Bienvenidos

Programacion Orientada a Objetos

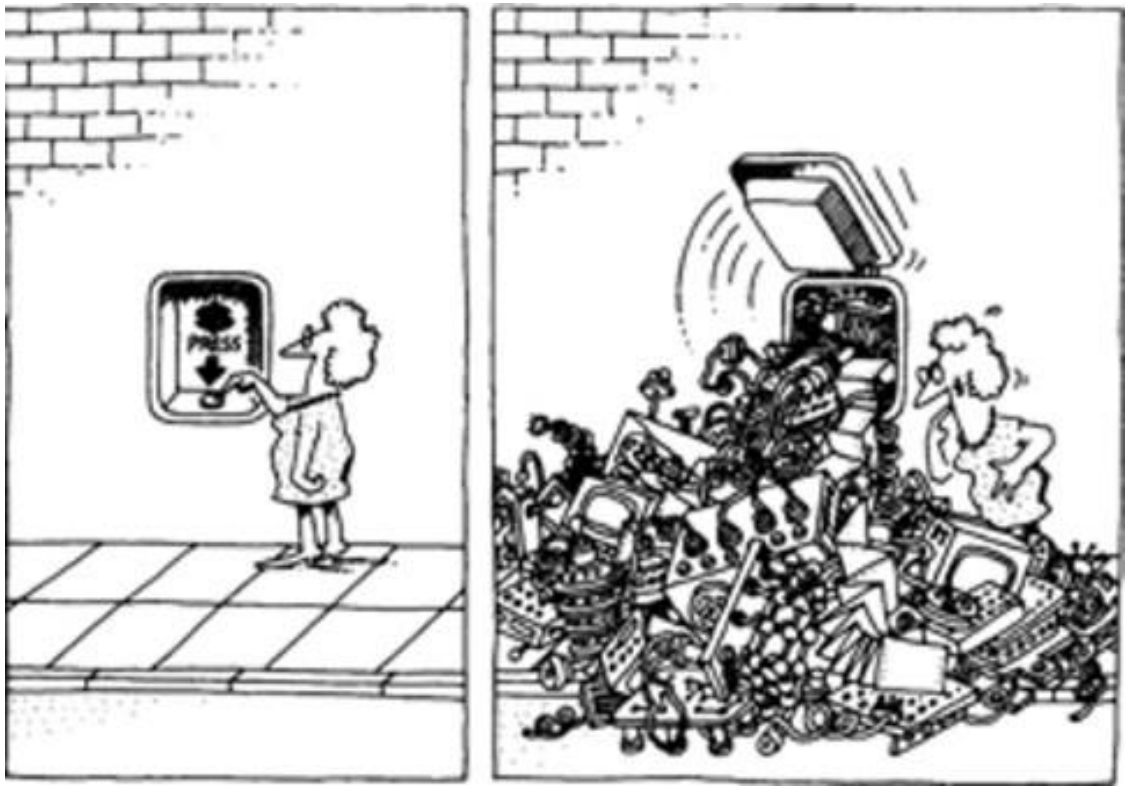
David R. Luna G.
davidrlunag@gmail.com
0412-3111011

Temario del Curso

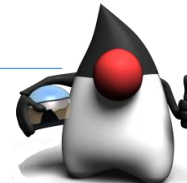
- ✓ Introducción al Lenguaje Java
- ✓ Estructura del Lenguaje Java
- ✓ Herramientas de Desarrollo Java
- ✓ **Programación Orientada a Objetos con Java**
- ✓ Manejo de Datos con Archivos
- ✓ Trabajando con Base de Datos Relaciones
- ✓ Desarrollando GUIs

Programación Orientada a Objetos

La Complejidad



La tarea del equipo de desarrollo de software es ofrecer ilusión de simplicidad, es decir, lograr que para el usuario esa complejidad no se note y por el contrario, hacer pensar que es algo muy sencillo.

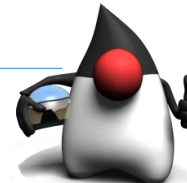


Programación Orientada a Objetos

Significa que el software está organizado como una colección de objetos que modelan las características de los problemas del mundo real -o no-, su comportamiento ante estas características y su forma de interactuar con otros elementos.

Características

- ✓Cambia nuestra forma de pensar sobre los Sistemas
- ✓Los sistemas puede construirse con objetos ya existentes
- ✓La complejidad de los objetos va en aumento
- ✓Creación de Sistemas de funcionamiento correcto

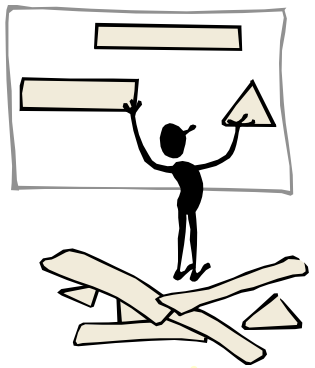


Programación Orientada a Objetos

Dominio del problema

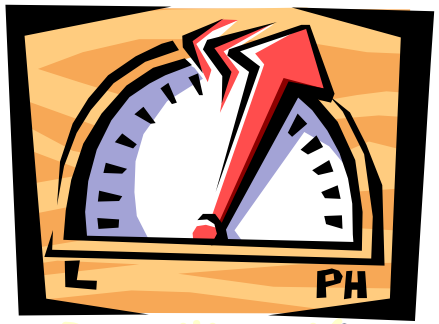
Comunicación.

Consistencia.

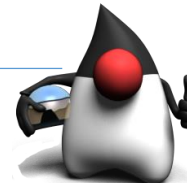


Resistencia al cambio.

Expresión de características comunes.



Reutilización.



Programación Orientada a Objetos

Estabilidad

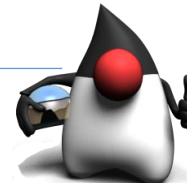
Confiabilidad

Integridad.

Diseño más rápido y de mayor calidad.

Mantenimiento más Sencillo

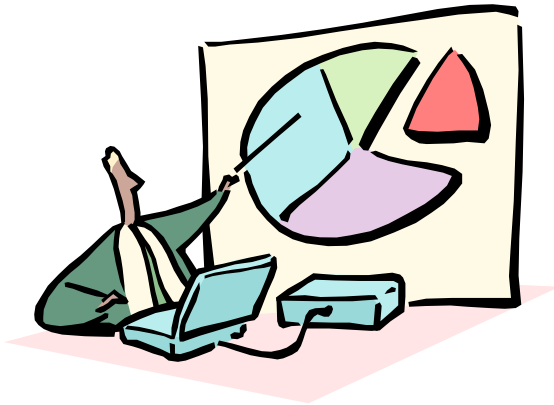
Creación de Bibliotecas de Clases.



Programación Orientada a Objetos

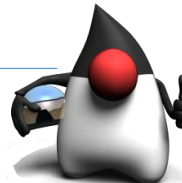
Modelo

Es una abstracción de algo, cuyo objetivo es comprenderlo antes de construirlo.



- ✓Probar antes de construir
- ✓Comunicación con el Cliente
- ✓Visualización
- ✓Reducción de la Complejidad

La tecnología orientada a objetos se apoya en los sólidos fundamentos de la ingeniería, cuyos elementos reciben el nombre global de modelo de objetos.



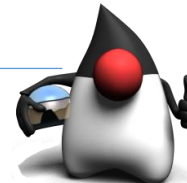
Programación Orientada a Objetos

Una metodología de ingeniería de software es un proceso para producir software de forma organizada, empleando una colección de técnicas y convenciones de notación predefinidas. La metodología suele presentarse como una serie de pasos, con técnicas y notaciones asociadas a cada paso.
La Metodología Orientada a Objetos comprende de varias fases:

Análisis Orientado a Objetos

Diseño Orientado a Objetos

Programación Orientado a Objetos

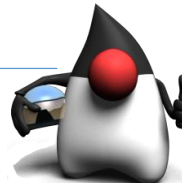


Programación Orientada a Objetos

Nos permite construir un diseño independiente del lenguaje que estará organizado en torno a esos objetos, de tal forma de promover una mejor comprensión de los requisitos, diseños más limpios y sistemas con mejor mantenimiento.

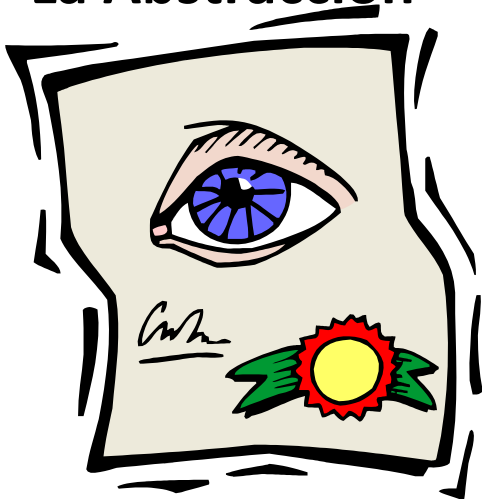
El modelo de objetos abarca los principios de:

- ✓Abstracción
- ✓Encapsulación
- ✓Modularidad
- ✓Jerarquía
- ✓Tipos
- ✓Concurrencia
- ✓Persistencia



Programación Orientada a Objetos

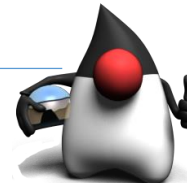
La Abstracción



Una **abstracción** denota las caraterísticas esenciales de un objeto que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador.

Nos permite separar el **comportamiento** de la implementación. Será más importante saber **qué se hace**, y no cómo se hace.

La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

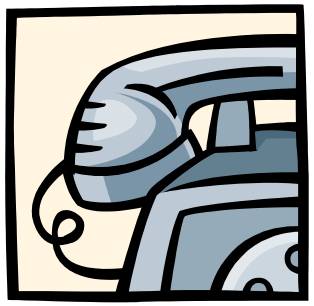


Programación Orientada a Objetos

El Encapsulamiento

Se trata de separar los aspectos externos de un objeto de los detalles internos de la implementación. Los cambios internos no necesariamente afectan la interface externa.

Es por tanto complementario de la abstracción.

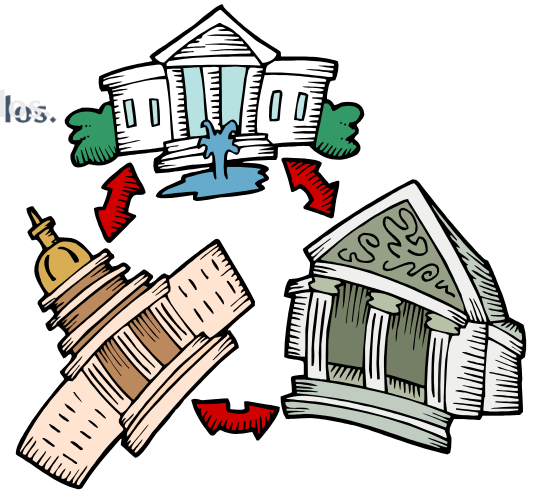


La principal forma de conseguir este objetivo es mediante la ocultación de la información.

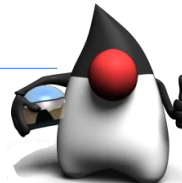
Programación Orientada a Objetos

La Modularidad

Consiste en separar el sistema en bloques poco ligados entre si. Es un concepto relacionado con el encapsulamiento. Es una tarea difícil aunque muy importante en sistemas grandes. Se suele aplicar un método de refinamiento progresivo de los modulos.



Los principios de abstracción, encapsulamiento y modularidad son sínergicos. Un objeto proporciona una frontera bien definida alrededor de una sola abstracción, y tanto el encapsulamiento como la modularidad proporcionan barreras que rodean a esta abstracción.

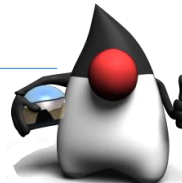


Programación Orientada a Objetos

La Jerarquia

Consiste en separar el sistema en bloques poco ligados entre si. Es un concepto relacionado con el encapsulamiento. Es una tarea difícil aunque muy importante en sistemas grandes. Se suele aplicar un método de refinamiento progresivo de los modulos.

Los principios de abstracción, encapsulamiento y modularidad son sínergicos. Un objeto proporciona una frontera bien definida alrededor de una sola abstracción, y tanto el encapsulamiento como la modularidad proporcionan barreras que rodean a esta abstracción.

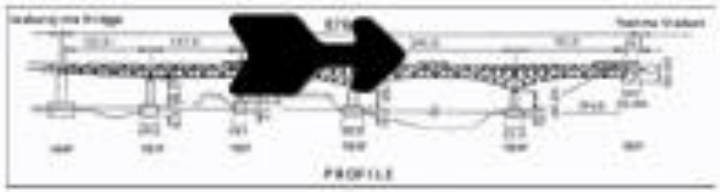


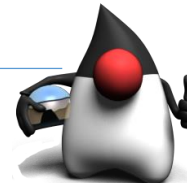
Programación Orientada a Objetos

UML

- Before they build the real thing...

...they first build models...and then learn from them

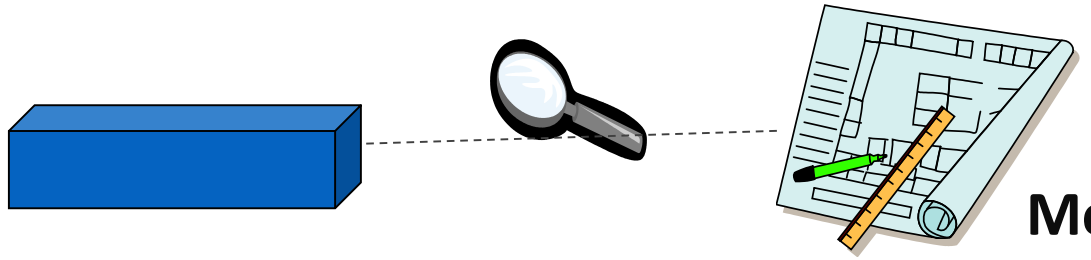




Programación Orientada a Objetos

UML

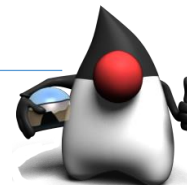
Es un esquema simplificado que describe un sistema o realidad desde un determinado punto de vista que facilita su estudio y compresión



Sistema Software
(complejo)

Modelo
(simplificado)

Los **modelos** de un sistema software se expresan **visualmente** mediante el **lenguaje** de modelado **UML**



Programación Orientada a Objetos

UML – Para que Usar Modelos

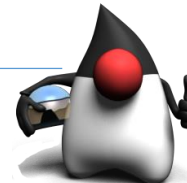
Para detectar errores u omisiones en el diseño antes de comprometer recursos para la implementación

- Analizar y experimentar
- Investigar y comparar soluciones alternativas
- Minimizar riesgos

Para comunicarse con los “stakeholders”

- Clientes, usuarios, implementadores, encargados de pruebas, documentadores, etc.

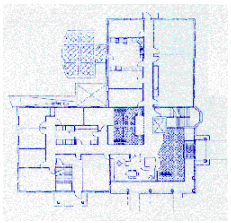
Para guiar la implementación



Programación Orientada a Objetos

UML – Para que Usar Modelos

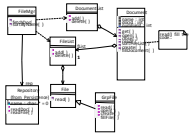
- Arquitectura/Ingeniería de Estructuras
  - Vistas Edificio
    - Vista 3D
    - Alzado/Planta Perfil
    - Estructura del edificio
    - Instalación Eléctrica
    - Instalación Aire Acc.

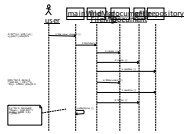


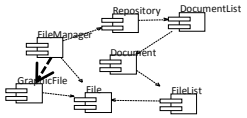
Diagramas

- Herramientas Modelado (ej)
  - Autocad

- Ingeniería Software
  - Modelos UML del Sistema Software
    - Modelo de Casos de uso
    - Modelo de Lógico
    - Modelo de Comportamiento
    - Modelo de Implementación
    - Modelo de Despliegue

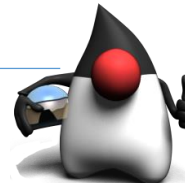






Diagramas

- Herramientas Modelado (ej)
  - Rational Rose



Programación Orientada a Objetos

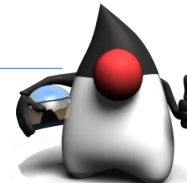
UML – Para que Usar Modelos

UML (*Unified Modeling Language*) es un lenguaje que permite modelar, construir y documentar los elementos que forman un sistema software orientado a objetos. Se ha convertido en el estándar de facto de la industria, debido a que ha sido concebido por los autores de los tres métodos más usados de orientación a objetos: Grady Booch, Ivar Jacobson y Jim Rumbaugh.

Esta notación ha sido ampliamente aceptada debido al prestigio de sus creadores y debido a que incorpora las principales ventajas de cada uno de los métodos particulares en los que se basa: Booch, OMT y OOSE. UML ha puesto fin a las llamadas “guerras de métodos” que se han mantenido a lo largo de los 90, en las que los principales métodos sacaban nuevas versiones que incorporaban las técnicas de los demás. Con UML se fusiona la notación de estas técnicas para formar una herramienta compartida entre todos los ingenieros software que trabajan en el desarrollo orientado a objetos.

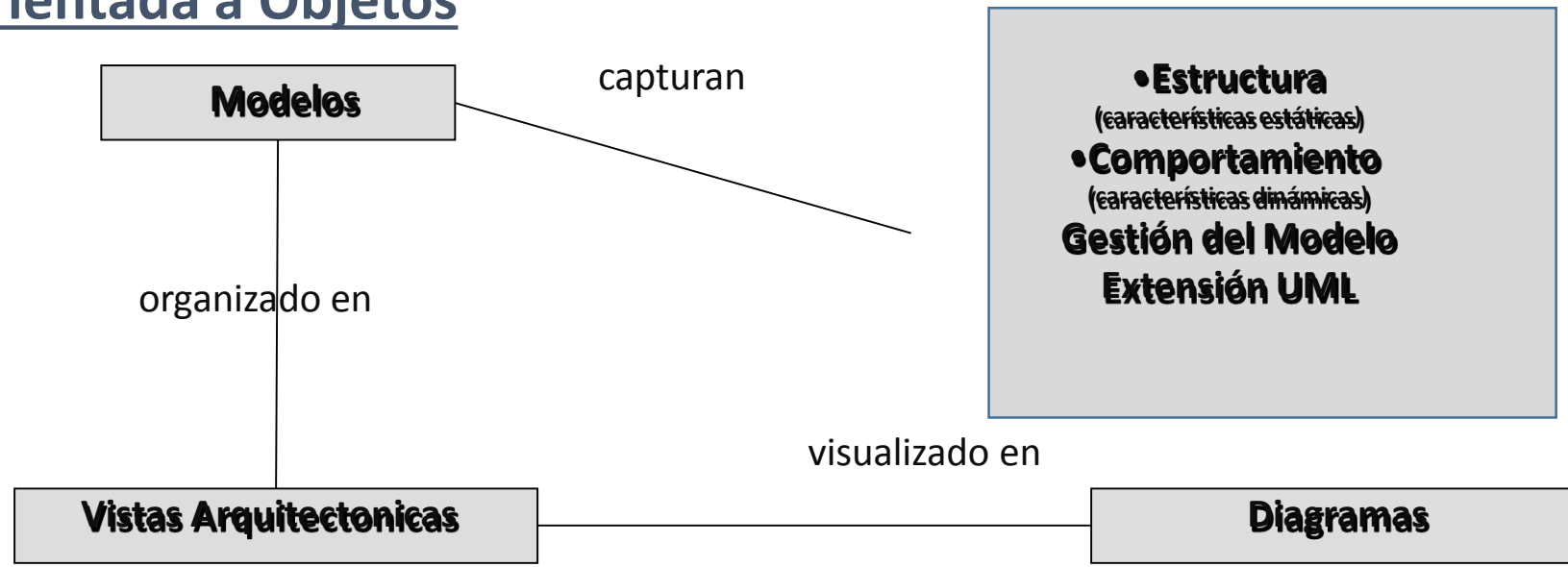
UML no es:

- **una metodología o proceso**
- **un lenguaje de programación**



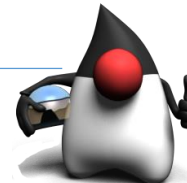
Programación Orientada a Objetos

UML



- Vista Estática
- Vista de casos de uso
- Vista de Implementación
- Vista de Despliegue
- Vista de Máquina de estados
- Vista de Actividad
- Vista de Interacción
- Vista de Gestión del Modelo
- Todas

- Diagramas de Casos de Uso
- Diagramas de Clases
- Diagramas de Componentes
- Diagramas de Despliegue
- Diagramas de Estados
- Diagramas de Actividad
- Diagramas de Secuencia
- Diagramas de Colaboración
- Diagramas de Clases
- Todos

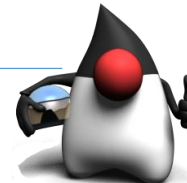


Programación Orientada a Objetos

UML

Con el nombre de Diagramas de Estructura Estática se engloba tanto al Modelo Conceptual de la fase de Análisis como al Diagrama de Clases de la fase de Diseño. Ambos son distintos conceptualmente, mientras el primero modela elementos del dominio el segundo presenta los elementos de la solución software. Sin embargo, ambos comparten la misma notación para los elementos que los forman (clases y objetos) y las relaciones que existen entre los mismos (asociaciones).

Diagrama de Clases

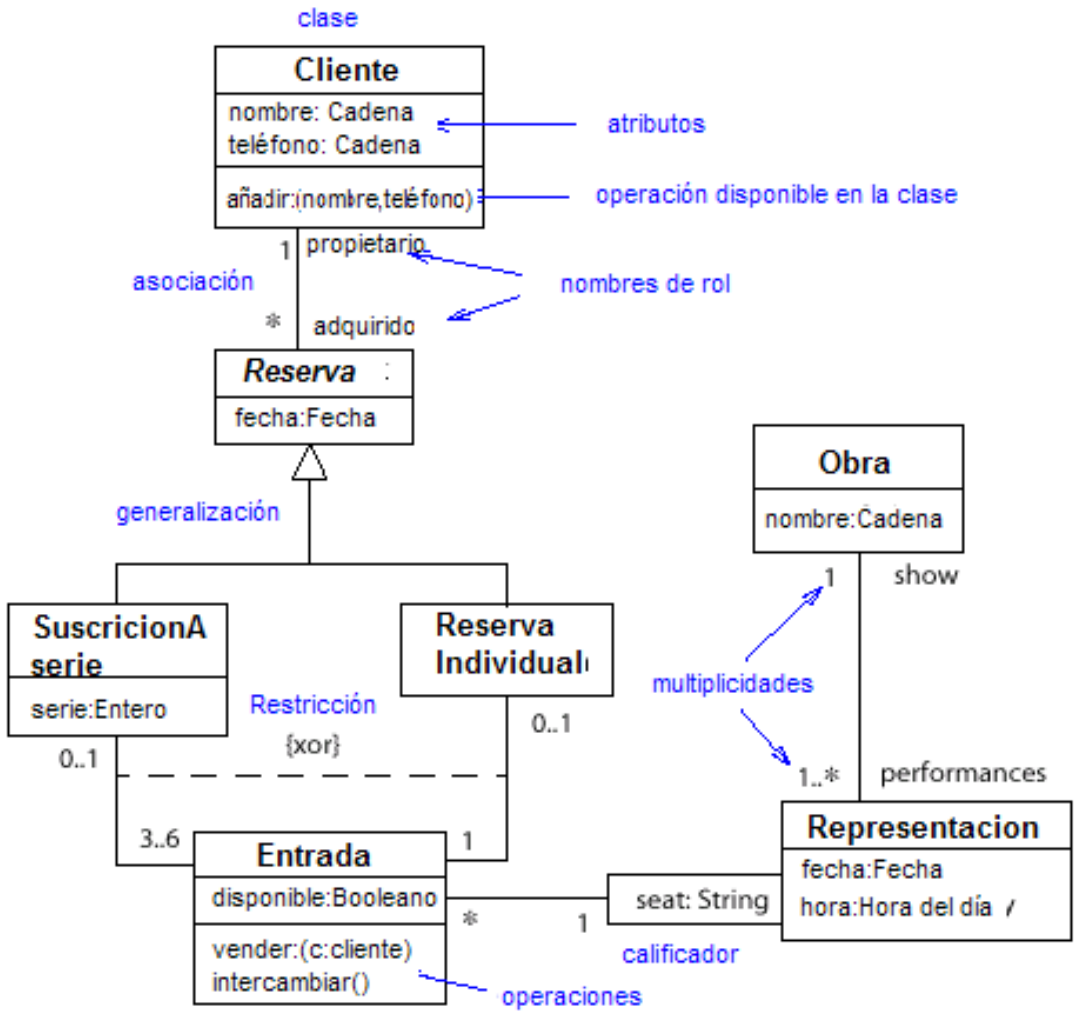


Programación Orientada a Objetos

UML – Diagrama de Clases

Una clase se representa mediante una caja subdividida en tres partes: En la superior se muestra el nombre de la clase, en la media los atributos y en la inferior las operaciones. Una clase puede representarse de forma esquemática (plegada), con los detalles como atributos y operaciones suprimidos, siendo entonces tan solo un rectángulo con el nombre de la clase.

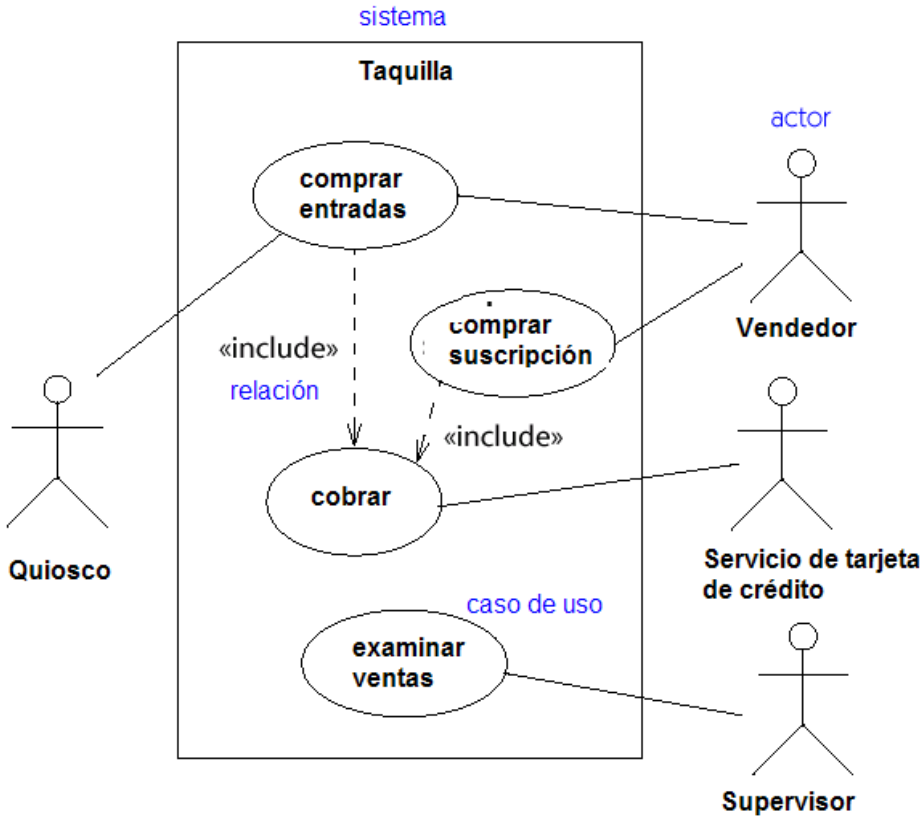
+ Public
- Private
 # Protected

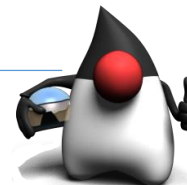


Programación Orientada a Objetos

UML – Diagrama de Casos de Uso

Un Diagrama de Casos de Uso muestra la relación entre los actores y los casos de uso del sistema. Representa la funcionalidad que ofrece el sistema en lo que se refiere a su interacción externa. Los elementos que pueden aparecer en un Diagrama de Casos de Uso son: actores, casos de uso y relaciones entre casos de uso.



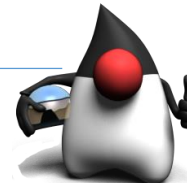


Programación Orientada a Objetos

UML – Diagrama de Estados

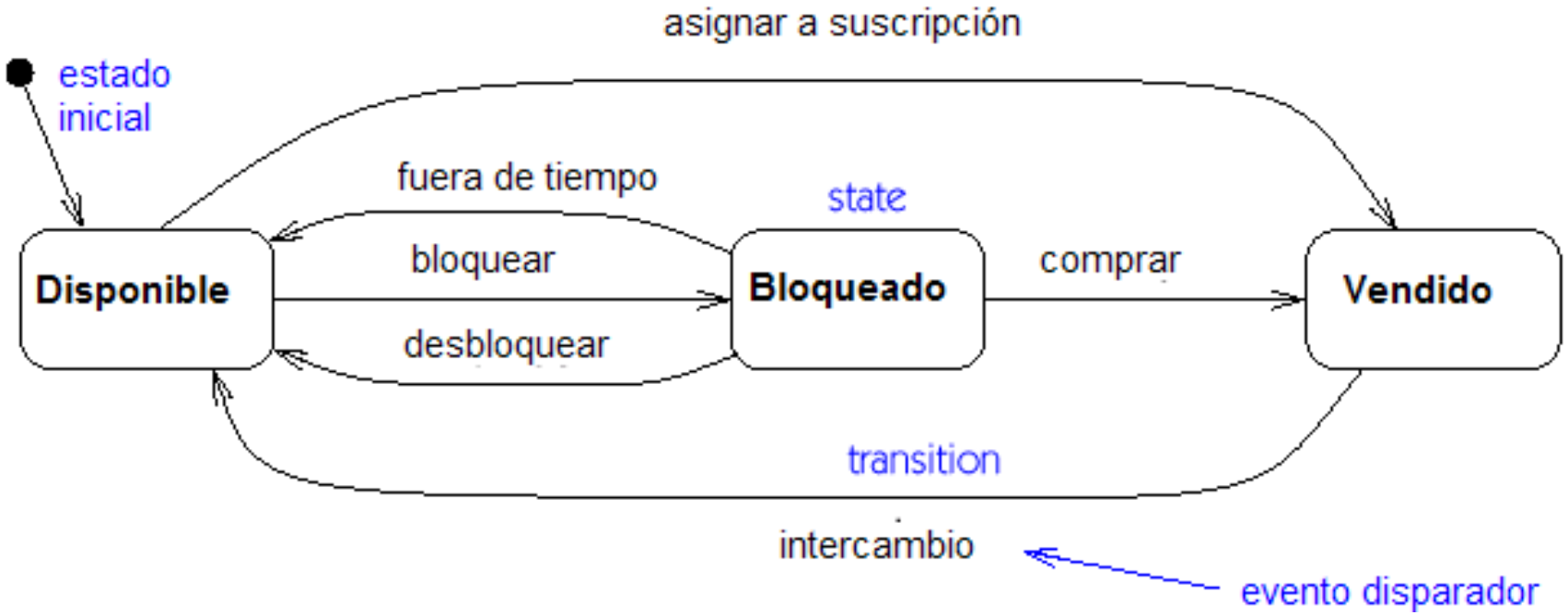
Un Diagrama de Estados muestra la secuencia de estados por los que pasa un caso de uso o un objeto a lo largo de su vida, indicando qué eventos hacen que se pase de un estado a otro y cuáles son las respuestas y acciones que genera.
En cuanto a la representación, un diagrama de estados es un grafo cuyos nodos son estados y cuyos arcos dirigidos son transiciones etiquetadas con los nombres de los eventos.
Un estado se representa como una caja redondeada con el nombre del estado en su interior. Una transición se representa como una flecha desde el estado origen al estado destino.

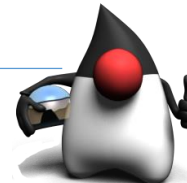
Un diagrama de estados puede representar ciclos continuos o bien una vida finita, en la que hay un estado inicial de creación y un estado final de destrucción (del caso de uso o del objeto). El estado inicial se muestra como un círculo sólido y el estado final como un círculo sólido rodeado de otro círculo. En realidad, los estados inicial y final son pseudoestados, pues un objeto no puede "estar" en esos estados, pero nos sirven para saber cuáles son las transiciones inicial y final(es).



Programación Orientada a Objetos

UML – Diagrama de Estados





Programación Orientada a Objetos

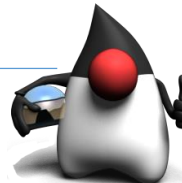
Herencia

La verdadera potencia de la programación orientada a objetos radica en su capacidad para reflejar la abstracción que el cerebro humano realiza automáticamente durante el proceso de aprendizaje y el proceso de análisis de información.

La herencia es el mecanismo fundamental de relación entre clases en la orientación a objetos. Relaciona las clases de manera jerárquica; una clase *padre* o *superclase* sobre otras clases *hijas* o *subclases*.

Los descendientes de una clase heredan todas las variables y métodos que sus ascendientes hayan especificado como *heredables*, además de crear los suyos propios.

La característica de herencia, nos permite definir nuevas clases derivadas de otra ya existente, que la especializan de alguna manera. Así logramos definir una jerarquía de clases, que se puede mostrar mediante un árbol de herencia



Programación Orientada a Objetos

Herencia

Para indicar que una clase deriva de otra, heredando sus propiedades (métodos y atributos), se usa el término *extends*

```
public class Persona{
    String cedula;
    String Nombre;
   String dir;
    int edad;

    public void metodo1(){
       System.out.println("SOY EL METODO 1");
    }
}
```

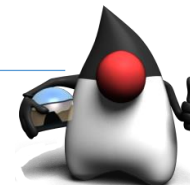
```
public class Empleado extends Persona{
    String Dpto;
    String Ingreso;
    float Sueldo;

    public void trabaja(){
        System.out.println("SOY UN EMPLEADO");
    }
}
```

En la terminología habitual, la clase que hereda las características de otra y la clase de partida reciben los calificativos de "subclase" y "superclase"

No se olvide que solo se puede heredar de una sola clase.

No hay herencia Multiple.



Programación Orientada a Objetos

Herencia

Limitaciones en la Herencia

Todos los campos y métodos de una clase son siempre accesibles para el código de la misma clase.

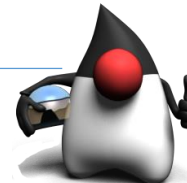
Para controlar el acceso desde otras clases, y para controlar la herencia por las subclase, los miembros (atributos y métodos) de las clases tienen tres modificadores posibles de control de acceso:

Public: Los miembros declarados *public* son accesibles en cualquier lugar en que sea accesible la clase, y son heredados por las subclases.

Private: Los miembros declarados *private* son accesibles sólo en la propia clase.

Protected: Los miembros declarados *protected* son accesibles sólo para sus subclases

Recuerda que una clase declarada como final, no podrá ser utilizada para que sea hererada.



Programación Orientada a Objetos

Interfaces

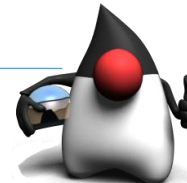
Una **interfaz** en Java es una colección de métodos abstractos y propiedades.
En ellas se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describan la lógica del comportamiento de los métodos.

La principal diferencia entre interface y abstract es que un interface proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

modificador_acceso interface NombreInterfaz { .... }

Para implementarla en una clase, se utiliza la forma:

class NombreClase implements NombreInterfaz1

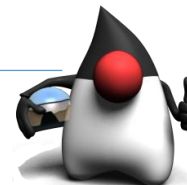


Programación Orientada a Objetos

Interfaces

```
interface Nave {
        public void moverPosicion (int x, int y);
        public void disparar();
}

public class NaveJugador implements Nave {
        public void moverPosicion (int x, int y){
                //Implementación del método
        }
        public void disparar(){
                //Implementación del método
        } .
}
```



Programación Orientada a Objetos

SobreEscritura

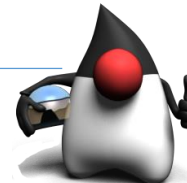
Durante una jerarquía de herencia puede interesar volver a escribir el cuerpo de un método, para realizar una funcionalidad de diferente manera dependiendo del nivel de abstracción en que nos encontremos. A esta modificación de funcionalidad se le llama sobrescritura de un método.

```
public class Prueba{
public void muestraData(int x){
System.out.println("X="+x);
}
}
```

```
public class Prueba2 extends Prueba{
public void muestraData(int x){
System.out.println("El Numero="+x);
}
}
```

```
public class PruebaTest{
public static void main(String a[]){
Prueba p1=new Prueba();
Prueba2 p2=new Prueba2();
p1.muestraData(8);//Muesta X=8
p2.muestraData(10);//Muestra El Numero= 10
}
}
```

La Maquina Virtual de Java, busca siempre la firma en la clase que está siendo llamada, si no lo consigue, lo busca en la superclase.



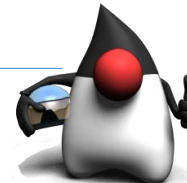
Programación Orientada a Objetos

SobreEscritura

Nos permite brindar una implementación diferente de un método para cada subclase. El compilador llamará al método sobreescrito. La estructura del método deberá ser exactamente igual que la de la subclase, incluyendo el tipo de retorno y la lista de parámetros, sino será tratado como sobrecarga.

Los métodos se seleccionan en función del tipo de la instancia en tiempo de ejecución, no a la clase en la cual se está ejecutando el método actual. A esto se le llama *selección Dinámica de Método*

Todos los métodos y las variables de instancia se pueden sobrescribir por defecto. Si se desea declarar que ya no se quiere permitir que las subclases sobrescriban las variables o métodos, éstos se pueden declarar como final. La Sobreescritura permite soportar el Polimorfismo.



Programación Orientada a Objetos

Polimorfismo

Consiste en la posibilidad de usar indistintamente todos los objetos de una clase y sus derivadas.

"Es el mecanismo que permite definir e Invocar funciones idénticas en denominación e interfaz, pero con implementaron diferente".

En otras palabras, podemos hacer que una superclase pueda tener una referencia de cualquiera de sus subclases, en donde solo se podrá utilizar los métodos sobreescritos en la subclase. Es de mucha utilidad, cuando queremos que se cumpla una función sin importar la instancia del objeto que va a ser la llamada del método.

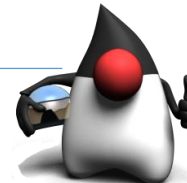
Para que exista polimorfismo debe haber una relación de herencia y sobreescritura.

```
public class Prueba{
public void muestraData(int x){
System.out.println("SUPER
CLASE");
System.out.println("X="+x);
}
}
```

```
public class Prueba3 extends Prueba{
public void muestraData(int x){
System.out.println("SOY PRUEBA3");
System.out.println("El Numero="+x);
}
```

```
public class Prueba2 extends Prueba{
public void muestraData(int x){
System.out.println("SOY PRUEBA2");
System.out.println("El Numero="+x);
}
}
```

```
public class Prueba4 extends Prueba{
public void muestraData(int x){
System.out.println("SOY PRUEBA4");
System.out.println("El Numero="+x);
}
}
```

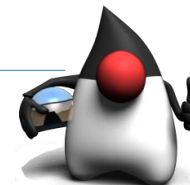


Programación Orientada a Objetos

Polimorfismo

Ahora podemos crear un método que llame al método muestraData, sin importar la instancia del objeto.

```
public class PruebaTest{
public void verData(Prueba x, int n){
x.muestraData(n);
}
public static void main(String a[]){
Prueba p1 =new Prueba2();
Prueba p2 = new Prueba3();
Prueba p3 = new Prueba4();
verData(p1, 19);
verData(p2, 12);
verData(p3, 10);
Prueba p4= new Prueba();
verData(p4, 8);
}
}
```



Programación Orientada a Objetos

Manejo de Clases Avanzado

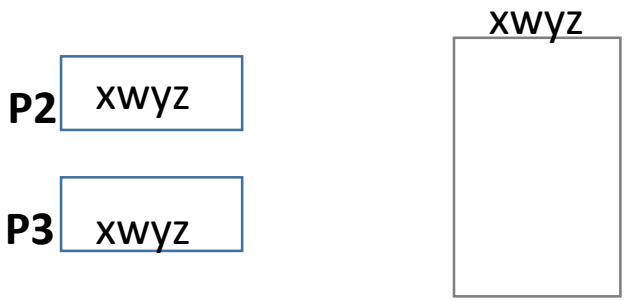
Cada vez que se crea una clase se añade otro tipo de dato que se puede utilizar igual que uno de los tipos simples. Por ello al declarar una nueva variable, se puede utilizar un nombre de clase como tipo. A estas variables se las conoce como *referencias a objeto*.

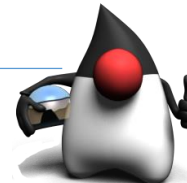
El operador *new* crea una instancia de una clase (*objetos*) y devuelve una referencia a ese objeto. Por ejemplo:

MiPunto p2 = new MiPunto(2,3);

Este es un ejemplo de la creación de una instancia de *MiPunto*, que es controlador por la referencia a objeto *p2*. Hay una distinción crítica entre la forma de manipular los tipos simples y las clases en Java: Las referencias a objetos realmente no contienen a los objetos a los que referencian. De esta forma se pueden crear múltiples referencias al mismo objeto, como por ejemplo:

MiPunto p3 =p2;





Programación Orientada a Objetos

Manejo de Clases Avanzado

El operador punto (.) se utiliza para acceder a las variables de instancia y los métodos contenidos en un objeto, mediante su referencia a objeto:

referencia_a_objeto.nombre_de_variable_de_instancia

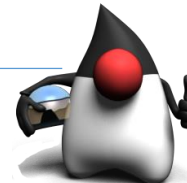
referencia_a_objeto.nombre_de_método(lista-de-parámetros).

Hemos creado un ejemplo completo que combina los operadores *new* y punto para crear un objeto *MiPunto*, almacenar algunos valores en él e imprimir sus valores finales:

```
MiPunto p6 = new MiPunto( 10, 20 );
System.out.println (“p6.- 1. X=“ + p6.x + “ , Y=“ + p6.y);
p6.inicia( 30, 40 );
System.out.println (“p6.- 2. X=“ + p6.x + “ , Y=“ + p6.y);
```

Cuando se ejecuta este programa, se observa la siguiente salida:

p6.- 1. X=10 , Y=20



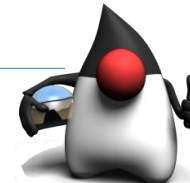
Programación Orientada a Objetos

Manejo de Clases Avanzado

Java incluye un valor de referencia especial llamado *this*, que se utiliza dentro de cualquier método para referirse al objeto actual. El valor *this* se refiere al objeto sobre el que ha sido llamado el método actual. Se puede utilizar *this* siempre que se requiera una referencia a un objeto del tipo de una clase actual. Si hay dos objetos que utilicen el mismo código, seleccionados a través de otras instancias, cada uno tiene su propio valor único de *this*.

En Java se permite declarar variables locales, incluyendo parámetros formales de métodos, que se solapen con los nombres de las variables de instancia. No se utilizan *x* e *y* como nombres de parámetro para el método *inicia*, porque ocultarían las variables de instancia *x* e *y* reales del ámbito del método. Si lo hubiésemos hecho, entonces x se hubiera referido al parámetro formal, ocultando la variable de instancia *x*.

```
void inicia2( int x, int y ) {
x = x; // Ojo, no modificamos la variable de instancia!!!
this.y = y; // Modificamos la variable de instancia!!!
}
}
```

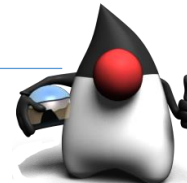


Programación Orientada a Objetos

Manejo de Clases Avanzado

Clases Finales

Una clase final se declara anteponiendo el modificador final a la palabra clave class.
Una clase final es una clase que no puede ser subclaseada, es decir, hererada.
¿Pórque crear clases finales?
Seguridad
Una subclase es “similar” a la superclase y, además puede implementar un comportamiento “peligroso”. Es posible a través de una subclase obtener información privada de la superclase. Por ejemplo: la clase String del paquete java.lang es una clase final ya que es vital para el funcionamiento del compilador y del intérprete. Al no poder subclasearse, cualquier clase que use String estará usando la clase java.lang.String.
Diseño
Por razones de diseño, una clase que está conceptualmente bien definida no tendría que ser subclaseada.



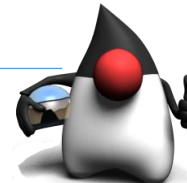
Programación Orientada a Objetos

Manejo de Clases Avanzado

Clases Finales

```
public final class Persona {
String nombre;
byte edad;
public void camina(byte distancia) {
for(int i=1;i<=distancia;i++)
System.out.println("Camine " + i + " Mts.");
}
}
```

Tenemos la clase Persona declarada como final, si intentamos crear una subclase a partir de ella obtendremos el siguiente error: cannot inherit from final NombreClase



Programación Orientada a Objetos

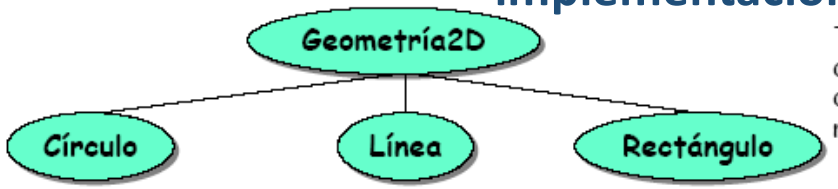
Manejo de Clases Avanzado

Clases Abstractas

Una clase abstracta representa un concepto abstracto.
Una clase abstracta sólo puede ser subclaseada, pero nunca puede ser instanciada.
Una clase abstracta se declara anteponiendo el modificador abstract a la palabra clave class.

```
abstract class Geometría2D {
.....
}
```

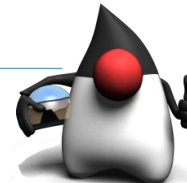
Una clase abstracta puede contener métodos abstractos y métodos con implementación.



Todos estos objetos tienen estado -posición y área de dibujo- y comportamiento -mover, cambiar tamaño, dibujar- común. Por ejemplo:
mover tiene implementación y dibujar es abtracto.

```
abstract class Geometría2D {
   int x,y;
   .....
   void mover(int nuevoX, int nuevoY) {
   ...........
   }
   abstract void dibujar();
}
```

```
Class Circulo extends Geometría2D {
        void dibujar() {
        ...........
        }
}
Class Linea extends Geometría2D {
        void dibujar() {
        ...........
        }
}
```



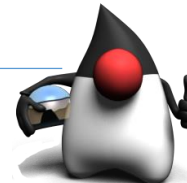
Programación Orientada a Objetos

Manejo de Clases Avanzado

Clases Abstractas

```
class Punto{
int x;
int y;
int color;
}
abstract class Figura{
Punto ptoOrigen;
abstract void dibujar();
}
class Rectangulo extends Figura{
Punto ptoFinal;
void dibujar() {...}
}
class Circulo extends Figura{
int radio;
void dibujar() {...}
}
```

Para tratar con ellas creo una clase genérica: figura.
Todos tienen dos métodos básicos, area y perímetro.
Todos se calculan diferente.
En figura empaqueto toda la parte común.
Sin embargo figura no representa ningún objeto real, por lo que no podemos implantar ningún método.
Uso métodos abstractos:
Se define con la palabra abstract
Es un método sin código, sólo tiene signatura seguida por un ;
Limitan las clases que los contienen
Una clase con un método abstract es abstracta y debe ser declarada como tal.
Una clase abstracta no puede ser instanciada (no podemos crear objetos de la misma)
Una subclase de una clase abstracta sólo puede ser instanciada si solapa todos los métodos abstractos de su superclase y los implementa. Una subclase de este tipo se denomina “concreta” para resaltar que no es abstracta
Si la subclase ni solapa ni implementa los métodos abstract de la superclase, es abstracta.
Los métodos static, private y final no pueden ser abstract, ya que no pueden ser solapados por la subclase.
Una clase final no podrá contener métodos abstractos
Una clase se puede declarar abstract aunque no tenga ningún método abstract
En nuestro ejemplo de figuras geométricas, debemos asegurarnos de definir todos los métodos abstractos de las superclase si queremos crear objetos de las mismas.
Una clase abstracta no puede tener métodos privados porque no podrían implementarse ni estáticos.



Programación Orientada a Objetos

Manejo de Clases Avanzado

Clases Anidadas

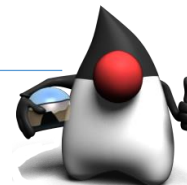
Una clase interna estatica tiene las mismas caracteristicas de cualquier miembro estatico de una clase. Podemos tener acceso a ella sin necesidad de crear un objeto de la clase que la contiene, y ellas pueden acceder solo a los miembros estaticos (variables y metodos) de la clase que la contiene.

Para declarar una clase anidada siempre se debe utilizar la palabra reservada static, se puede definir tambier un modificador de acceso (public, protected or private), pero por defecto la clase anidada tiene acceso al paquete de la clase que la contiene.

Aquí tenemos un ejemplo:

```
class MyOuter {
public static class MyInner {
//...
public void function1() {}
//more function (static and more)
}
}
```

Cuando compilamos tenemos lo siguiente:
MyOuter.class : Engloba la clase contenedora.
MyOuter$MyInner.class : Engloba la clase interna



Programación Orientada a Objetos

Manejo de Clases Avanzado

Clases Anidadas

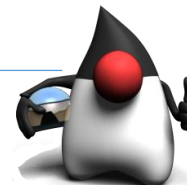
Para construir una instancia de la clase anidada, sería:

```
public MyInner myclass = MyOuter.MyInner();
```

Si la clase MyOuter fue definida en otro paquete llamado my.package se haría de la siguiente forma:

```
MyInner myclass = my.package.MyOuter.MyInner();
```

Solo tiene acceso a los miembros estaticos de la clase que lo contiene.



Programación Orientada a Objetos

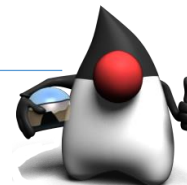
Manejo de Clases Avanzado

Clases Internas

Las clases internas difieren de las clases anidadas, porque no tienen el modificador static. Cada instancia de una clase interna requiere una instancia de la clase que la contiene y esta puede tener varias instancias de las clase interna. Existen tres tipos de clases internas:

Clases Internas Miembro: Este tipo de clase interna es definida como miembro de la clase que la contiene, son definidas al mismo nivel que las variables y los metodos miembros, por lo que tendra sus mismas caracteristicas.

```
class MyOuter {
 private float variable = 0;
public void donothing() { //sentencias}
private class MyInner {
        public void innerfunction() {//sentencias}
        public void function() {MyOutter.this.donothing(); }
 }
```



Programación Orientada a Objetos

Manejo de Clases Avanzado

Clases Internas

Clases Internas Anonimas:

Este es un tipo muy especial de clase interna. Una clase anonima no tiene nombre. Se utilizan mucho en el desarrollo de interfaces graficas con Java.

Ejemplo:

```
class Outer extends java.awt.Frame{
      public Outer(){
            java.awt.Button button = new java.awt.Button("Click on me");
            button.addActionListener(new ActionListener(){
                  public void actionPerformed(ActionEvent e){
                        System.out.println("You clicked");
                  }
            });
      }
}
```

En el codigo la clase ActionListener es una clase anonima.

No pueden ser abstractas ni estaticas. Es considerada como final.

Puede extender de otra clase o implementar una interface. No ambas.