES ABSTRACTAS

Para explicar la abstracción de clase se utilizará el ejemplo de InstrumentoMusical, obsérvese, cualquier clase que herede tendrá que tener nombre e implementar un método llamado tocar, pero la abstracción no solo se limita a la primera abstracción, para el ejemplo se ha abstraído InstrumentoCuerda que extiende a InstrumentoMusical, análogamente se podría haber hecho con instrumentos de viento para el caso de que los objetos fuesen flautas, saxofón o zampoña.

Dado que en el ejemplo se quiere mostrar, instrumentos de cuerdas, un ejemplo es la GuitaraElectrica como se aprecia en la imagen 1, la que se extiende a InstrumentoCuerda y por consecuencia a InstrumentoMusical, lo que se grafica al momento de crear el constructor de esta clase, que por defecto tendrá el nombre que lo hereda, de InstrumentoMusical y NumeroDeCuerdas que lo hereda InstrumentoCuerda.

```
abstract class InstrumentoMusical { //clase abstracta general
    protected String nombre;  //comun a todos los instrumentos Musicales
    abstract public void tocar(); //abstraccion de tocar
}
abstract class InstrumentoCuerda // abstraccion de instrumentos de cuerda
        extends InstrumentoMusical { //extiende a instrumentos Musicales
    protected int numeroDeCuerdas; //propio de los instrumentos de cuerda
}
public class GuitaraElectrica extends InstrumentoCuerda {
    /*la guitarra electrica es un tipo de instrumento de cuerda
     y por ende un tipo de intrumento Musical */
    public GuitaraElectrica() {
        super(); //invoca el constructor de la clase superior
        this.nombre = "Guitarra";
        this.numeroDeCuerdas = 6; //la guitarra generica tiene 6 cuerdas
    }
    public void tocar() {//la implementacion de tocar para GuitaraElectrica
        System.out.println("Esta " + nombre + " electrica con "
        + numeroDeCuerdas + " cuerdas  rockea!");
    }
}
```

Imagen 1. Clase Instrumento Musical
Fuente: Elaboración Propia

1.1.2 MÉTODOS ABSTRACTOS EN JAVA

Los métodos abstractos propios de las clases abstractas, permiten definir métodos que las clases que hereden pueden implementar o redefinir, siguiendo con el ejemplo de la imagen 1 de los InstrumentoMusical la clase GuitaraElectrica, tiene que tener un método que redefina la implementación de tocar() que generar *"Esta Guitarra eléctrica con 6 cueras rockea!,* de esta manera un método que estaba definido en una clase abstracta se implementa en una clase de uso y la clase GuitaraElectrica quedaría como en la Imagen 2.

```
public class GuitaraElectrica extends InstrumentoCuerda {

    public GuitaraElectrica() {
        super();
        this.nombre = "Guitarra";
        this.numeroDeCuerdas = 6;
    }

    public void tocar() {
        System.out.println("Esta " + nombre + " electrica con "
        + numeroDeCuerdas + " cuerdas  rockea!");
    }
}
```

Imagen 2. Declaración de un método abstracto
Fuente: Elaboración Propia

1.2. MODULARIDAD EN JAVA

IMPORTANTE

Reflexionemos:

¿Divide y vencerás, aplica esto en Java?

En palabras sencillas, es la división de una aplicación en módulos independientes, estos deben ser compilados por separado sin perder la interconexión con los restantes módulos, esto además permite que el desarrollo sea compartido dado que un módulo puede ser desarrollado por otro programador, tampoco se debe olvidar que es sobre este modularidad que Java se hace fuerte, con sus múltiples librerías que permiten la reutilización de código.

La modularidad no solo facilita el desarrollo, si no que en el caso particular de Java permite que módulos externos se ejecuten, solo al memento de instanciarlos manteniendo al programa principal ligero y rápido.

Un claro ejemplo de modularidad es la impresión, en general cuando se realiza una aplicación, esta proporciona resultados, y, si se partiese desde cero no tendría sentido realizar un código específico para imprimir. Sin embargo, cada vez que se le requiera, realiza un módulo de impresión, el cual define, qué variables de entrada necesita y realiza la función de imprimir independiente de quién le esté llamando, lo cual es lo más práctico.

1.3. ENCAPSULAMIENTO

El encapsulamiento de la las variables y procedimientos busca tener datos cohesionados que no permitan el acceso directo desde otras instancias, se requiere del uso de funciones para acceder a éstos, los ya mencionados accesadores y mutadores, en el caso de métodos, estos también pueden limitar el ámbito en el que serán vistos, de manera que se limita quienes puede acceder a estos.

```
class ClasePrivada
{   //no se puede acceder parametro si no es por un mimbro de la clase
    private int parametro;
    protected void setParametro (int valorParametro){
        //asigna valor a parametro
        this.parametro = valorParametro;
    }
    protected int getParametro (){
        //retorna valor a parametro
        return this.parametro;
    }
}
```

```
public class ClasePublica
{
 public static void main(String[] argv){
   //creamos una nueva innstancia de Clase privada
   ClasePrivada accessoEncapsulado = new ClasePrivada();
   //establecemos el parametro privado (oculto)
   accessoEncapsulado.setParametro(5);
   //desplegamos el valor almacenado
   System.out.println("Por encapsulamiento se le asigno"+
   " al parametro el valor de "+ accessoEncapsulado.getParametro());
    }
}
```

Imagen 3. Ejemplo de encapsulamiento de una clase
Fuente: Elaboración Propia

1.4. HERENCIA

IMPORTANTE

Reflexionemos:

¿Se utiliza el principio de la herencia realmente en programación orientada a objetos?

La herencia de objetos, es la capacidad de una clase (subclase) de heredar de otra sus atributos y métodos de manera jerárquica, es decir, o el que hereda recibe atributos y métodos de una clase superior (superclase).

En programación orientada a objetos, una superclase es una generalización y la relación de la superclase con sus subclases se le conoce como herencia. En el caso particular de Java, la herencia además de la relación entre una abstracción y una clase puede ser entre clases normales a través de la extensión de clase.

1.4.1. EXTENSIÓN DE UNA CLASE

Para extender una clase no es mandatorio que la superclase sea una clase abstracta, siguiendo con el ejemplo de la GuitarraElectrica al declarar la clase se define que esta extiende a otra.

```
public class GuitaraElectrica extends InstrumentoCuerda
```

Imagen 4. Extensión de una clase
Fuente: Elaboración Propia

1.4.2. REPRESENTACIÓN GRÁFICA EN BLUEJ

Ya se observó cómo se crea una clase en *BlueJ,* para este caso es mejor definir la clase *Abstract Classs* luego de seleccionar una nueva clase, asignar un nombre, para seguir con el ejemplo se creará InstrumentoMusical e InstrumentoCuerda como clases de abstracción y GuitaraElectrica como *Class*, una vez creada se selecciona la flecha de herencia, esta es de línea continua, y se une pinchando desde el heredero y sin soltar, se desplaza hasta la superclase y luego se suelta el botón del ratón.

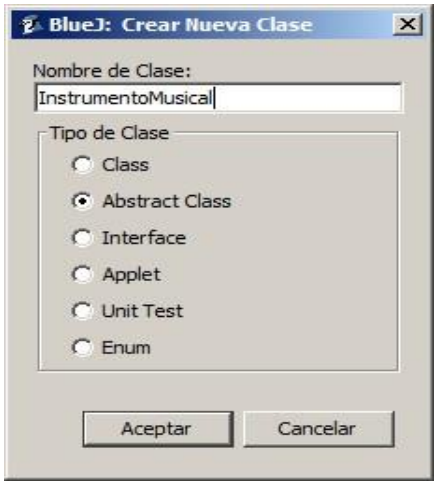

Imagen 5. Creación de Clase Abstracta en BlueJ
Fuente: Elaboración Propia

Se genera la siguiente representación, posteriormente hay que editar cada clase para que tenga el código de ejemplo que ya se mostró, por defecto trae un código que no tiene relevancia con este ejemplo:

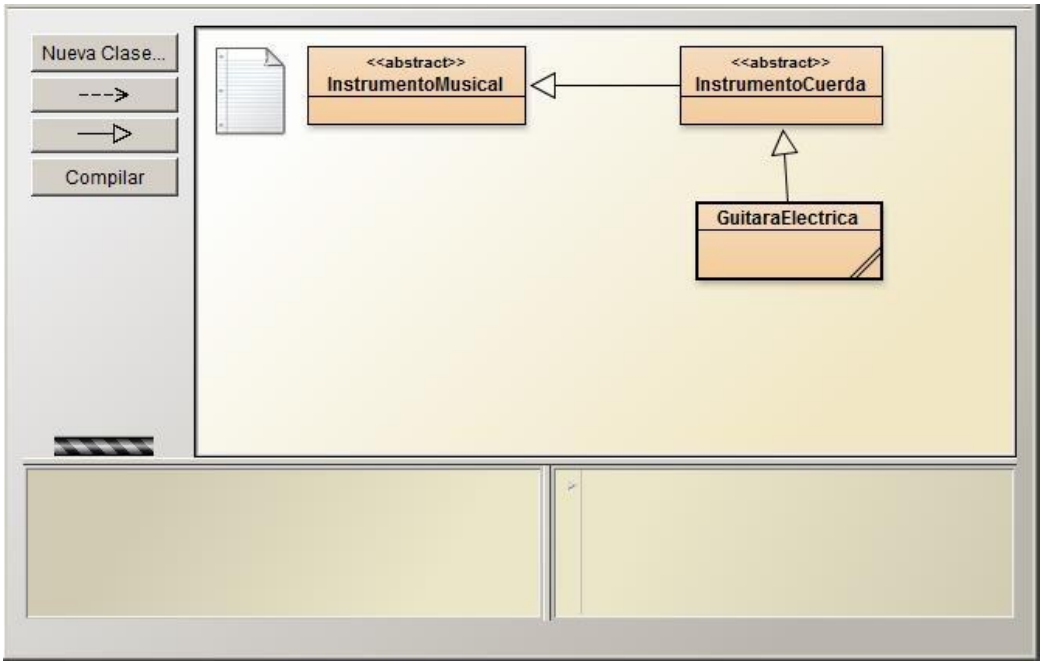

Imagen 6. Representación Gráfica de la Clase Instrumento Musical
Fuente: Elaboración Propia

1.4.3. SINTAXIS DE LA HERENCIA EN JAVA

Dado que lo que se hereda no son solo parámetros, sino que también métodos, la manera de invocar al constructor de la clase sobre la que se extiende es mediante el método *super(),* este puede ser con o sin parámetros, dependiendo del constructor de la clase que se hereda. En el caso de no incluirlo y que la superclase no tenga constructor, el compilador lo hará automáticamente, de todas maneras, es buena práctica incluirlo.

En la imagen 2 el constructor de GuitaraElectrica invoca *super()* para luego inicializar las variables que heredó.

También en la misma imagen se observa como un método que fue redefinido por la clase que heredó, el método *tocar()* venía definido desde InstrumentoMusical , dado que por herencia el método tenía que ser implementado.

1.5. POLIMORFISMO

IMPORTANTE

Reflexionemos:

¿Es posible que un método ejecute múltiples acciones dentro de un programa en JAVA?

Polimorfismo es la capacidad de un método de realizar múltiples acciones dependiendo de cómo se le invoque, o de dónde se le invoque, si la invocación varía en cuanto a parámetros de inicialización se le conoce como polimorfismo por sobrecarga.

Por otro lado, cuando más de una clase hereda el mismo método, se está frente al caso de que la implementación de un método no necesariamente es idéntica para cada subclase y al ser invocada en una subclase diferente, puede realizar funciones distintas. En el caso de la abstracción, sus métodos son genéricos y deben ser redefinidos en una subclase permitiendo el polimorfismo paramétrico.

1.5.1. POLIMORFISMO PARAMÉTRICO:

Tómese el caso simplificado de los InstrumentosMusicales.

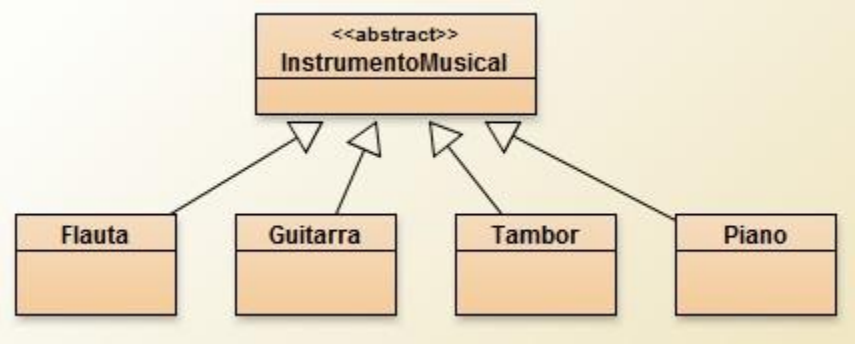

Imagen 7. Representación del Polimorfismo en la Clase Instrumento Musical
Fuente: Elaboración Propia

Si mantiene a "InstrumentoMusical" con todas las subclases Flauta, Guitarra, Tambor y Piano tendrán que implementar el método *tocar()*, pero no todas se tocan igual. El polimorfismo paramétrico o universal es el que redefine en cada instancia, el ¿cómo se implementará un método?, es buena práctica, al usar el comando *@Override* antes de invocar al método de esta manera, se está informando al compilador que el método va ser reescrito, en la siguiente imagen se verá de forma recortada como se implementa el polimorfismo paramétrico, cada subclase redefine de manera independiente que acción realiza, todos los mensajes de salida son independientes y distintos.

1.5.2. POLIMORFISMO POR SOBRECARGA

Este dependerá de los parámetros de entrada, el cómo se ejecute es común en los constructores dado que Java permite múltiples constructores con diferente definición de datos de entrada, de manera tal, que el código realiza múltiples funciones dependiendo de la cantidad y el tipo de datos. Volviendo al ejemplo inicial de la GuitaraElectrica, en la imagen 9 se muestra el polimorfismo por sobrecarga de su constructor.

```
public class GuitaraElectrica extends InstrumentoCuerda {

    public GuitaraElectrica() {
        super();
        this.nombre = "Guitarra";
        this.numeroDeCuerdas = 6;
    }

    public GuitaraElectrica(int numeroDeCuerdas) {
        super();
        this.nombre = "Guitarra";
        this.numeroDeCuerdas = numeroDeCuerdas;
    }
}
```

Imagen 8. Ejemplo de Polimorfismo por Sobrecarga
Fuente: Elaboración Propia

1.6. MÉTODOS

IMPORTANTE

Reflexionemos:

¿Cuál entonces es la finalidad de un método?

Como se ha estudiado en las semanas anteriores, los métodos son las funciones propias de cada clase, si a esto se agrega el concepto de herencia, también, son propios los que se heredan de otra clase las que se pueden redefinir.

1.6.1. DECLARACIÓN DE MÉTODOS

La declaración de un método es el nombre que se le da, la accesibilidad que le figura a través del modificador, el tipo de valor que retorna a menos que sea *void*, los parámetros de entrada y el cuerpo del método.

Modificador VariableDeRetorno NombreDeMetodo (VariablesDeEntrada)

1.6.2. MENSAJES

En programación orientada a objetos, se le conoce como mensaje a la comunicación entre objetos, lo esencial para esto es la correcta invocación de un método de la clase, la acción que el método realice con el mensaje es propia de éste, puede ser una actualización un cambio de variable o devolver un valor, o todas a la vez.

Más específicamente, un mensaje tiene un objeto receptor, un método a llamar y argumentos, por ejemplo, volviendo a la GuitaraElectrica: para mostrar el funcionamiento se crea un objeto de GuitarraElectricav en otra clase la que se llamará TestDeGuitarra y luego le enviará el mensaje de que invoque el método *tocar()* pero de este nuevo objeto.

```
public class TestDeGuitarra
{

    public TestDeGuitarra()
    {
        GuitaraElectrica miGuitarra = new GuitaraElectrica(7);
        miGuitarra.tocar();
    }
}
```

Imagen 10. Ejemplo de Mensaje
Fuente: Elaboración

Al correr esta clase su salida por pantalla será: *"Esta Guitarra eléctrica con 7 cuerdas rockea!"*

1.6.3. PASO DE PARÁMETROS

En Java el paso de parámetros es por valor, es decir, se crea una copia del valor del objeto y sobre esa copia se realizan los cálculos y modificaciones por lo que en ningún momento el original se ve afectado, dependiendo del método, este puede recibir para un campo más de un tipo de parámetro, por ejemplo, puede recibir tanto un entero como un número real.

```
public class PasoDeParametro
{
    public static void main(String[] args)throws Exception
    {
        int primeraVariable = 11; //se le asigna un valor
        int segundaVariable = 22;
        // se le asigna una copia del valor de la primera
        int terceraVariable = primeraVariable;
        primeraVariable = 33;
        System.out.print("Actualmente la primera = "+primeraVariable+
        " , la segunda = "+segundaVariable+" y la tercer = "+terceraVariable);
        //esto imprimira a pantalla
        // Actualmente la primera = 33 , la segunda = 22 y la tercer = 11
    }
}
```

Imagen 11. Paso de Parámetros
Fuente: Elaboración Propia

1.6.4. RETORNO DE VALORES

Está determinado por la definición del método y en algunos casos, que éste este sobrecargado, también dependerá de los valores entregados en la invocación del método.

```
public class RetornoDeValor
{
    private int x; //usaemos una variable de clase
    public RetornoDeValor(){}
    public int metodDePrueba(int y)
    {
        //este metodo retorna un entero
        return x + y;
    }
    public String metodDePrueba(int y, int z)
    {
        //este metodo retorna un texto a diferencia de del ejemplo anterior
        int temp = x + y+ z;
        return "Este metodo retorna: "+Integer.toString(temp);
    }
    public static void main(String[] args)throws Exception
        {
        RetornoDeValor test = new RetornoDeValor ();
        test.x = 3;
        test.metodDePrueba ( 12);
        System.out.println("Con una sola variable retorno: "+ test.metodDePrueba (12));
        //deplegara : Con una sola variable retorno: 15
        System.out.println(test.metodDePrueba (12,5));
        //deplegara : Este metodo retorna: 20
    }
}
```

Imagen 12. Retorno de Variables
Fuente: Elaboración Propia

En el ejemplo se ve, como el MetodDePrueba existe dos veces en la clase y el valor de retorno varía según la cantidad de parámetros con los que se invoque, si es uno devuelve un entero y si es dos devuelve una cadena de caracteres, esto grafica cómo el valor de retorno varía según la definición del método.

1.7. CONTRUCTORES Y DESTRUCTORES

Constructores son procedimientos especiales que no retornan valor y poseen el mismo nombre de la clase. Puede existir más de uno, de no ser declarados se generará uno por defecto que es equivalente al constructor sin ningún código. Dado que Java es una plataforma con recolector de basura, el uso de destructores no está propiamente implementado, este se encarga de validar cuando las variables no son necesarias y las retira, liberando memoria. Pese a eso Java tiene un procedimiento *finalize()* que es lo más cercano a un destructor, el cual no puede ser sobrecargado, no tiene ni devuelve argumentos y su definición es opcional dado que el compilador lo implementa, por uno en caso de no definirlo.

```
public class Automovil
{
 public String placaPatene,  fabricante, modelo, color;
 public int anno, numeroPuerta;
 public Automovil(){}
 public Automovil
 (String placaPatene, String fabricante,String modelo,int anno,int numeroPuerta,String color){
     this.placaPatene = placaPatene; //this hace referencia al objeto
     this.fabricante = fabricante;
     this.modelo = modelo;
     this.anno = anno;
     this.numeroPuerta = numeroPuerta;
     this.color = color;
    }
 static void imprimeAuto(Automovil imprimir){
     System.out.println("Auto fabricado por "+ imprimir.fabricante+ " modelo "+
     imprimir.modelo + " Placa "+ imprimir.placaPatene+" del año "+ imprimir.anno
     +"  color "+imprimir.color +" y de"+ imprimir.numeroPuerta +" puertas " );
     }
 public static void main(String[] args)throws Exception
    {
     Automovil miAuto = new Automovil("ZB9999""Ferrari","testarossa",2015,3,"Rojo Italiano");
     Automovil unoDesconocido = new Automovil();
     unoDesconocido.placaPatene = "AA66666";
     unoDesconocido.fabricante = "Fiat";
     unoDesconocido.modelo = "Palio";
     unoDesconocido.anno = 1989;
     unoDesconocido.numeroPuerta=3;
     unoDesconocido.color="gris";
     imprimeAuto (miAuto);
     imprimeAuto (unoDesconocido);
    }
}
```

Imagen 13. Constructor de ejemplo: Automóvil
Fuente: Elaboración Propia

En el ejemplo, se ve como automóvil tiene dos constructores, el por defecto que obliga manualmente a asignar cada uno de los valores del objeto, como se ve en el método *main* y un constructor que le asigna los valores a cada una de las variables de la clase automóvil, esto es muy útil y facilita la lectura del código.

IMPORTANTE

El objetivo de un constructor, es el de inicializar un objeto cuando éste es creado. Se asignarán los valores iniciales, así como los procesos que esta clase deba realizar. Se utiliza para crear tablas de métodos virtuales y poder así desarrollar el polimorfismo, una de las herramientas de la programación orientada a objetos (POO). Al utilizar un constructor, el compilador determina cuál de los objetos va a responder al mensaje (virtual) que hemos creado. Tiene un tipo de acceso, un nombre y un paréntesis.

1.8. SOBRECARGA DE MÉTODOS

IMPORTANTE

Reflexionemos:

Luego de tener claridad en la importancia de los métodos en JAVA ¿Cuándo debemos utilizar la sobrecarga?

Es la creación de múltiples métodos que realizan acciones distintas dependiendo de la cantidad y el tipo de parámetros que se le entreguen, para esto tiene que existir en una clase, más de una copia de un método que realice cosas distintas para entradas distintas, lo que une a estos métodos es un nombre en común.

En el siguiente ejemplo se aprecia cómo se declara un método sobrecargado que no es un constructor, el caso de desplegar si el primer dato es carácter lo imprimirá directo, si los parámetros son un carácter y un entero, imprimirá el carácter un espacio y el número.

```
public class SobreCargaDeMetodo
{
   public void desplegar(char c)
    {
         System.out.println(c);
    }
    public void desplegar(char c, int numero)
    {
         System.out.println(c + " "+numero);
    }
}

class EjemploDeSobreCarga
{
   public static void main(String args[])
   {
       SobreCargaDeMetodo objeto = new SobreCargaDeMetodo();
       objeto.desplegar('a');
       objeto.desplegar('a',10);
   }
}
```

Imagen 14. Ejemplo de Sobrecarga
Fuente: Elaboración Propia

IMPORTANTE

La sobrecarga de métodos es la creación de varios métodos con el mismo nombre, pero con diferente lista de tipos de parámetros. Java, utiliza el número y tipo de parámetros para seleccionar cuál definición de método ejecutar. Java diferencia los métodos sobrecargados con base en el número y tipo de parámetros o argumentos que tiene el método y no por el tipo que devuelve. También existe la sobrecarga de constructores: Cuando en una clase existen constructores múltiples, se dice que hay sobrecarga de constructores.

```
class EjemploDeVisbilidad
{
    public void desplegar(char c)
    {
        System.out.println(c);
    }
    protected void desplegar(char c, int numero)
    {
        System.out.println(c + " "+numero);
    }
    private void desplegar( int numero,char c)
    {
        System.out.println(c + " "+numero);
    }
}

public class TestDeVisibilidad
{
    public static void main(String args[])
    {
        EjemploDeVisbilidad objeto = new EjemploDeVisbilidad();
        objeto.desplegar('a');
        objeto.desplegar('a',10);
        objeto.desplegar(10,'a');
    }
}
```

Imagen 15. Ejemplo de sobrecarga de métodos
Fuente: Elaboración Propia

1.9. MODIFICADORES DE VISIBILIDAD

Si bien ya se ha planteado el concepto de modificador, lo relevante ahora es su interacción con la herencia.

De esta manera, si al método de una clase no se le asigna ningún modificador esta podrá ser heredada por cualquier subclase que este dentro del mismo *package,* y, accedida por cualquier clase dentro del mismo *package,* pero esto sin ser ilegal, es una falta de adhesión a los estándares de calidad. Se encuentran modificadores:

- Privado: con el modificador *private* no podrá ser ni heredada ni accedida por ninguna clase o subclase, dicho esto, no se puede especificar un método abstracto privado, no solo por lo obvio que se declara un método que nadie lo va implementar, sino que también genera un error de compilación, además al ser invocado desde otra clase como en la Imagen 12 (la última línea de código) genera un error de compilación.

- Público: si el modificador es *public* cualquier clase puede heredarla y accederla.
- Protegido: en el caso que se le asigne el modificador *protected* solo podrá ser heredada por las subclases de su clase y accedida por cualquier clase dentro del mismo *package.*

COMENTARIO FINAL

Se han estudiado los aspectos más relevantes de la orientación a objetos que propone Java, la herencia de clases a través de la extensión de clases abstractas, la generación de clases abstractas, el uso de métodos abstractos y cómo estos se implementan en polimorfismo, cómo estos se representan en BlueJ y cómo se implementan constructores que son heredados de clases abstractas.

Se profundizó, en los métodos sus declaraciones y la comunicación entre objetos a través de mensajes, los parámetros y el retorno de valores.

Se revisó, como es el polimorfismo paramétrico y por sobrecarga, en el primer caso se estudió qué significa sobrescribir un método para que cada clase pueda implementar el polimorfismo paramétrico.

Del polimorfismo por sobrecarga se observó un ejemplo en que se invocó un constructor de clase, y, otro en que se analizó la sobrecarga de métodos simples, para concluir con los modificadores de visibilidad, tanto de clases abstractas como de clases simples y ¿qué pasa al designar un modificador a un método y luego invocarlo desde otra clase?

REFERENCIAS

Dean, J. & Dean R. (2009). Introducción a la Programación en Java. Editorial Mc Graw Hill. México.

Joyanes, L. (2011). Programación en Java 6. Editorial Mc Graw Hill. México

PARA REFERENCIAR ESTE DOCUMENTO, CONSIDERE:

IACC (2020). *Características de la POO*. Programación Orientada a Objeto I. Semana 5.

iacc