PROGRAMACION ORIENTADA A OBJETOS

CLASE

Una clase es una plantilla o prototipo que define las variables y los métodos comunes a todos los objetos de cierto tipo. Las clases definen estado(variables) y comportamiento (métodos) de todos los objetos.
Las clases son el mecanismo por el que se pueden crear nuevos Tipos en Java. Las clases son el punto central sobre el que giran la mayoría de los conceptos de la Orientación a Objetos.
Una clase es una agrupación de datos y de código (métodos) que actúa sobre esos datos, a la que se le da un nombre.

Una clase contiene:
Datos (se denominan Datos Miembro). Estos pueden ser de tipos primitivos o referencias.
Métodos (se denominan Métodos Miembro).
La sintaxis general para la declaración de una clase es:

```
modificadores class nombre_clase {
      declaraciones_de_miembros ;
}
```

Los modificadores son palabras clave que afectan al comportamiento de la clase.
Por ejemplo crearemos la clase Rectangulo cuyos atributos son base y altura, ademas queremos calcular el area, perímetro y diagonal del Rectangulo

```
import java.io.*;
class Rectangulo{
   private double base;
   private double altura;
   public Rectangulo(double b, double h)   // Constructor
   {
     base = b;
     altura=h;
   }
   public void setBase(double b)
   {
         base=b;
   }
   public void setAltura(double h)
  {
         altura=h;
   }
   public double getBase()
  {
         return base;
  }
  public double getAltura()
 {
         return altura;
  }
```

```
	public double area()
	{
		return base*altura;
	}

	public double perimetro()
	{
		return 2*base+2*altura;
	}

	public double diagonal()
	{
		return Math.sqrt(Math.pow(base,2)+Math.pow(altura,2));
	}

	public String toString()
	{
		return "base = "+base+"  "+altura;
	}
}
```

La clase Rectángulo tiene 2 atributos base y altura los cuales son privados esto quiere decir que estas 2 variables son visibles en la clase Rectángulo.
El primer método que se ha implementado es el ***constructor*** , este método se llama igual que la clase y no devuelve ningún valor y permite inicializar los atributos de la clase. Este método se llama en el momento de crear un objeto.
Como los atributos base y altura son privados, para que los usuarios que usan los objetos puedan modificar los atributos se crean los métodos setBase(double b) y setAltura(double h). Y si deseamos obtener los valores de los atributos creamos los métodos getBase() y getAltura().
Además se han creado los métodos area(), perímetro() y diagonal() que permiten calcular el area, perímetro y diagonal del rectangulo.
En el método toString() (a cadena) se crea una cadena con la información de los atributos de la clase. En realidad podemos colocar cualquier información.

```
public class pruebaRectangulo{
   public static void main(String args[]) throws IOException
  {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        double b, h;
        Rectangulo R;
        System.out.print("Ingrese base : ");
        b=Double.parseDouble(br.readLine());
        System.out.print("Ingrese altura : ");
        h=Double.parseDouble(br.readLine());
        R = new Rectangulo(b,h);
        System.out.println("Rectangulo : "+R);
        System.out.println("Area : "+R.area());
        System.out.println(“Perimetro : “+R.perimetro());
        System.out.println("Diagonal : "+R.diagonal());
```

```
  }
}
```
Dentro del metodo main de la clase PruebaRectangulo se ha declarado dos variables de tipo primitivo b,h y una variable R que es de tipo Rectangulo.
Al colocar :
Rectangulo R;
Se esta declarando a R como un Objeto de la Clase Rectangulo.
La declaración no crea nuevos objetos. En la declaración (Rectangulo R) se declara una variable llamada R la cual será usada para referirnos a un Objeto Rectangulo. La referencia esta vacía. Una referencia vacía es conocida como referencia nula.

Al colocar :
R = new Rectangulo(3,4);
Con el operador new creamos un objeto de la clase Rectangulo.El operador new instancia una clase asignando memoria para el nuevo Objeto.
El operador new requiere una llamada a un constructor de la clase a instanciar. El constructor inicializa el nuevo objeto.El operador new retorna una referencia al objeto creado.
Una vez creado el objeto para poder llamar a sus metodos usamos lo siguiente objeto.nombredeMétodo. Por ejemplo para calcular el area usamos R.area(), para calcular el perímetro R.perimetro() y para calcular la diagonal R.diagonal().
Al escribir System.out.println("Rectangulo : "+R); en realidad se esta llamando tácitamente al método toString de la clase R.
Si se deseara modificar el atributo base del Objeto se debe usar el método setBase por ejemplo si después de crear el objeto queremos que base tenga el valor 10, se colocaria la siguiente instrucción : R.setBase(10); lo mismo se hace si se quiere modificar la altura.
Si se desea saber el valor de algún atributo del objeto se usa los métodos get, por ejemplo si quisiera imprimir el valor de la base del objeto R se tendria que escribir lo siguiente :

```
System.out.println("La base es : "+R.getBase());
```

Ejercicios

1) Crear la clase Cilindro con atributos radio y altura y que se pueda calcular el area y el volumen del cilindro.
2) Crear la clase numeros que tenga como atributos dos numeros y se calcule su suma, resta, multiplicación, división.
3) Crear la clase Alumno que tenga como atributos nombre, nota1 y nota2 y permita calcular el promedio y su condicion (aprobado o desaprobado)
4) Crear la clase Trabajador que tenga como atributos nombre, preciHora y horasTrabajadas y se calcule salario Bruto, impuestos(10% del Salario Bruto) y salario Neto (Salario Bruto – Impuestos)
5) Crear la clase Movil con atributos velocidad Inicial, aceleración y tiempo y se pueda calcular el espacio recorrido por el móvil

1) Crear la clase Cilindro con atributos radio y altura y que se pueda calcular el area y el volumen del cilindro.

```
import java.io.*;

class Cilindro{
   private double radio;
   private double altura;

   public Cilindro(double r, double a)
   {
                radio=r;
                altura=a;
     }

     public void setRadio(double r)
     {
                radio=r;
     }

     public void setAltura(double a)
     {
                altura=a;
     }

     public double getRadio()
     {
                return radio;
     }

     public double getAltura()
     {
                return altura;
     }

     public double area()
     {
                return 2*Math.PI*Math.pow(radio,2)+2*Math.PI*radio*altura;
     }

     public double volumen()
     {
                return Math.PI*Math.pow(radio,2)*altura;
     }

     public String toString()
     {
                return "Radio = "+radio+" Altura = "+altura;
     }
 }

public class PruebaCilindro
{
     public static void main(String args[]) throws IOException
     {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                double r, h;
                Cilindro C;
                System.out.print("Ingrese radio: ");
```

```
                r=Double.parseDouble(br.readLine());
                System.out.print("Ingrese altura : ");
                h=Double.parseDouble(br.readLine());
                C = new Cilindro(r,h);
                System.out.println("Cilindro : "+C);
                System.out.println("Area : "+C.area());
                System.out.println("Volumen : "+C.volumen());
        }
    }
```

2) Crear la clase numeros que tenga como atributos dos numeros y se calcule su suma, resta, multiplicación, división.

```
import java.io.*;

class Numeros{
    private double numero1;
    private double numero2;
    public Numeros(double n1,double n2)
    {
      numero1=n1;
      numero2=n2;
    }

    public void setNumero1(double n1)
    {
      numero1=n1;
    }

    public void setNumero2(double n2)
    {
                numero2=n2;
        }

        public double getNumero1()
        {
                return numero1;
        }

        public double getNumero2()
        {
                return numero2;
        }

        public double suma()
        {
                return numero1+numero2;
        }

        public double resta()
        {
                return numero1-numero2;
        }

        public double multiplicacion()
        {
                return numero1*numero2;
        }

        public double division()
```

```
	{
		return numero1/numero2;
	}

	public String toString()
	{
		return "numero1 = "+numero1+" numero2 = "+numero2;
	}
}

public class PruebaNumeros
{
	public static void main(String args[]) throws IOException
	{
		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
		double n1,n2;
		Numeros A;
		System.out.print("Ingrese primero Numero : ");
		n1=Double.parseDouble(br.readLine());
		System.out.print("Ingrese segundo Numero: ");
		n2=Double.parseDouble(br.readLine());
		A = new Numeros(n1,n2);
		System.out.println("Numeros : "+A);
		System.out.println("suma : "+A.suma());
		System.out.println("resta : "+A.resta());
		System.out.println("Multiplicacion : "+A.multiplicacion());
		System.out.println("Division : "+A.division());
	}
}
```

3) Crear la clase Alumno que tenga como atributos nombre, nota1 y nota2 y permita calcular el promedio y su condicion (aprobado o desaprobado)

```
import java.io.*;

class Alumno{
    private String nombre;
    private double nota1;
    private double nota2;

    public Alumno(String nom, double n1, double n2)
    {
		nombre=nom;
		nota1=n1;
		nota2=n2;
    }

    public void setNombre(String nom)
    {
		nombre=nom;
	}

	public void setNota1(double n1)
	{
		nota1=n1;
	}

	public void setNota2(double n2)
	{
		nota2=n2;
```

```
	}

	public String getNombre()
	{
		return nombre;
	}

	public double getNota1()
	{
		return nota1;
	}

	public double getNota2()
	{
		return nota2;
	}

	public double promedio()
	{
		return (nota1+nota2)/2;
	}

	public String condicion()
	{
		if(promedio()>=10.5)
		    return "aprobado";
		else
		    return "desaprobado";
	}

	public String toString()
	{
		return "nombre : "+nombre +"nota1 = "+nota1+" nota2 = "+nota2;
	}
}

public class PruebaAlumno{
	public static void main(String args[]) throws IOException
	{
		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
		String nom;
		double n1,n2;
		Alumno A;
		System.out.print("Ingrese nombre : ");
		nom= br.readLine();
		System.out.print("Ingrese nota1 : ");
		n1=Double.parseDouble(br.readLine());
		System.out.print("Ingrese nota2 : ");
		n2=Double.parseDouble(br.readLine());
		A = new Alumno(nom,n1,n2);
		System.out.println("Alumno : "+A);
		System.out.println("Promedio : "+A.promedio());
		System.out.println("Condicion : "+A.condicion());
	}
}
```

4) Crear la clase Trabajador que tenga como atributos nombre, preciHora y horasTrabajadas y se calcule salario Bruto, impuestos(10% del Salario Bruto) y salario Neto (Salario Bruto – Impuestos)

```
import java.io.*;

class Trabajador{
    private String nombre;
    private double horasTrabajadas;
    private double precioHora;

    public Trabajador(String nom, double ht, double ph)
    {
                    nombre=nom;
                    horasTrabajadas=ht;
                    precioHora=ph;
    }

    public void setNombre(String nom)
    {
                    nombre=nom;
            }

            public void setHorasTrabajadas(double ht)
            {
                    horasTrabajadas=ht;
            }

            public void setPrecioHora(double ph)
            {
                    precioHora=ph;
            }

            public String getNombre()
            {
                    return nombre;
            }

            public double getHorasTrabajadas()
            {
                    return horasTrabajadas;
            }

            public double getPrecioHora()
            {
                    return precioHora;
            }

            public double salarioBruto()
            {
                    return precioHora*horasTrabajadas;
            }

            public double impuestos()
            {
                    return 0.10*salarioBruto();
            }

            public double salarioNeto()
            {
```

```
                return salarioBruto()-impuestos();
        }

        public String toString()
        {
            return "nombre : "+nombre+ " Horas Trabajadas : "+horasTrabajadas+" Precio Hora : "+precioHora;
        }
}


public class PruebaTrabajador
{
        public static void main(String args[]) throws IOException
        {
                BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                String nom;
                double ph,ht;
                Trabajador T;
                System.out.print("Ingrese nombre : ");
                nom= br.readLine();
                System.out.print("Ingrese numero de horas Trabajadas : ");
                ht=Double.parseDouble(br.readLine());
                System.out.print("Ingrese precio de la Hora : ");
                ph=Double.parseDouble(br.readLine());
                T = new Trabajador(nom,ht,ph);
                System.out.println("Trabajador : "+T);
                System.out.println("Salario Bruto : "+T.salarioBruto());
                System.out.println("Impuestos : "+T.impuestos());
                System.out.println("Salario Neto  : "+T.salarioNeto());
        }
}
```

5) Crear la clase Móvil con atributos velocidad Inicial, aceleración y tiempo y se pueda calcular el espacio recorrido por el móvil

```
import java.io.*;

class Movil{
    private double velocidadInicial;
    private double aceleracion;
    private double tiempo;

    public Movil(double vi, double a, double t)
    {
      velocidadInicial=vi;
      aceleracion=a;
      tiempo=t;
    }

    public void setVelocidadInicial(double vi)
    {
                velocidadInicial=vi;
        }

        public void setAceleracion(double a)
        {
                aceleracion=a;
        }

        public void setTiempo(double t)
```

```
	{
		tiempo=t;
	}

	public double getVelocidadInicial()
	{
		return velocidadInicial;
	}

	public double getAceleracion()
	{
		return aceleracion;
	}

	public double getTiempo()
	{
		return tiempo;
	}

	public String toString()
	{
		return "Velocidad Inicial = "+velocidadInicial+" Aceleracion = "+aceleracion+"Tiempo
= "+tiempo;
	}

	public double espacioRecorrido()
	{
		return velocidadInicial*tiempo+(1.0/2.0)*aceleracion*Math.pow(tiempo,2);
	}
}

public class PruebaMovil
{
	public static void main(String args[]) throws IOException
	{
		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
		double vi,a,t;
		Movil M;
		System.out.print("Ingrese velocidad Inicial : ");
		vi=Double.parseDouble(br.readLine());
		System.out.print("Ingrese aceleracion  : ");
		a=Double.parseDouble(br.readLine());
		System.out.print("Ingrese tiempo : ");
		t=Double.parseDouble(br.readLine());
		M = new Movil(vi,a,t);
		System.out.println("Movil : "+M);
		System.out.println("Espacio Recorrido : "+M.espacioRecorrido());
	}
}
```

Controlando el acceso a los miembros de una clase

Private

El nivel de acceso más restrictivo es private. Un miembro private es accesible solo en la clase en la cual es definida. Se debe usar este acceso para declarar miembros que solamente deben ser usados en la clase.

Para declarar un miembro privado, se usa la palabra private en su declaración. La siguiente clase contiene una variable miembro privada y un método privado.

```
class Alpha {
    private int x;
    private void privateMethod() {
        System.out.println("privateMethod");
    }
}
```

Protected

Permite que las clases, subclases y todas las clases del mismo paquete puedan acceder a sus miembros.

Public

Cualquier clase en cualquier paquete tienen acceso a miembros publicos de las clases.
Ejm:

```
package Greek;

public class Alpha {
    public int iampublic;
    public void publicMethod() {
        System.out.println("publicMethod");
    }
}

import Greek.*;

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampublic = 10;       // legal
        a.publicMethod();       // legal
    }
}
```

Package

El nivel de acceso package es el que se obtiene si no se coloca explícitamente otros niveles de acceso. Este nivel de acceso permite a las clases en el mismo paquete como su clases acceder a los miembros. Este nivel de acceso asume que clases en el mismo paquete son amigas de confianza.
Ejemplo:

```
package Greek;

class Alpha {
    int iampackage;
    void packageMethod() {
        System.out.println("packageMethod");
    }
}
```

```
En la clase Alpha iampackage y packageMethodo tienen acceso nivel de
acceso package. Todas las clases declaradas dentro del mismo paquete
como Alpha también tienen acceso a iampackage y packageMethod.
```

Supongan que Alpha y Beta fueron declaradas como parte del paquete Greek Package

```
package Greek;

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampackage = 10;     // legal
        a.packageMethod();     // legal
    }
}
```

```
Beta puede legalmente acceder a iampackage y packageMethod como se
muestra.
```

La referencia this

Cada objeto tiene acceso a una referencia a si mismo, llamada referencia *this*.
La referencia *this* se usa explícitamente para referirse tanto a los atributos como a los metodos de un objeto.
Ejemplo:

```
class Numero{
     private int x;
     public Numero(int x)
     {
        this.x = x;
     }
    public void setX(int x)
    {
       this.x = x;
    }
    public String toString()
   {
      Return “x = “+this.x;
    }
}
```

Por ejemplo en el constructor el nombre del Parametro es x y el nombre del atributo de la clase es x. Para que Java no se confunda se utiliza this.x para referirse al atributo de la clase.

Miembros de clase Estaticos

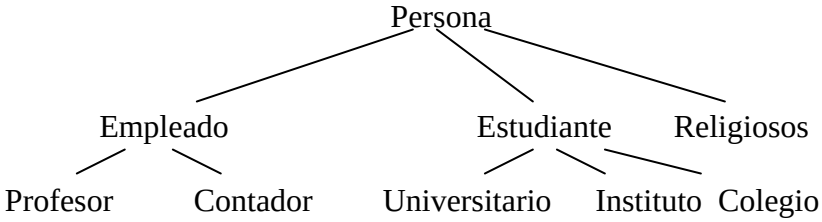
Cada objeto de una clase tiene su propia copia de todas las variables de ejemplar de clase. En ciertos casos, una sola copia de la variable en particular debe ser compartida por todos los objetos de la clase. Por esta y otras razones utilizamos las variables de clase static (estáticas). Una variable de clase static representa información “que abarca toda la clase”. La declaración de un método estático comienza con la palabra clave **static**.
Ejemplo:

```
import java.io.*;

class Empleado{
    private String nombres;
    private String apellidos;
    private static int contador;

    public Empleado(String nom, String ape)
    {
      nombres=nom;
      apellidos=ape;
      contador++;
    }

    public void finalize()
    {
     --contador;
    }

    public void setNombres(String nom)
            {
                    nombres=nom;
            }

            public void setApellidos(String ape)
            {
                    apellidos=ape;
            }

            public static void setContador(int cont)
            {
               contador=cont;
            }

    public String getNombres()
    {
                    return nombres;
            }

            public String getApellidos()
            {
                    return apellidos;
            }

            public static int getContador()
            {
                    return contador;
            }

            public String toString()
            {
                    return apellidos+" "+nombres;
            }
}

public class pruebaEmpleadoVariableEstatica{
          public static void main(String args[]) throws IOException
          {
                  System.out.println(“Numero de objetos creados : “+Empleado.getContador());
                  Empleado e1= new Empleado("Torres","Fidel");
                  System.out.println(e1);
                  System.out.println("Número de objetos creados : "+e1.getContador());
                  Empleado e2= new Empleado("Villanueva","Nelsa");
                  System.out.println(e2);
                  System.out.println("Número de objetos creados : "+Empleado.getContador());
          }
}
```

En el programa anterior usamos un atributo private static y un método public static.
El atributo contador se inicializa en cero por omisión. Esta variable va contando el numero de Objetos de la Clase Empleado que se van creando, esta variable se va incrementando en el constructor cada vez que se crea un objeto.
Para saber cuantos objetos hemos creados llamamos al método estatico getContador() que devuelve el valor de la variable contador.
Un método declarado static no puede acceder a miembros de clase no estáticos. Un método static no tiene referencia this porque las variables de clase static y los métodos static existen independientemente de que existan o no objetos de clase.
En la primera linea del Programa colocamos llamamos Empleado.getContador() pues como es un método extático no es necesario usar un objeto de la clase par llamar al método, solo se usa el nombre de la Clase y luego el método getContador().

Herencia

A través de la herencia, una clase nueva hereda los atributos y métodos de una superclase previamente definida. En este caso decimos que la nueva clase es una subclase.
En la herencia simple, una clase se hereda de una superclase. Java no reconoce la herencia múltiple, pero si maneja el concepto de interfaces. Las interfaces ayudan a java a lograr muchas de las ventajas de la herencia múltiple.
Una subclase normalmente agrega sus propios atributos y métodos. De modo que una subclase generalmente es mayor que su superclase. Una subclase es más específica que su superclase y representa un grupo más pequeño de objetos. El verdadero valor de la herencia radica en la capacidad de definir en la subclase adiciones a las características heredadas de la superclase o sustituciones de estas.
Todo objeto de una subclase es también un objeto de la superclase de esa subclase. Sin embargo no se cumple lo opuesto: los objetos de una superclase no son objetos de las subclases de esa superclase.
Las relaciones de herencia forman estructuras jerárquicas similares a un árbol. Una superclase existe en una relación Jerárquica con sus subclases. Sin duda, una clase puede existir sola, pero es cuando una clase se emplea con el mecanismo de herencia que se convierte ya sea en una superclase que proporciona atributos y comportamientos a otras clases, o en una subclase que hereda dichos atributos y comportamientos.
Ejemplos:



Por ejemplo en una ciudad existen Personas las cuales pueden ser Empleados, Estudiantes, Religiosos. Los Empleados pueden ser Profesor, Contador. Los estudiantes pueden ser Universitarios, de Institutos y Colegios.

Miembros Protected

Los miembros protected de una superclase sólo están accesibles para los métodos de la superclase, los métodos de las subclases y los métodos de otras clases del mismo paquete.

Relacion entre Objetos de superclase y objetos de subclase

Un objeto de una subclase se puede tratar como objeto de su superclase correspondiente. Esto hace posible ciertas manipulaciones interesantes. Por ejemplo, a pesar del hecho de que los objetos de diversas clases derivadas de una superclase en particular pueden ser muy diferentes entre sí, podemos crear un arreglo de ellos, en tanto los tratemos como objetos de la superclase. Lo contrario no se cumple un objeto de una superclase no es automáticamente también un objeto de la subclase.

```
import java.io.*;

class Punto{
  private double x;
  private double y;

  public Punto()
  {
            x=0;
            y=0;
  }

  public Punto(double a, double b)
  {
    x=a;
    y=b;
  }

  public void setX(double a)
  {
    x=a;
  }

  public void setY(double b)
  {
    y=b;
  }

  public double getX()
  {
   return x;
  }

  public double getY()
  {
   return y;
  }

  public String toString()
  {
            return "("+x+","+y+")";

  }
}


class Circulo extends Punto{
          private double radio;

          public Circulo()
          {
```

```
                super();
                radio=0;
        }

        public Circulo(double a, double b, double r)
        {
                super(a,b);
                radio=r;
        }

        public void setRadio(double a)
        {
                radio=a;
        }

        public double getRadio()
        {
                return radio;
        }

        public double area()
        {
                return Math.PI*Math.pow(radio,2);
        }

        public String toString()
        {
                return "Centro = "+super.toString()+" Radio = "+radio;
        }

}

public class HerenciaPuntoCirculo{
        public static void main(String args[]) throws IOException
        {
    Punto P= new Punto(3,4);
    Circulo C=new Circulo(6,9,12);
    System.out.println(P);
    System.out.println(C);

    // Como circulo hereda de Punto se puede hacer
    // la asignacion P=C pero no a la inversa.

    P=C;
    System.out.println("Circulo via Punto"+P);
    // Si se trabaja con P para calcular el area primero debemos
    // Convertir P a tipo Circulo.
    System.out.println("Area  del Circulo : "+((Circulo)P).area());

        }
}
```

La clase Circulo hereda de la clase Punto esto se especifica en:

```
class Circulo extends Punto{
```

La palabra clave extends (extiende) de la definición de clase indica herencia. Todos los miembros (no private) de la clase Punto se heredan en la clase Circulo.

Los constructores de Circulo deben invocar al constructor de Punto para inicializar la porción de superclase de un objeto de la clase Circulo. La primer línea del cuerpo de cada constructor invoca al constructor de Punto mediante la referencia a super.
Una subclase pude redefinir un método de superclase empleando el mismo nombre, esto se denomina supeditar un método de superclase. Se puede usar la referencia super seguida por el operador punto para acceder a la versión de superclase de ese método desde la subclase.
En el ejemplo anterior la clase Circulo supedita el método toString() de la clase Punto.

Conversión implicita de objeto de SubClase a objeto de SuperClase
Una referencia a un objeto de subclase puede convertirse implícitamente en una referencia a un objeto de superclase porque un objeto de subclase es un objeto de superclase gracias a la herencia

Composición frente a herencia
La relación ***es un*** se implementa mediante la herencia. La relación ***tiene un*** en la que una clase puede tener objetos de otras clases como miembros; tales relaciones crean clases nuevas mediante la composición de clases ya existentes

Clases Abstractas

Las clases abstractas suelen representar conceptos generales, las características comunes de una serie de objetos.
Ejemplo:
Persona es una clase abstracta en un contexto de trabajadores de una empresa. De igual forma Figura es una clase abstracta, siendo Rectangulo, Circulo, clases concretas.
De una clase abstracta no se obtienen elementos activos, no tiene objetos.
En Java el modificador abstract declara una clase abstracta:

```
abstract class NombreClase{ //….}
```

Ejemplo:

```
public abstract class Persona{
        private String dni;
        private String apellidos;
        private String nombres;
        …..
}
```

Las clases abstractas pueden definir métodos no abstractos y atributos, y normalmente métodos abstractos. El Lenguaje java obliga a que si una clase tiene un método abstracto la clase se declara abstracta.
Una característica importante es que no se pueden crear objetos de una clase abstracta. Pero si se pueden declarar variables de una clase abstracta. Como estas están en lo más alto de la Jerarquía de clases, son superclases base, hay una conversión automática de referencias de clases derivadas a clases base. Así, se puede asignar a una variable de clase base abstracta cualquier objeto concreto de las clases derivadas.
Una clase derivada que no redefine un método abstracto es también clase abstracta.

Por ejemplo, las siguientes instrucciones definen una clase abstracta llamada InsectosVoladores:

```
public abstract class InsectosVoladores{
    public abstract int volar(float velocidad);
    // Aquí pueden ir otros métodos
}

public class Abeja extends InsectosVoladores{
    private String nombreEspecie;
    public  int volar(float velocidad)
   {
       // Aquí se especifica como hacer volar una abeja
    }
}
```

La clase abstracta tiene un método sin implementación. Al crear la clase Abeja , se debe implementar el método. Como se ve, las clases abstractas en cierta forma obligan a crear y utilizar los métodos que definen.

Polimorfismo

Mediante el polimorfismo, se pueden escribir programas que procesen genéricamente – como objetos de superclase- objetos de todas las clases existentes en una jerarquía. Las clases que no existen durante el desarrollo de los programas se pueden agregar con poca o ninguna modificación de la parte genérica del programa, en tanto esas clases formen parte de la jerarquía que se esta procesando genéricamente.

Ejemplos:

```
import java.io.*;

abstract class Empleado{
    protected String apellidos;
    protected String nombres;

    public Empleado(String ape, String nom)
    {
                apellidos=ape;
                nombres=nom;
        }

        public void setApellidos(String ape)
        {
                apellidos=ape;
        }

        public void setNombres(String nom)
        {
                nombres = nom;
        }

        public String getApellidos()
        {
                return apellidos;
        }
```

```
    public String getNombres()
    {
        return nombres;
    }

    abstract double ganancias();
}


final class Jefe extends Empleado{
    public double salario;

    public Jefe(String ape, String nom,double s)
    {
        super(ape,nom);
        salario=s;
    }

    public void setSalario(double s)
    {
        salario=s;
    }

    public double getSalario()
    {
        return salario;
    }

    public double ganancias()
    {
        return salario;
    }

    public String toString()
    {
        return "Jefe : "+apellidos+"  "+nombres;
    }
}

final class EmpleadoPorComision extends Empleado
{
    private double salarioBase;  // salario Base
    private double comisionPorArticulo; // comision por articulo vendido
    private int cantidadDeArticulos;   // cantidad de articulos vendidos

    public EmpleadoPorComision(String ape, String nom,double sb, double com, int cant)
    {
        super(ape,nom);
        salarioBase=sb;
        comisionPorArticulo=com;
        cantidadDeArticulos=cant;
    }

    public void setSalarioBase(double sb)
    {
        salarioBase=sb;
    }

    public void setComisionPorArticulo(double com)
```

```
	{
		comisionPorArticulo=com;
	}

	public void setCantidadDeArticulos(int cant)
	{
		cantidadDeArticulos=cant;
	}

	public double getSalarioBase()
	{
		return salarioBase;
	}

	public double getComisionPorArticulo()
	{
		return comisionPorArticulo;
	}

	public int getCantidad()
	{
		return cantidadDeArticulos;
	}

	public String toString()
	{
		return "Empleado por Comision : "+apellidos+"  "+nombres;
	}

	public double ganancias()
	{
		return salarioBase+comisionPorArticulo*cantidadDeArticulos;
	}
}

final class EmpleadoADestajo extends Empleado{
	private double salarioPorPieza;
	private int cantidad;

	public EmpleadoADestajo(String ape, String nom,double sp, int cant)
	{
		super(ape,nom);
		salarioPorPieza=sp;
		cantidad=cant;
	}

	public void setSalarioPorPieza(double sp)
	{
		salarioPorPieza = sp;
	}

	public void setCantidad(int cant)
	{
		cantidad=cant;
	}

	public double getSalarioPorPieza()
	{
		return salarioPorPieza;
	}
```

```
	public double getCantidad()
	{
		return cantidad;
	}

	public double ganancias()
	{
		return salarioPorPieza*cantidad;
	}

	public String toString()
	{
		return "Empleado a Destajo : "+apellidos+"  "+nombres;
	}
}

final class EmpleadoPorHora extends Empleado
{
	protected double salarioPorHora;
	protected double horasTrabajadas;

	public EmpleadoPorHora(String ape, String nom, double sh, double ht)
	{
		super(ape,nom);
		salarioPorHora= sh;
		horasTrabajadas=ht;
	}

	public void setSalarioPorHora(double sh)
	{
		salarioPorHora=sh;
	}

	public void setHorasTrabajadas(double ht)
	{
		horasTrabajadas=ht;
	}

	public double getSalarioPorHora()
	{
		return salarioPorHora;
	}

	public double getHorasTrabajadas()
	{
		return horasTrabajadas;
	}

	public String toString()
	{
		return "Empleado por Hora : "+apellidos+"  "+nombres;
	}

	public double ganancias()
	{
		return salarioPorHora*horasTrabajadas;
	}

}
```

```
public class PruebaEmpleado{
	public static void main(String args[])
	{
		Empleado E;
		Jefe J=new Jefe("Torres","Marcelino",2500);
		EmpleadoPorComision C=new EmpleadoPorComision("Zavaleta","Juan",300,4,200);
		EmpleadoPorHora H = new EmpleadoPorHora("Narvaez","Robinson",10,40);
		EmpleadoADestajo D = new EmpleadoADestajo("Marin","Alejandro",20,5);

		E = J;
		System.out.println(E.toString()+" gano "+E.ganancias());

		E = C;
		System.out.println(E.toString()+" gano "+E.ganancias());

		E = H;
		System.out.println(E.toString()+" gano "+E.ganancias());

		E = D;
		System.out.println(E.toString()+" gano "+E.ganancias());

	}
}

/*
```

La Linea
E = J;

Coloca en la referencia de la Superclase E una referencia al objeto J de la subclase Jefe. Esto es precisamente lo que debemos hacer para lograr un comportamienteo Polimorfico.

La expresion :

E.toString()

Innvoca al metodo toString() del objeto al que E hace referencia.
El sistema invoca al método toString() del objeto de la subclase, precisamente lo que se llama comportamiento polimorfico. Esta llamada de metodo es un ejemplo de ligado dinamico
de métodos; la decision respecto a cual método invocar se aplaza hasta el momento de ejecucion.

La llamada al Metodo

E.ganancias()

invoca al metodo ganancias del objeto al que E hace referencia. El sistema invoca el metodo
ganancias del objeto de la subclase en lugar del metodo ganancias de la superclase. Esto es otro ejemplo de ligado dinamico de metodos.

```
import java.io.*;

abstract class Figura{
   public double area()
   {
     return 0;
   }
   public double volumen()
   {
     return 0;
   }
   public abstract String getNombre();
}

class Punto extends Figura{
        protected double x;
        protected double y;

        public Punto(double a, double b)
        {
                x=a;
                y=b;
        }

        public void setX(double x)
        {
                this.x=x;
        }

        public void setY(double y)
        {
                this.y=y;
        }

        public double getX()
        {
                return x;
        }

        public double getY()
        {
                return y;
        }

        public String toString()
        {
                return "("+x+","+y+")";
        }

        public String getNombre()
        {
                return "Punto";

        }

}

class Circulo extends Punto{
        protected double radio;
```

```
	public Circulo(double a, double b, double r)
	{
		super(a,b);
		radio=r;
	}

	public void setRadio(double r)
	{
		radio=r;
	}

	public double getRadio()
	{
		return radio;
	}

	public double area()
	{
		return Math.PI*Math.pow(radio,2);
	}

	public String toString()
	{
		return "Centro "+ super.toString()+", Radio = "+radio;
	}

	public String getNombre()
	{
		return "Circulo";
	}
}

class Cilindro extends Circulo{
	protected double altura;

	public Cilindro(double a,double b, double r, double h)
	{
		super(a,b,r);
		altura=h;
	}

	public void setAltura(double h)
	{
		altura=h;
	}

	public double getAltura()
	{
		return altura;
	}

	public double area()
	{
		return 2*super.area()+2*Math.PI*radio*altura;
	}


	public double volumen()
	{
```

```
		return super.area()*altura;
	}

	public String toString()
	{
		return super.toString()+ "; Altura = "+altura;
	}

	public String getNombre()
	{
		return "Cilindro";
	}
}

public class PruebaFigura{

	 public static void main(String args[]) throws IOException
	 {
		Punto p = new Punto(3,4);
		Circulo c = new Circulo(12,20,10);
		Cilindro k = new Cilindro (100,100,50,25);
		Figura F[] = new Figura[3];

		F[0]=p;
		F[1]=c;
		F[2]=k;

		System.out.println(p.getNombre()+" : "+p);
		System.out.println(c.getNombre()+" : "+c);
		System.out.println(k.getNombre()+" : "+k);


		// Procesamos ahora el arreglo de Figuras e imprimimos el nombre, area y volumen
		// de cada objeto

		for(int i=0;i<3;i++)
		{
			System.out.println();
			System.out.println(F[i].getNombre()+" : "+F[i]);
			System.out.println("Area ="+F[i].area());
			System.out.println("Volumen"+F[i].volumen());
			System.out.println("==========");

		}


	 }
}
```

INTERFACES

Concepto de Interface

El concepto de interface lleva un paso más adelante la idea de las clases abstractas. En Java una interface es una clase abstracta pura, es decir una clase donde todos los métodos son abstractos (no se implementa ninguno). Permite al diseñador de clases establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno, pero no bloques de código). Una interface puede también contener datos

miembro, pero estos son siempre static y final. Una interface sirve para establecer un 'protocolo' entre clases.

Para crear una interface, se utiliza la palabra clave interface en lugar de class. La interface puede definirse public o sin modificador de acceso, y tiene el mismo significado que para las clases. Todos los métodos que declara una interface son siempre public.

Para indicar que una clase implementa los métodos de una interface se utiliza la palabra clave *implements*. El compilador se encargará de verificar que la clase efectivamente declare e implemente todos los métodos de la interface. Una clase puede implementar más de una interface.

Una interface se declara:

```
interface nombre_interface {
   tipo_retorno nombre_metodo ( lista_argumentos ) ;
   . . .
}
```

Por ejemplo:

```
interface InstrumentoMusical {
   void tocar();
   void afinar();
   String tipoInstrumento();
}
```

Y una clase que implementa la interface:

```
class InstrumentoViento extends Object implements InstrumentoMusical {
   void tocar() { . . . };
   void afinar() { . . .};
   String tipoInstrumento() {}
}

class Guitarra extends InstrumentoViento {
   String tipoInstrumento() {
      return "Guitarra";
   }
}
```

La clase InstrumentoViento implementa la interface, declarando los métodos y escribiendo el código correspondiente. Una clase derivada puede también redefinir si es necesario alguno de los métodos de la interface.

Referencias a Interfaces

Es posible crear referencias a interfaces, pero las interfaces no pueden ser instanciadas. Una referencia a una interface puede ser asignada a cualquier objeto que implemente la interface. Por ejemplo:

```
InstrumentoMusical instrumento = new Guitarra();
instrumento.play();
System.out.prinln(instrumento.tipoInstrumento());

InstrumentoMusical i2 = new InstrumentoMusical(); //error.No se puede instanciar
```

Extensión de interfaces

Las interfaces pueden extender otras interfaces y, a diferencia de las clases, una interface puede extender más de una interface. La sintaxis es:

```
interface nombre_interface  extends nombre_interface  , . . . {
   tipo_retorno nombre_metodo ( lista_argumentos ) ;
   . . .
}
```

Agrupaciones de constantes

Dado que, por definición, todos los datos miembros que se definen en una interface son static y final, y dado que las interfaces no pueden instanciarse resultan una buena herramienta para implantar grupos de constantes. Por ejemplo:

```
public interface Meses {
   int ENERO = 1 , FEBRERO = 2 . . . ;
   String [] NOMBRES_MESES = { " " , "Enero" , "Febrero" , . . . };
}
```

Esto puede usarse simplemente:

```
System.out.println(Meses.NOMBRES_MESES[ENERO]);
```

Un ejemplo casi real

El ejemplo mostrado a continuación es una simplificación de como funciona realmente la gestión de eventos en el sistema gráfico de usuario soportado por el API de Java (AWT o swing). Se han cambiado los nombres y se ha simplificado para mostrar un caso real en que el uso de interfaces resuelve un problema concreto.

Supongamos que tenemos una clase que representa un botón de acción en un entorno gráfico de usuario (el típico botón de confirmación de una acción o de cancelación). Esta clase pertenecerá a una amplia jerarquía de clases y tendrá mecanismos complejos de definición y uso que no son objeto del ejemplo. Sin embargo podríamos pensar que la clase Boton tiene miembros como los siguientes.

```
class Boton extends . . . {
   protected int x , y, ancho, alto; // posicion del boton
   protected String texto;  // texto del boton
   Boton(. . .) {
      . . .
   }
   void dibujar() { . . .}
   public void asignarTexto(String t) { . . .}
```

```
   public String obtenerTexto() { . . .)
   . . .
}
```

Lo que aquí nos interesa es ver lo que sucede cuando el usuario, utilizando el ratón pulsa sobre el botón. Supongamos que la clase Boton tiene un método, de nombre por ejemplo click(), que es invocado por el gestor de ventanas cuando ha detectado que el usuario ha pulsado el botón del ratón sobre él. El botón deberá realizar alguna acción como dibujarse en posición 'pulsado' (si tiene efectos de tres dimensiones) y además, probablemente, querrá informar a alguien de que se ha producido la acción del usuario. Es en este mecanismo de 'notificación' donde entra el concepto de interface. Para ello definimos una interface Oyente de la siguiente forma:

```
interface Oyente {
   void botonPulsado(Boton b);
}
```

La interface define un único método botonPulsado. La idea es que este método sea invocado por la clase Boton cuando el usuario pulse el botón. Para que esto sea posible en algún momento hay que notificar al Boton quien es el Oyente que debe ser notificado. La clase Boton quedaría:

```
class Boton extends . . . {
   . . .
   private Oyente oyente;
   void registrarOyente(Oyente o) {
      oyente = o;
   }
   void click() {
      . . .
      oyente.botonPulsado(this);
   }
}
```

El método registrarOyente sirve para que alguien pueda 'apuntarse' como receptor de las acciones del usuario. Obsérvese que existe una referencia de tipo Oyente. A Boton no le importa que clase va a recibir su notificación. Simplemente le importa que implante la interface Oyente para poder invocar el método botonPulsado. En el método click se invoca este método. En el ejemplo se le pasa como parámetro una referencia al propio objeto Boton. En la realidad lo que se pasa es un objeto 'Evento' con información detallada de lo que ha ocurrido.

Con todo esto la clase que utiliza este mecanismo podría tener el siguiente aspecto:

```
class miAplicacion extends . . . implements Oyente {
   public static main(String [] args) {
      new miAplicacion(. . .);
      . . .
   }
    . . .
```

```
    miAplicacion(. . .) {
        . . .
        Boton b = new Boton(. . .);
        b.registrarOyente(this);
    }

    . . .
    void botonPulsado(Boton x) {
        // procesar click
        . . .
    }
}
```

Obsérvese en el método registrarOyente que se pasa la referencia thisque en el lado de la clase Boton es recogido como una referencia a la interface Oyente. Esto es posible porque la clase miAplicacion implementa la interface Oyente . En términos clásicos de herencia miAplicacion ES un Oyente .

Paquetes

Claúsula package

Un package es una agrupación de clases afines. Equivale al concepto de librería existente en otros lenguajes o sistemas. Una clase puede definirse como perteneciente a un package y puede usar otras clases definidas en ese o en otros packages.
Los packages delimitan el espacio de nombres (space name). El nombre de una clase debe ser único dentro del package donde se define. Dos clases con el mismo nombre en dos packages distintos pueden coexistir e incluso pueden ser usadas en el mismo programa.

Una clase se declara perteneciente a un package con la clausula package, cuya sintaxis es:
package nombre_package;

La clausula package debe ser la primera sentencia del archivo fuente. Cualquier clase declarada en ese archivo pertenece al package indicado.
Por ejemplo, un archivo que contenga las sentencias:

```
package miPackage;
. . .
class miClase {
. . .
```

declara que la clase miClase pertenece al package miPackage.

La claúsula package es opcional. Si no se utiliza, las clases declaradas en el archivo fuente no pertenecen a ningún package concreto, sino que pertenecen a un package por defecto sin nombre.

La agrupación de clases en packages es conveniente desde el punto de vista organizativo, para mantener bajo una ubicación común clases relacionadas que cooperan desde algún punto de vista. También resulta importante por la implicación que los packages tienen en los modificadores de acceso, que se explican en un capítulo posterior.

Claúsula import

Cuando se referencia cualquier clase dentro de otra se asume, si no se indica otra cosa, que ésta otra está declarada en el mismo package. Por ejemplo:

```
package Geometria;
. . .
class Circulo {
   Punto centro;
   . . .
}
```

En esta declaración definimos la clase Circulo perteneciente al package Geometria. Esta clase usa la clase Punto. El compilador y la JVM asumen que Punto pertenece también al package Geometria, y tal como está hecha la definición, para que la clase Punto sea accesible (conocida) por el compilador, es necesario que esté definida en el mismo package.

Si esto no es así, es necesario hacer accesible el espacio de nombres donde está definida la clase Punto a nuestra nueva clase. Esto se hace con la clausula import. Supongamos que la clase Punto estuviera definida de esta forma:

```
package GeometriaBase;
class Punto {
   int x , y;
}
```

Entonces, para usar la clase Punto en nuestra clase Circulo deberiamos poner:

```
package GeometriaAmpliada;

import GeometriaBase.*;

class Circulo {
   Punto centro;
   . . .
}
```

Con la claúsula import GeometriaBase.*; se hacen accesibles todos los nombres (todas las clases) declaradas en el package GeometriaBase. Si sólo se quisiera tener accesible la clase Punto se podría declarar: import GeometriaBase.Punto;

También es posible hacer accesibles los nombres de un package sin usar la clausula import calificando completamente los nombres de aquellas clases pertenecientes a otros packages. Por ejemplo:

```
package GeometriaAmpliada;

class Circulo {
   GeometriaBase.Punto centro;
   . . .
}
```

Sin embargo si no se usa import es necesario especificar el nombre del package cada vez que se usa el nombre Punto.

La claúsula import simplemente indica al compilador donde debe buscar clases adicionales, cuando no pueda encontrarlas en el package actual y delimita los espacios de nombres y modificadores de acceso. Sin embargo, no tiene la implicación de 'importar' o copiar código fuente u objeto alguno. En una clase puede haber tantas sentencias import como sean necesarias. Las cláusulas import se colocan después de la cláusula package (si es que existe) y antes de las definiciones de las clases.

Nombres de los packages

Los packages se pueden nombrar usando nombres compuestos separados por puntos, de forma similar a como se componen las direcciones URL de Internet. Por ejemplo se puede tener un package de nombre misPackages.Geometria.Base. Cuando se utiliza esta estructura se habla de packages y subpackages. En el ejemplo misPackages es el Package base, Geometria es un subpackage de misPackages y Base es un subpackage de Geometria.

De esta forma se pueden tener los packages ordenados según una jerarquía equivalente a un sistema de archivos jerárquico.

El API de java está estructurado de esta forma, con un primer calificador (java o javax) que indica la base, un segundo calificador (awt, util, swing, etc.) que indica el grupo funcional de clases y opcionalmente subpackages en un tercer nivel, dependiendo de la amplitud del grupo. Cuando se crean packages de usuario no es recomendable usar nombres de packages que empiecen por java o javax.

Ubicación de packages en el sistema de archivos

Además del significado lógico descrito hasta ahora, los packages también tienen un significado físico que sirve para almacenar los módulos ejecutables (ficheros con extensión .class) en el sistema de archivos del ordenador.

Supongamos que definimos una clase de nombre miClase que pertenece a un package de nombre misPackages.Geometria.Base. Cuando la JVM vaya a cargar en memoria miClase buscará el módulo ejecutable (de nombre miClase.class) en un directorio en la ruta de acceso misPackages/Geometria/Base. Está ruta deberá existir y estar accesible a la JVM para que encuentre las clases. En el capítulo siguiente se dan detalles sobre compilación y ejecución de programas usando el compilador y la máquina virtural distribuida por SUN Microsystems (JDK).

Si una clase no pertenece a ningún package (no existe clausula package) se asume que pertenece a un package por defecto sin nombre, y la JVM buscará el archivo .class en el directorio actual.

Para que una clase pueda ser usada fuera del package donde se definió debe ser declarada con el modificador de acceso public, de la siguiente forma:

```
package GeometriaBase;

public class Punto {
   int x , y;
}
```

Si una clase no se declara public sólo puede ser usada por clases que pertenezcan al mismo package