Universidad Nacional Autónoma de México
Dirección General de Cómputo y de Tecnologías de Información y Comunicación
Dirección de Colaboración y Vinculación

Curso
Introducción a la Programación

Orientación a Objetos

Instructor: **Daniel Barajas González**
Correo: **ldanielbg@comunidad.unam.mx**

Presentación del curso

- **Objetivo General**: Identificar y aplicar los conceptos derivados de modelo de programación orientado a objetos.
- **Duración**: 10 horas (**Del 15 – 22 de agosto de 2017. El 18 de agosto no habrá clase**)
- **Horario:** 14:00 a 16:00 horas
- **Evaluación**: 70% tareas y 30% exámenes

Reglas

- **Ser puntuales.** Avisar al instructor con anticipación en caso de inasistencia
  - La clase empieza: **14:00**
  - Primer receso: **14:30 a 14:45 (15 minutos)**
  - Segundo receso: **15:15 a 15:30 (15 minutos)**
- **Cumplir con las asignaciones a tiempo.** Se penalizarán las entregas tardías
- **No comer ni beber en clase** para evitar dañar los equipos de cómputo
- Las **calificaciones** se entregarán una semana después de finalizado el curso.

Temario

1. **Objetos y clasificaciones**
   a. Definición de objeto, clase, instancia, atributos y métodos
   b. Beneficios de la orientación a objetos
   c. Representación de clases con UML
2. **Clases e Instancias**
   a. Definición
   b. Miembros de una clase - *Propiedades y Métodos de instancia y de clase*
   c. Constructores y Destructores
   d. Acceso a los miembros de la clase
   e. Representación de los miembros de una clase con UML
3. **Sobrecarga (*overloading*) y sobre escritura (*overriding*) de métodos**
   a. Definición
   b. Sobrecarga de constructores
   c. Sobrecarga de métodos
   d. Sobre escritura de métodos
4. **Encapsulamiento**
   a. Definición
   b. Visibilidad pública, privada y protegida
   c. Definición de la interfaz pública
5. **Polimorfismo**
   a. Definición
   b. Tipos de polimorfismo
6. **Herencia**
   a. Jerarquía de clases - Clases y sub-clases
   b. Para reutilizar la implementación
   c. Por diferencia de tipos

Universidad Nacional Autónoma de México
Dirección General de Cómputo y de Tecnologías de Información y Comunicación
Dirección de Colaboración y Vinculación

Curso
Introducción a la Programación

Instalación de Herramientas

Instructor: **Daniel Barajas González**
Correo: **ldanielbg@comunidad.unam.mx**

Herramientas de desarrollo

- **Realiza las prácticas #1 y #2 para instalar el JDK y Python 3 en la máquina virtual.**
- **Verifica la instalación Java Development Kit (JDK)**
  - **En una terminal, ejecuta el comando: $ javac –versión**
  - **La salida debe ser la versión del programa.**
  - **Ejemplo:**

```
[patito@localhost ~]$ javac -version
javac 1.8.0_141
```

- **Verifica la instalación de Python 3**
  - **En una terminal, ejecuta el comando: $ python3 –V**

```
[patito@localhost ~]$ python3 -V
Python 3.4.5
```

Universidad Nacional Autónoma de México
Dirección General de Cómputo y de Tecnologías de Información y Comunicación
Dirección de Colaboración y Vinculación

Curso
Introducción a la Programación Orientación a Objetos

Clases, Objetos e Instancias

Instructor: **Daniel Barajas González**
Correo: **ldanielbg@comunidad.unam.mx**

Objetivos

1. Establecer la definición correspondientes a los elementos: **Clase, Objeto e Instancia**

2. Poner en práctica los conceptos utilizando **lenguajes de programación** que soportan el paradigma orientado a objetos

3. Lograr un primer acercamiento a la notación que nos permiten modelar **Clases y Objetos** de manera visual

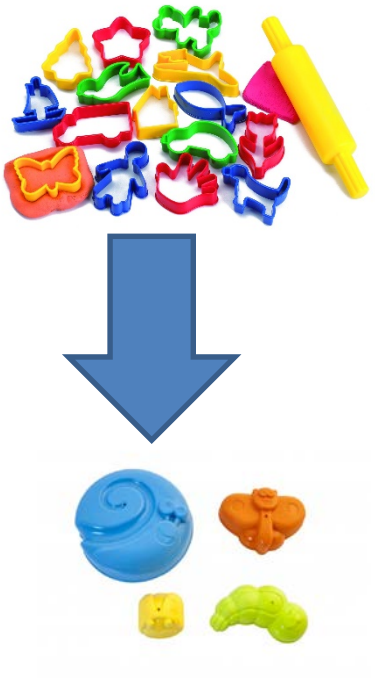
Qué es una clase

- Es la agrupación que le damos a un conjunto de cosas
- Dicha agrupación está basada en las características generales que poseen las cosas de manera conjunta
- En programación orientada a objetos, es la plantilla a partir de la cual vamos a crear objetos. La clase define los atributos y el comportamiento
- La clase no es un componente ejecutable por sí misma

La clase puede ser comprendida como la plantilla para crear objetos

Qué es un objeto

- En términos generales, es la denominación que damos a una cosa cuando no precisamos su definición
- Puede ser algo material, inmaterial, real o imaginario
- Tiene atributos generales y particulares. En ocasiones, los objetos pueden realizar acciones.
- A menudo, los atributos de un objeto pueden ser vistos como objetos independientes: Por ejemplo, un auto y su motor
- En programación orientada a objetos, los objetos tienen atributos y comportamiento.
- Los atributos son representados mediante variables y el comportamiento, mediante funciones.



Los objetos son la materialización de una clase

Atributos

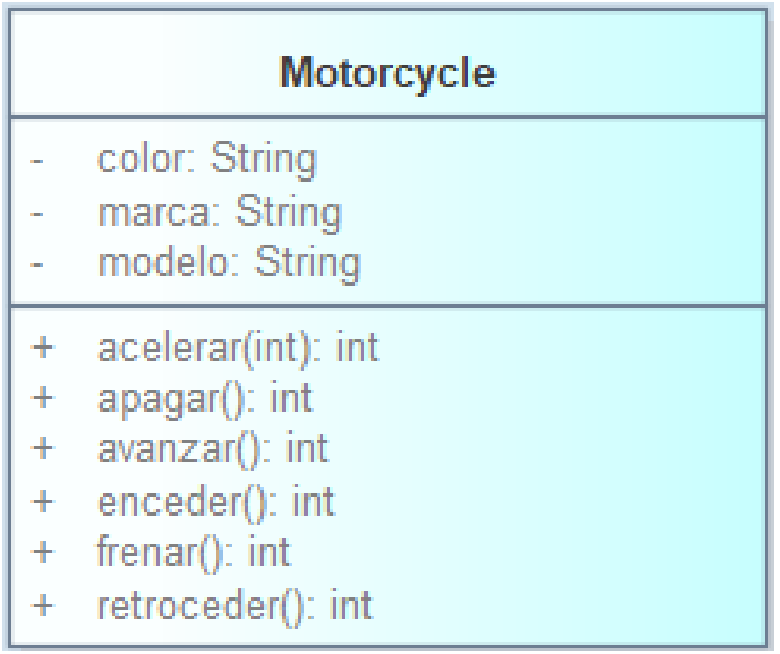
- En la programación orientada a objetos, los atributos se **representan mediante variables**
- Pueden ser de **cualquier tipo** (simples o complejos) incluso pueden ser referencias a otras clases
- Una clase puede tener un gran número de atributos, sin embargo no es recomendable
- Los valores que cada instancia puede tener en sus atributos es independiente de las otras instancias

Variables de clase

- También son representadas mediante variables
- A diferencia de los atributos, las variables de clase son compartidas por todas las instancias
- Todas las instancias tienen acceso a la misma variable y, dependiendo el lenguaje, pueden modificar el valor
- También es posible definir métodos de clase. Por ejemplo, en Java todos los métodos de la clase Math son métodos de clase.

Diagrama de clases en UML

- En UML se representa mediante un cuadrilátero dividido en 3 secciones: nombre, atributos y métodos
- Las secciones para atributos y métodos puede ser opcional
- Puede emplearse para representar incluso tablas de una base de datos



Ejercicio #1

- Modela una clase, con atributos y métodos, los siguientes sustantivos:

1. Un automóvil
2. Una fecha de calendario
3. Un alumno para un sistema de calificaciones
4. Un profesor para un sistema de inscripciones
5. Una cuenta bancaria de ahorro
6. Los helados que vendería una tienda
7. Un perro, un pato, un gato, un pez y un canario

Relaciones entre clases

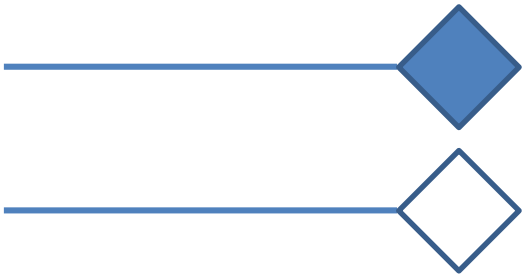
Existen 4 asociaciones básicas para relacionar clases.

1) Asociación / Asociación dirigida

2) Generalización / Especialización

3) Uso / Dependencia

4) Agregación / Composición



Asociación

- Es el tipo de asociación que usarías cuando sabes que dos objetos se relacionan pero aun no estas seguro de la forma en que interactúan

- Puede ser dirigida o no dirigida

Generalización / Especialización

- Indica que existe una relación de herencia entre las dos clases
- La clase donde inicia termina la flecha será la súper clase

Uso o dependencia

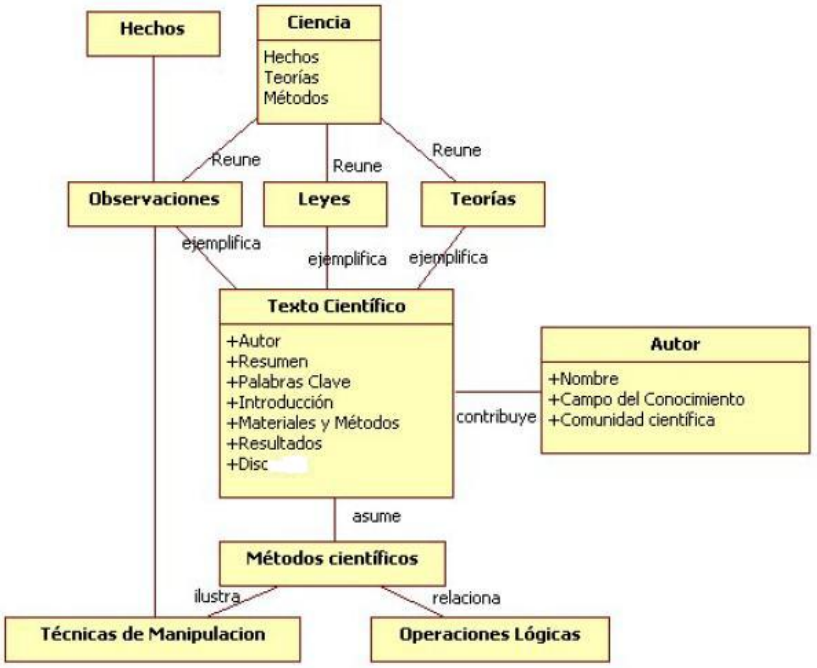
- En este tipo de relación se indica que una clase utiliza o depende de los métodos de otra
- La clase donde termina la flecha es la clase utilizada

Agregación / Composición

- Es una relación en la cual una clase es parte de otra
- En la agregación, las partes pueden existir de manera independiente
- En la composición, las partes no pueden existir sin el objeto compuesto

Modelo de dominio

- Para empezar de modelar un problema mediante el paradigma orientado a objetos, es útil crear un modelo de dominio
- Un modelo de domino representa los objetos de interés para el problema mediante clases así como sus relaciones
- También es posible emplear estereotipos de UML para etiquetar las relaciones entre clases



Ejercicio #2

- Crea un modelo de dominio para los siguientes problemas:

a) Un automóvil puede sufrir averías mecánicas o eléctricas y existe un mecánico especialista para cada una de esas averías

b) Una máquina expendedora de refrescos ofrece cuatro sabores diferentes. La máquina recibe el importe, el comprador selecciona el refresco y, la máquina entrega el producto comprado.

c) Un mapa, puede ser visto como un grafo con nodos y arcos. Cada nodo es un lugar que se puede visitar y cada arco es dirigido. Cada arco tiene un “peso” que representa la distancia entre dos nodos. Dado un origen y un destino, debería ser factible obtener la distancia más corta.

Ejercicio #2 (continúa)

- Crea un modelo de dominio para los siguientes problemas:

a) Un punto bidimensional se conforma de dos coordenadas: x, y. Una recta está conformada por dos puntos bidimensionales. Un círculo tiene un radio y un centro representado por un punto bidimensional. Un triángulo está formado por dos puntos y un cuadrilátero esta formado por la conexión entre 4 puntos. Un polígono puede conformarse de varios puntos. Debe ser posible calcular el perímetro de cada una de las figuras

Universidad Nacional Autónoma de México
Dirección General de Cómputo y de Tecnologías de Información y Comunicación
Dirección de Colaboración y Vinculación

Curso
Introducción a la Programación Orientación a Objetos

Encaspulamiento

Instructor: **Daniel Barajas González**
Correo: **ldanielbg@comunidad.unam.mx**

Objetivos

1. Definir la abstracción y el encapsulamiento como elementos de la programación orientada a objetos

2. Poner en práctica el concepto de encapsulamiento

3. Comprender los conceptos de abstracción, ocultamiento de implementación y división de responsabilidades aplicados a la programación orientada a objetos.

Abstracción

- Es el proceso de simplificar un problema complejo

- Simplificamos un problema al enfocarnos en aspectos relevantes para la solución

- Este proceso facilita la resolución de problemas y contribuye a crear componentes reutilizables.

Encapsulamiento

- Mecanismo de la orientación a objetos que permite dividir un programa en componentes más pequeños e independientes entre sí
- Se logra esta independencia al ocultar los detalles internos de cada componente
- Se trata de generar componentes autónomos
- Hay que ver a los componentes como cajas negras provistas de una interfaz externa

Interfaz externa o pública

- La interfaz externa lista los servicios proporcionados por un componente
- Es un contrato que establece la clase con el mundo exterior
- Al ocultar su implementación, un objeto es libre de cambiarla sin afectar a los objetos que la utilizan

Modificadores de acceso

Los miembros de una clase pueden tener diferentes niveles de acceso. Los más comunes son **public, protected** y **private**, sin embargo, también depende del lenguaje de programación.

- Public. Tienen acceso todos los objetos. Se especifica en UML usando un signo de +
- Protected. Tiene acceso la instancia y cualquier subclase
- Private. Tiene acceso solamente la instancia

Universidad Nacional Autónoma de México
Dirección General de Cómputo y de Tecnologías de Información y Comunicación
Dirección de Colaboración y Vinculación

Curso
Introducción a la Programación Orientada a Objetos

Herencia

Instructor: **Daniel Barajas González**
Correo: **ldanielbg@comunidad.unam.mx**

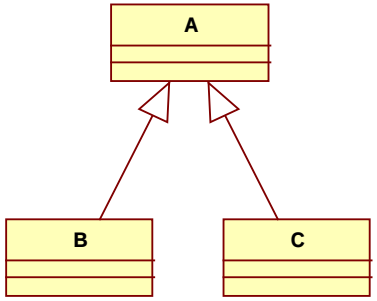
Objetivos

1. Definir la herencia como un elemento de la programación orientada a objetos

2. Poner en práctica el concepto de herencia así como los tipos de herencia existentes

Qué es la herencia

- En el contexto de la programación orientada a objetos, es otra de las **formas en que se relacionan los objetos**
- Es un mecanismo que define una **relación del tipo ancestro-descendiente** entre dos clases de objetos
- Permite definir una **nueva clase** en función de otra existente
- Agrupa **clases que son similares. Regla "*es-un*"**
- Así como en la herencia biológica se comparten genes, en la herencia orientada a objetos **se comparten tipos de dato**



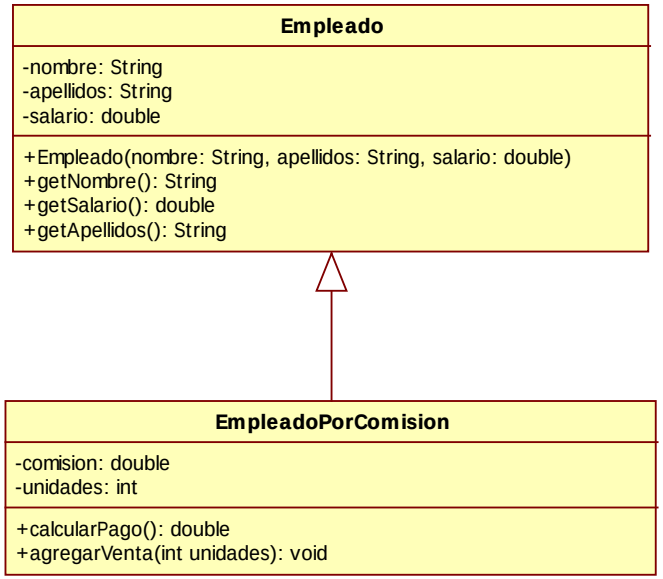
Razones para utilizar la herencia

- Existen 3 razones principales para recurrir a la herencia en un diseño orientado a objetos:

1. **Para reutilizar implementación**
2. **Por diferencia**
3. **Para sustitución de tipos**

Para reutilizar implementación

- La herencia permite a una clase reutilizar la implementación de otra clase
- La regla de oro es aplicar la herencia cuando las clases pasen la prueba “es-un” para mantener relaciones lógicas
- Ejemplo: Un **Empleado** *es un* **Empleado por comisión**



La Clase **EmpleadoPorComision** reutiliza la implementación de su clase base.

La súper clase Empleado

```
package curso;

public class Empleado {
   private String nombre;
   private String apellidos;
   private double salario;

   public Empleado(String nombre, String apellidos, double salario){
       this.nombre = nombre;
       this.apellidos = apellidos;
       this.salario = salario;
   }

   public double getSalario(){
       return this.salario;
   }

   public String getNombre(){
       return this.nombre;
   }

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

La subclase EmpleadoPorComision

```
package curso;

public class EmpleadoPorComision extends Empleado {
    private double comision;
    private int unidades;

    public EmpleadoPorComision(String nombre, String apellidos, double salario, double comision){
        super(nombre, apellidos, salario);
        this.comision = comision;
    }

    public double calcularPago(){
        return getSalario() + (this.comision * this.unidades);
    }

    public void sumarVentas(int unidades){
        this.unidades += unidades;
    }

    public void reiniciarVentas(){
        this.unidades = 0;
    }
}
```

Prueba

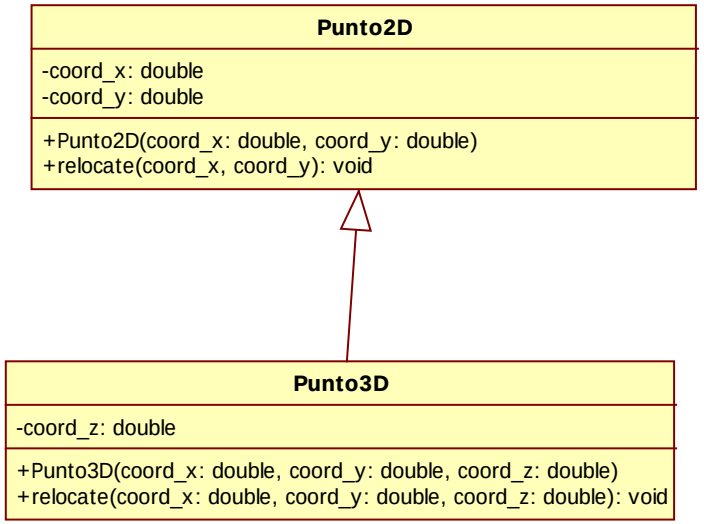
```
package curso;

public class TestEmpleados {

    public static void main(String args[]){
        Empleado empleado = new Empleado("Juan", "Perez", 8000d);
        System.out.println("Empleado            1: " + empleado.getNombre() + " " + empleado.getApellidos());
        System.out.println("Salario Empleado    1: $ " + empleado.getSalario());

        EmpleadoPorComision porComision = new EmpleadoPorComision("Pedro", "Perez", 8000d, 0.10d);
        System.out.println("Empleado            2: " + porComision.getNombre() + " " + porComision.getApellidos());
        porComision.sumarVentas(30);
        System.out.println("Salario Empleado    2: $ " + porComision.getSalario());
        System.out.println("Pago total Empleado 2: $ " + porComision.calcularPago());
    }
}
```

Salida:

```
run:
Empleado            1: Juan Perez
Salario Empleado    1: $ 8000.0
Empleado            2: Pedro Perez
Salario Empleado    2: $ 8000.0
Pago total Empleado 2: $ 8003.0
```

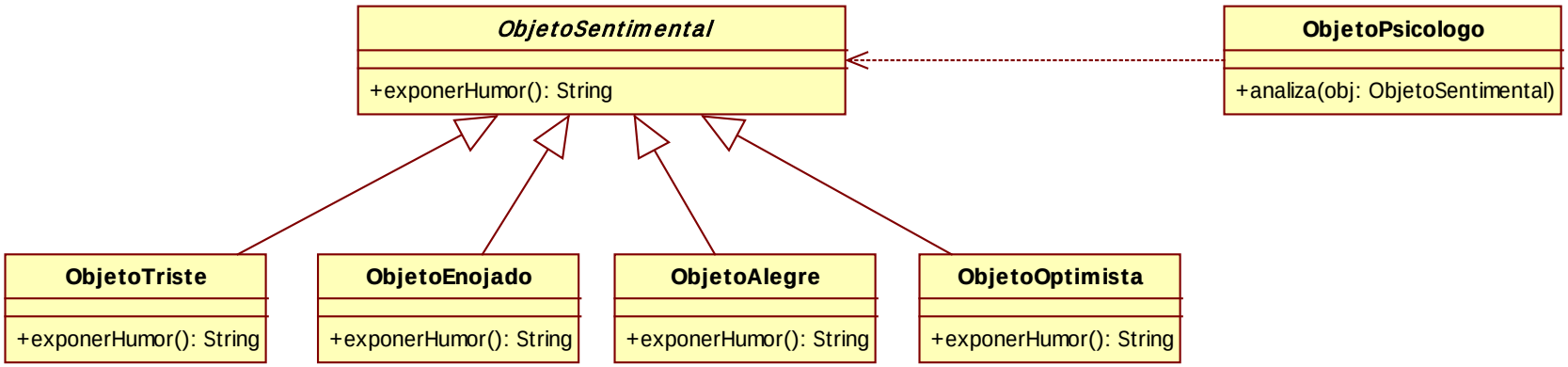
Por diferencia

- Programar por diferencia significa especificar solamente las **diferencias entre las clases relacionadas**
- Consiste en heredar una clase y **agregar atributos nuevos** o bien, **redefinir los comportamientos** que difieren
- A esto también se le conoce como **especialización**



Por sustitución de tipos

- La herencia permite establecer relaciones donde los objetos son reemplazables según su tipo
- Permite extender funcionalidad del software al agregar nuevos tipos sin modificar la funcionalidad existente.



1) La clase base es abstracta

```
package curso;

public abstract class ObjetoSentimental {

    public abstract String exponerHumor();
}
```

- Solamente contiene la declaración del método abstracto que deberán redefinir todas las subclases
- En Java, una clase es abstracta cuando al menos uno de sus métodos es abstracto
- En Java una clase abstracta no puede ser instanciada directamente, es necesario definir e instanciar una sub-clase
- Las clases abstractas sirven para declarar tipos de datos muy generales que sirvan como base para otros

2) La clase base como tipo de dato

```
package curso;

public class ObjetoPsicologo {

    public String analizar(ObjetoSentimental obj){
        return obj.exponerHumor();
    }

}
```

- El **ObjetoPsicologo** declara un método **analizar**
- Dicho método requiere como parámetro un tipo **ObjetoSentimental** como argumento para trabajar
- Debido a que la clase es abstracta, es necesario crear **subclases** de ObjetoSentimental

3) Subclases

```java
package curso;

public class ObjetoTriste extends ObjetoSentimental {
    @Override
    public String exponerHumor() {
        return "Hoy me siento muy triste";
    }

}
```

```java
package curso;

public class ObjetoAlegre extends ObjetoSentimental {

    @Override
    public String exponerHumor() {
        return "Hoy me siento muy alegre!!";
    }

}
```

```java
package curso;

public class ObjetoEnojado extends ObjetoSentimental {

    @Override
    public String exponerHumor() {
        return "Hoy me siento muy enojado!";
    }

}
```

```java
package curso;

public class ObjetoOptimista extends ObjetoSentimental {

    @Override
    public String exponerHumor() {
        return "Siento que hoy todo puede mejorar!";
    }

}
```

- Se crean **subclases** de ObjetoSentimental que **implementan el comportamiento** del método abstracto **exponerHumor()** **adaptado a su propio contexto**

3) Aplicando la sustitución de tipos

El tipo declarado es la súper clase

```
package curso;

public class TestObjetoPsicologo {

    public static void main(String args[]){

        ObjetoPsicologo psicologo = new ObjetoPsicologo();

        ObjetoSentimental objTriste = new ObjetoTriste();
        ObjetoSentimental objEnojado = new ObjetoEnojado();

        System.out.println("Cómo te sientes?");
        System.out.println(psicologo.analizar(objTriste));
        System.out.println("Cómo te sientes?");
        System.out.println(psicologo.analizar(objEnojado));
    }
}
```

Las instancias utilizadas son sub-tipos

Conclusiones

- La herencia es un mecanismo de la programación orientada a objetos que establece una relación de descendencia entre clases
- Permite la reutilización de atributos y métodos así como la redefinición del comportamiento y la adición de atributos
- Aplicamos la herencia cuando se cumple la regla “es-un” entre clases. Así aseguramos que las relaciones son lógicas
- Aunque al crear relaciones de herencia, el código es más pequeño, los objetos en memoria son más grandes por lo que es recomendable crear jerarquías de herencia cortas

Universidad Nacional Autónoma de México
Dirección General de Cómputo y de Tecnologías de Información y Comunicación
Dirección de Colaboración y Vinculación

Curso
Introducción a la Programación Orientada a Objetos

Polimorfismo

Instructor: **Daniel Barajas González**
Correo: **ldanielbg@comunidad.unam.mx**

Objetivos

1. Definir el polimorfismo como un elemento de la programación orientada a objetos

2. Distinguir entre los tipos principales de polimorfismo

3. Poner en práctica los principales tipos de polimorfismo

Qué es el polimorfismo

- De manera general, polimorfismo significa “**muchas formas**”
- En el contexto de la programación orientada a objetos, es un mecanismo que **permite asociar comportamientos distintos a un mismo nombre** de métodos y/o tipos
- Por ejemplo, la acción “abrir” es aplicable a cualquiera de los siguientes objetos, pero cada uno lo realiza de formas distintas:




Tipos de polimorfismo

- Cada lenguaje de programación suele implementar el polimorfismo de diferentes maneras pero existen **4 tipos de polimorfismo** generalmente aceptados:

- **Polimorfismo de inclusión.** Ocurre cuando en una jerarquía de clases, cualquiera de las sub-clases puede ser tratada de forma genérica

- **Polimorfismo paramétrico.** Permite a una clase operar con distintos tipos no relacionados

método()

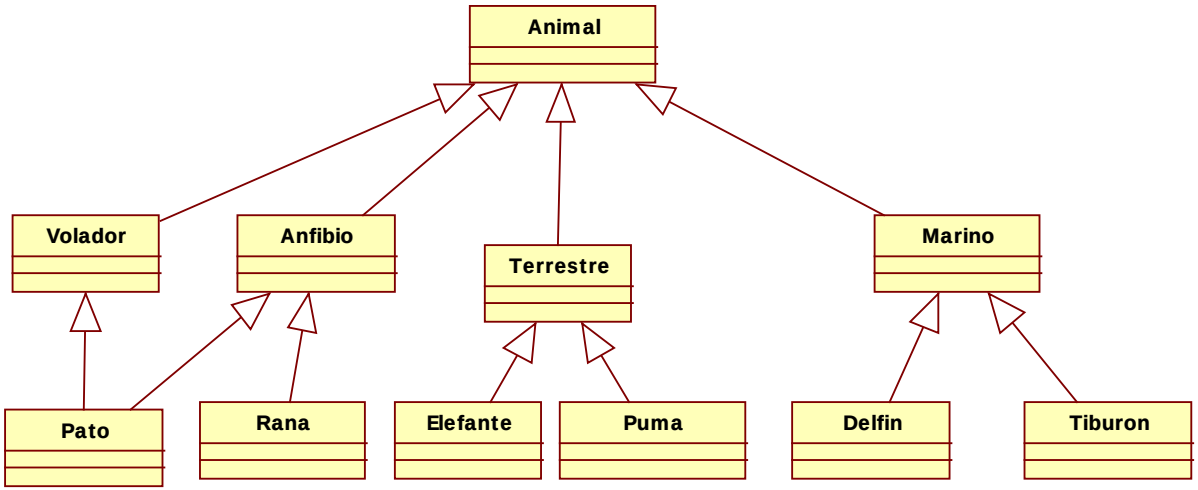
- **Redefinición (*overriding*)** Se presenta en relaciones de herencia cuando una sub-clase establece su propia implementación para un método de su súper-clase

método()

- **Sobrecarga (*overloading*)** También conocido como polimorfismo ad-hoc. Se observa cuando existen varios métodos con el mismo nombre pero los parámetros difieren entre sí, ya sea por tipo o cantidad

Polimorfismo de inclusión

- Dada una jerarquía de herencia, es posible tratar a las subclases de manera genérica
- Una sub-clase es polimórfica pues pertenece a todos los tipos de los cuales hereda



Polimorfismo paramétrico

- Una clase presenta polimorfismo paramétrico cuando puede operar sobre distintos tipos aunque no estén relacionados

- Por ejemplo, Java implementa el mecanismo de tipos genéricos lo cual permite un cierto grado de polimorfismo paramétrico

Ejemplo de polimorfismo paramétrico

- **Problema**: Tenemos 2 clases y necesitamos poder crear listas de ambos e imprimirlas.

```java
class Alumno {

    private String nombre;

    public Alumno(String nombre){
        this.nombre = nombre;
    }

    @Override
    public String toString(){
        return nombre;
    }
}
```

```java
class Libro {

    private String titulo;
    private String autor;

    public Libro(String titulo, String autor) {
        this.titulo = titulo;
        this.autor = autor;
    }

    @Override
    public String toString(){
        return "'"+this.titulo+"' - " + this.autor;
    }
}
```

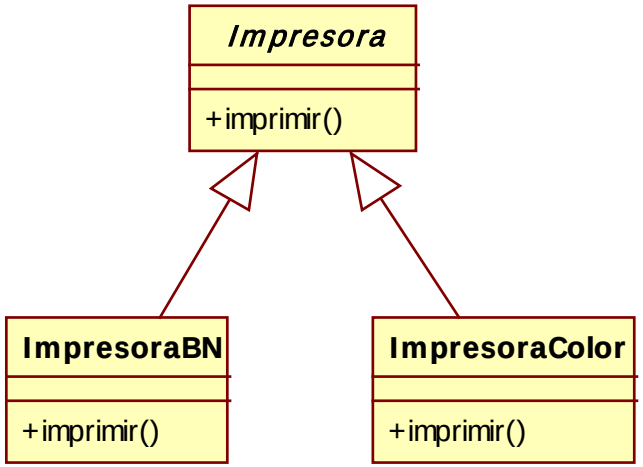
Ejemplo de polimorfismo paramétrico

- Utilizamos el objeto **ArrayList**
- El objeto permite crear instancias que guardan tipos particulares; en este caso Libro y Alumno.
- El objeto es parte del paquete **java.útil**
- El operador <> permite indicar el tipo

```
package curso;

import java.util.ArrayList;

public class ListasObjetos {

    public static void imprimir(ArrayList lista){
        for(Object o : lista){
            System.out.println("Elemento: " + o.toString());
        }
    }

    public static void main(String args[]){

        ArrayList lst = new ArrayList<Alumno>();

        lst.add(new Alumno("Juan Perez"));
        lst.add(new Alumno("Armando Gómez"));

        System.out.println("Listado de Alumnos:");
        ListasObjetos.imprimir(lst);

        ArrayList libs = new ArrayList<Libro>();
        libs.add(new Libro("La Divina Comedia", "Dante Alighieri"));
        libs.add(new Libro("La Odisea", "Homero"));

        System.out.println("Listado de libros:");
        ListasObjetos.imprimir(libs);
    }
}
```

Redefinición (overriding)

- La redefinición o sobre-escritura de métodos se presenta cuando una sub-clase reemplaza el comportamiento de uno de los métodos que hereda



"Imprime en blanco en negro"

"Imprime en color!"

Sobrecarga (overloading)

- Ocurre cuando se tiene más de un método con el **mismo nombre** pero que recibe **diferentes tipos o número de parámetros**

- **La sobrecarga puede ocurrir en una clase o en una subclase**

Canvas
+draw(Line unaLinea) +draw(Polygon unPoligono) +draw(Circle unCirculo)