

Java Inicial

(20 horas)





Temario

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. POO en Java
5. **Relaciones entre Objetos**
6. Polimorfismo, abstracción e interfaces
7. Excepciones
8. Conceptos avanzados



Tema 5

Relaciones entre Objetos



Objetivos

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. POO en Java
5. **Relaciones entre Objetos**
6. Polimorfismo, abstracción e interfaces
7. Excepciones
8. Conceptos avanzados

- Relación de Composición
- Relación de asociación
- Relación de Uso
- Herencia
 - Tipos de herencia
 - Sintaxis
 - Constructores = Super
 - Redefinición de métodos

RELACIONES ENTRE OBJETOS

- Un conjunto de objetos aislados tiene escasa capacidad para resolver un problema.
- En una aplicación útil los objetos colaboran e intercambian información, mantienen distintos tipos de relaciones entre ellos

- ¿Todo lo podemos hacer con una UNICA clase?
 - MODULARIDAD: Descomponer en módulos.
 - Vida real de los objetos:
 - Unos están compuestos de otros (Coche y Motor)
 - Unos necesitan de la colaboración de otros para llevar a cabo ciertas tareas (Mecánico y Herramienta)
 - 4 tipo de relaciones
 - **Composición:** Al crear A se crea B
 - **Asociación:** Un atributo de A es una referencia a un objeto B
 - **Uso:** A necesita de B para hacer una funcionalidad
 - **Herencia:** Una Clase obtiene funcionalidad de otra, añadiendo, bien nuevas características (*atributos*), y por tanto un comportamiento más complejo, bien modificando su comportamiento.

■ Relaciones de Asociación: Ejemplo

□ Clase Cliente

```
public class Cliente
{
    //Declaracion de ATRIBUTOS
    private String nombre, domicilio, dni;
```

□ Clase Cuenta Corriente

```
public class CuentaCorriente
{
    //Declaracion de ATRIBUTOS
    int id;
    double saldo;
    Cliente titular;
```

■ Ejercicio práctico:

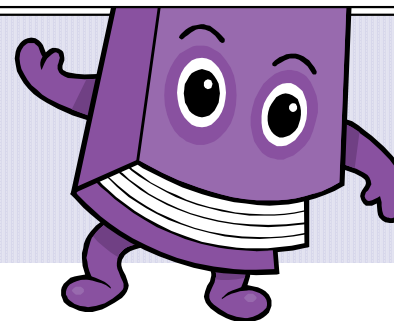
□ Definir la Clase Cliente y la Clase Cuenta

```
public class Cuenta {
    long numero;
    Cliente titular;
    private float saldo;
    float interesAnual;

    // Constructor general
    public Cuenta (long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = aInteresAnual;
    }

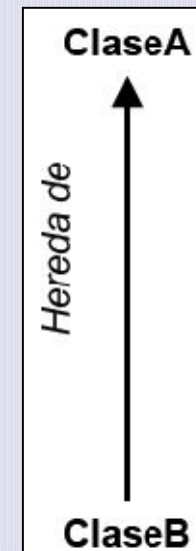
    Cliente leerTitular () { return titular; }
}
```

■ Suerte!!!



■ Relación de Herencia

- Permite que una Clase obtenga la funcionalidad de otra, añadiendo nuevas características (*atributos*) y/o bien modificando su comportamiento.
- Hereda Absolutamente todo
 - ClaseA = **clase Base**,
clase Padre o Superclase.
 - ClaseB = **clase Derivada**,
clase Hija o Subclase.



■ Clases internas

- Clases que se definen **dentro** de otras.
- Empaquetar en una clase aquellas que no tienen utilidad fuera del contexto de esta.
- La clase interior puede ser *public*, *private* o *protected*.
- Los objetos '*interiores*' quedan permanentemente ligado al *englobante*.

■ Clases internas: *Ejemplo*

```
public class Cuenta {
    long numero;
    Cliente titular;
    private float saldo;
    float interesAnual;
    LinkedList movimientos; // Lista de movimientos

    private class movimiento {
        Date fecha;
        char tipo;
        float importe;
        float saldoAct;

        public Movimiento (Date aFecha, char aTipo, float aImporte) {
            fecha = aFecha;
            tipo = aTipo;
            importe = aImporte;
            saldoAct = saldo; // Copiamos el saldo actual
        }
    }

    // Constructor general
    public Cuenta (long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero; titular = aTitular; saldo = 0;
        interesAnual = aInteresAnual;
        movimientos = new LinkedList ();
    }
}
```

- Autoreferencia: El puntero `this`
 - Sirve para obtener una referencia a la implementación del propio objeto.
 - Cuando encontremos *this* en una expresión podemos sustituirlo por '*este objeto*'
 - Aunque no es necesarios (*redundante*) podemos llamar desde la implementación de una operación a otra operación del propio mismo objeto.
 - Aunque redundante → clarifica!

- Autoreferencia: El puntero this
 - Supongamos la siguiente definición de clase

```
class ClaseA
{
    public int n;
    public void metodoM()
```

```
class ClaseA
{
    public void metodoM()
    {
        this.n = 25; // n es el atributo del objeto
                    // que recibe el mensaje
        ...
    }
}
```

```
}
}
```

■ El puntero *this*: Ejemplo

- Llamada a la operación *ingreso* desde *ingresoInteresMes*

```
public class Cuenta {
    long numero;
    Cliente titular;
    private float saldo;
    float interesAnual;

    public void ingresoInteresMes () { this.ingreso (interesAnual * saldo / 1200); }

    // Resto de las operaciones de la clase Cuenta
}
```

- Implementar la operación *transferirDesde* llamando a una operación *transferirA*

```
public class Cuenta {
    long numero;
    Cliente titular;
    private float saldo;
    float interesAnual;

    public void transferirA (Cuenta c, float cant) {
        reintegro (cant); c.ingreso (cant);
    }
    public void transferirDesde (Cuenta c, float cant) {
        c.transferir_a (this, cant);
    }
}
```

■ El puntero *this*: Ejemplo 2

- Otra utilidad puede ser llamar a un constructor desde otro constructor.

```
public class cuenta {
    long numero;
    Cliente titular;
    private float saldo;
    float interesAnual;

    // Constructor general
    public Cuenta (long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero; titular = aTitular; saldo = 0; interesAnual = aInteresAnual;
        movimientos = new LinkedList ();
    }

    // Constructor específico para cuentas de ahorro (interesAnual = 0.1%)
    public Cuenta (long aNumero, Cliente aTitular) {
        this (aNumero, aTitular, 0.1);
    }
}
```



Un constructor no es una operación ordinaria. Únicamente puede llamarse a un constructor desde otro constructor y debe ser la primera instrucción de su implementación.

■ Herencia:

- Mecanismo de la OOP que permite construir una clase incorporando de **manera implícita** todas las características de una clase previamente existente.

■ Herencia: Razones

- Modelado de la realidad. Son frecuentes las relaciones de especialización/generalización entre las entidades del mundo real, por tanto es lógico que dispongamos de un mecanismo similar entre las clases de objetos
- Evitar redundancias. Toda la funcionalidad que aporta una clase de objetos es adoptada de manera inmediata por la clase que hereda, por tanto evitamos la repetición de código entre clases semejantes
- Sirve de soporte para el polimorfismo

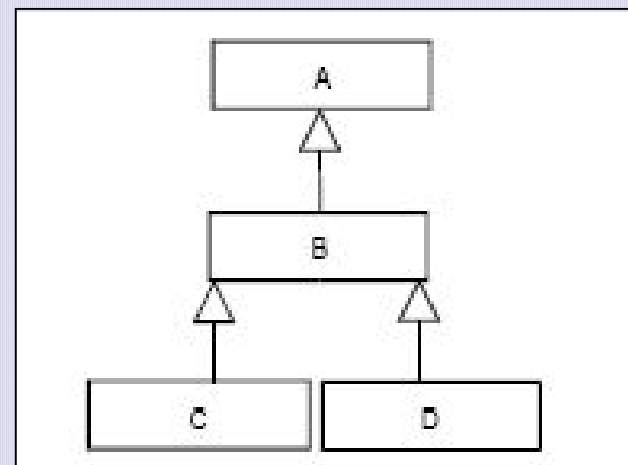
■ Herencia: Definición

□ Sea una clase A. Si una segunda clase B hereda de A entonces decimos:

- A es ascendiente o superclase de B. Si la herencia entre A y B es directa decimos además que A es la clase padre de B
- B es un descendiente o subclase de A. Si la herencia entre A y B es directa decimos además que B es una clase hija de A

□ Siempre tenemos un padre

En java es la superclase *Object*.

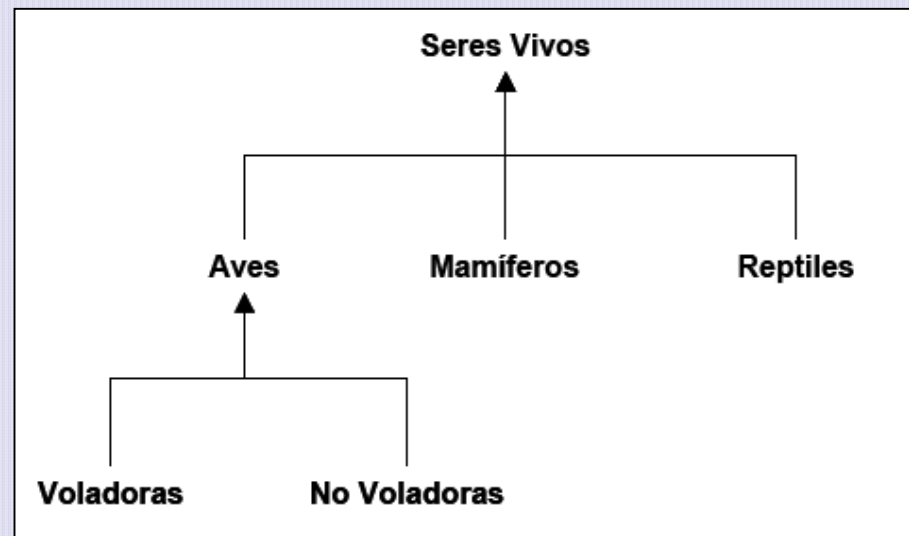


■ Tipos de Herencia

- **Herencia Simple:** una clase Derivada hereda, directamente, de una única clase Base (*ver esquema de los Seres Vivos*).
- **Herencia Múltiple:** una clase Derivada hereda, directamente, de más de una clase Base. Java NO LA SOPORTA.
- **Herencia Restrictiva:** una clase Derivada hereda solamente parte de una clase Base. Java NO LA SOPORTA.

■ Herencia: Ejemplo Seres Vivos

- Todo Mamífero ES UN Ser Vivo
- Todo Ave Voladora ES UN Ave y POR TANTO un Mamífero



TODO objeto de la ClaseB ES UN objeto de la ClaseA

■ Herencia: Sintaxis

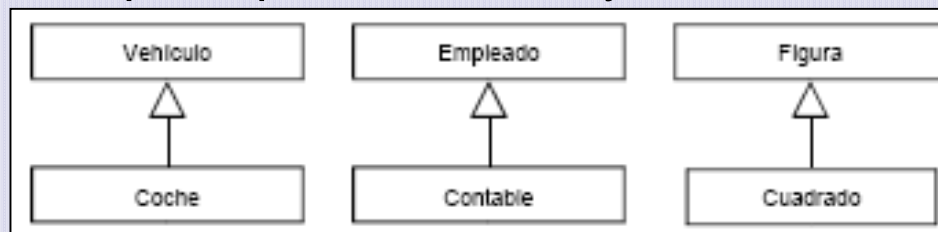
```
class ClaseB extends ClaseA
{
    ...
}
```

- La ClaseB tiene toda la funcionalidad que posee la ClaseA
 - Incluyendo sus atributos en caso que su visibilidad se lo permita (No sea *private*)
- Nota: Todas las clases heredan de **Object**

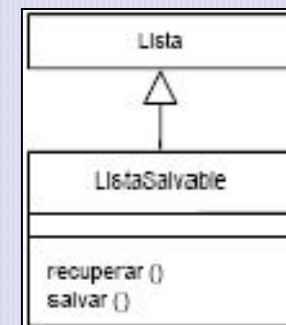
■ Herencia: Situaciones

□ Puede aplicarse herencia en:

- **Especialización.** Dado un concepto B y otro concepto A que representa una especialización de B, entonces puede establecerse una relación de herencia entre las clases de objetos que representan a A y B.



- **Extensión.** Una clase puede servir para extender la funcionalidad de una superclase sin que represente necesariamente un concepto más específico.



■ Ejemplo workspace:

- En `com.slashmobility.PruebaCirculo3` y `com.slashmobility.PruebaCirculo4` se pueden ver ejemplos de herencia.

■ Ejercicio de reutilización:

- En un ejercicio anterior se realizó la conceptualización de la Universidad y Estudiante.
- Extender el ejemplo de la siguiente manera:
 - La Universidad, en lugar de tener estudiantes, tiene Personas.
 - Las Personas pueden ser de tipo Estudiante o Trabajador
 - Un Trabajador puede ser o bien un Profesor, un Becario, o un Administrativo
 - Definir atributos que se crean convenientes a cada uno de ellos

■ Constructores Herencia: *Super*

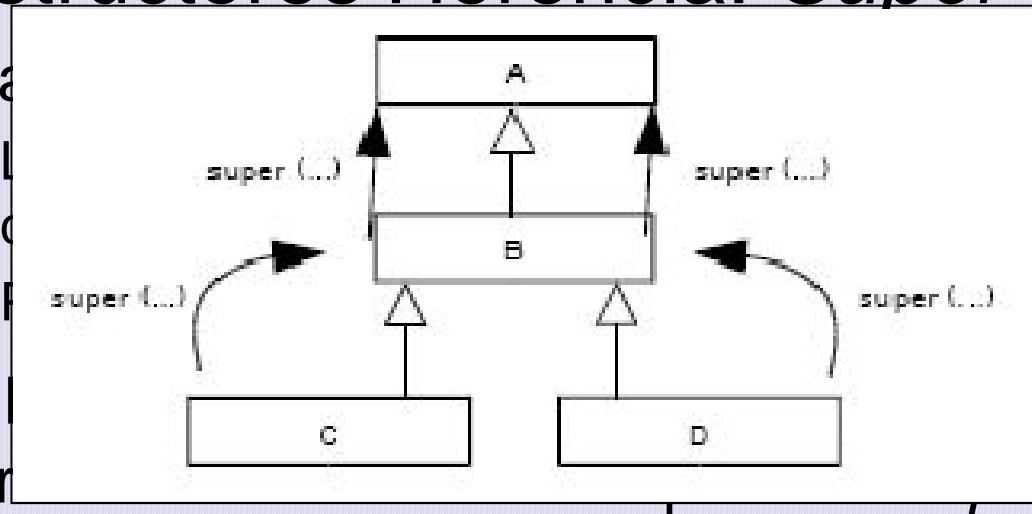
□ Cuando

■

■

□ La

□ Par



de Derivada

atributos de la

se Base.

tor del padre.

()

```

class ClaseA
{
    private int a;
    public ClaseA()
    {
        a = 0;
    }
    public ClaseA(int v)
    {
        a = v;
    }
}
    
```

```

class ClaseB extends ClaseA
{
    private int b;
    public ClaseB()
    {
        super();
        b = 0;
    }
    public ClaseB(int v1,int v2)
    {
        super(v1);
        b = v2;
    }
}
    
```

■ Constructores Herencia: *Super (II)*

- La inicialización de los atributos de una superclase en el constructor de una subclase presenta varios inconvenientes serios:
 - Resulta redundante. La superclase tiene ya un constructor que hace ese trabajo. ¿Por qué repetir código entonces?
 - Si la clase tiene una larga lista de ascendientes, entonces el constructor sería muy largo
 - La superclase puede tener una inicialización compleja, y la inclusión del código de inicialización en la subclase puede requerir un conocimiento excesivo de la superclase por parte del implementador

■ Constructores Herencia: *Super (II)*

- El procedimiento correcto consiste en realizar una llamada al constructor de la superclase para que realice la inicialización de los atributos heredados

```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        super (aPeriodoSegs);
        msg = aMsg;
    }

    public String leerMsg () { return msg; }
}
```

■ Herencia: *Final*

- Para terminar, es posible impedir la herencia a partir de una clase declarándola como **final**
- **Prudencia**, ya que puede restringir en exceso la extensión y reutilización de las clases del sistema en el futuro

```
import java.lang.Runtime;
import java.io.IOException;

final public class TPEjecucion extends TareaPeriodica {
    String cmd;

    public TPEjecucion(String aCmd, int aPeriodoSegs) {
        super (aPeriodoSegs);
        cmd = aCmd;
    }

    String leerCmd () { return cmd; }
}
```

- **Herencia: Adición, redefinición y anulación**
 - La herencia en sí no sería tan interesante si no fuera por la posibilidad de adaptar en el descendiente los miembros heredados
 - **Adición:** Trivialmente el descendiente puede añadir nuevos atributos y operaciones que se suman a los recibidos a través de la herencia
 - **Redefinición:** Es posible redefinir la implementación de una operación heredada para adaptarla a las características de la clase descendiente. También es posible cambiar el tipo de un atributo heredado
 - **Anulación:** Cuando un atributo u operación heredados no tienen utilidad en el descendientes, pueden ser anulados para impedir su utilización

- Herencia: Redefinición de métodos
 - Redefinir = Cambiar el comportamiento de un método heredado
 - **OJO** Distinto a la sobrecarga
 - No consiste en hacerse un método nuevo.
 - Ejemplo
 - Uno de los métodos que estamos redefiniendo y heredados de la clase object es:
 - *public String toString()*.

■ Herencia: Redefinición()

```
public class TPReloj extends TareaPeriodica {

    public TPReloj () {
        super (60);
    }

    public String leerHora () {
        Calendar cal = new GregorianCalendar ();
        return cal.get (Calendar.HOUR_OF_DAY) + ":" + cal.get (Calendar.MINUTE);
    }

    public void ejecutarTarea () {
        Calendar cal = new GregorianCalendar ();
        int min = cal.get (Calendar.MINUTE);

        if (min == 0 || min == 30)
            System.out.println ("Hora: " + cal.get (Calendar.HOUR_OF_DAY) + " " - min);
    }
}
```

```
public class TPEjecucion extends TareaPeriodica {
    String cmd;

    public TPEjecucion(String aCmd, int aPeriodoSegs) {
        super (aPeriodoSegs);
        cmd = aCmd;
    }

    String leerCmd () { return cmd; }

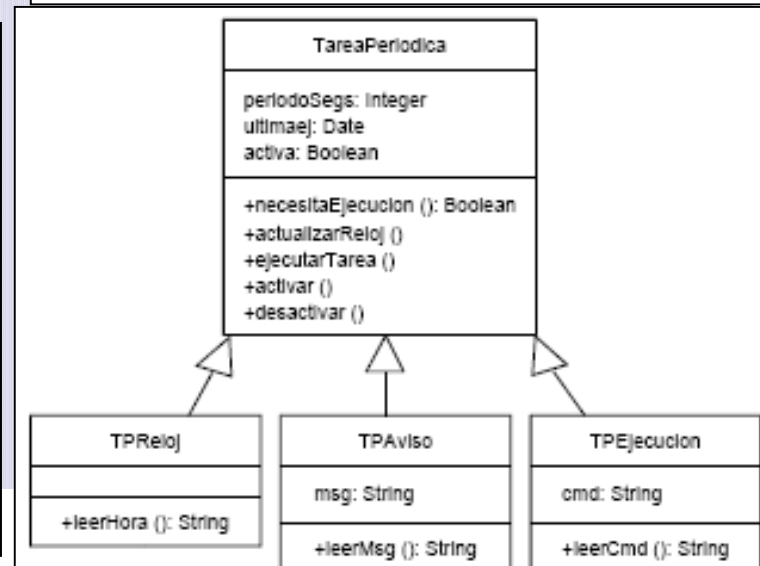
    public void ejecutarTarea () {
        try {
            Runtime.getRuntime ().exec (cmd);
        }
        catch (IOException e) {
            system.out.println ("Imposible ejecutar comando: " + cmd);
        }
    }
}
```

```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        super (aPeriodoSegs);
        msg = aMsg;
    }

    public String leerMsg () { return msg; }

    public void ejecutarTarea () {
        System.out.println ("ATENCIÓN AVISO: " + msg);
        desactivar ();
    }
}
```



■ Herencia: Redefinición: Ejemplo

- Cada tarea ejecuta ahora su función, aunque la llamada es aparentemente la misma

```
public class AppGestorTareas {
    public static void main (String[] args) {
        TareaPeriodica tp = new TareaPeriodica (5);
        TPAviso tpa = new TPAviso ("Estudiar Programación Avanzada I", 60);
        TPEjecucion tpe = new TPEjecucion ("rm -/tmp/*", 3600);

        while (!tp.necesitaEjecucion ())
            System.println ("Esperando ejecución de tarea periódica...");
        tp.ejecutarTarea ();

        while (!tpa.necesitaEjecucion ())
            System.println ("Esperando ejecución de aviso...");
        tpa.ejecutarTarea ();

        while (!tpe.necesitaEjecucion ())
            System.println ("Esperando ejecución de comando...");
        tpe.ejecutarTarea ();
    }
}
```

■ Herencia: Anulación *Atributo*

- En Java es posible impedir el acceso a un **atributo** redeclarándolo en una subclase como privado o protegido, según sea el nivel de protección que se desee

```
public class TPAviso extends TareaPeriodica {  
    String msg;  
  
    // Impedir el acceso desde el exterior y las subclases  
    // al atributo activa  
    private boolean activa;  
  
    public TPAviso(String aMsg, int aPeriodoSegs) {  
        super (aPeriodoSegs);  
        msg = aMsg;  
    }  
}
```

■ Herencia: Anulación *método*

- Sin embargo, Java no permite redefinir una operación haciendo su nivel de acceso más restrictivo
- Una solución parcial consistiría en redefinirla como vacía o incluyendo un código que impida su utilización

```
public class TPAviso extends TareaPeriodica {
    String msg;

    public TPAviso(String aMsg, int aPeriodoSegs) {
        super (aPeriodoSegs);
        msg = aMsg;
    }

    public void activar () {}

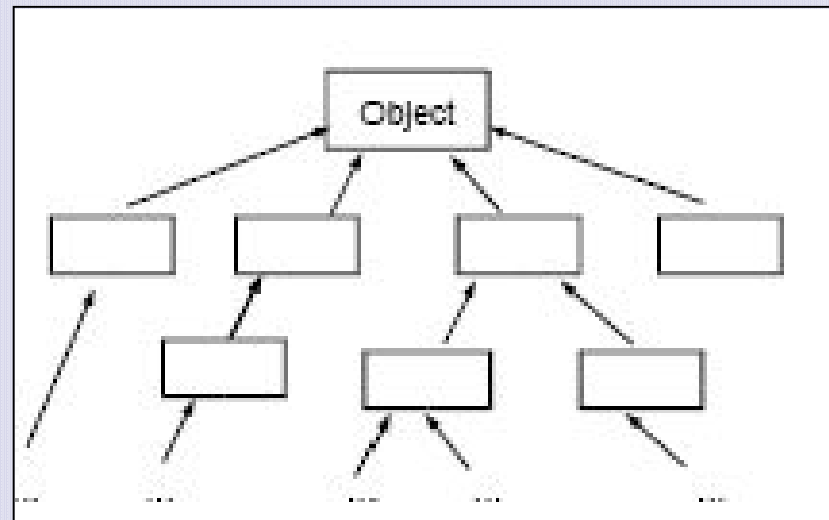
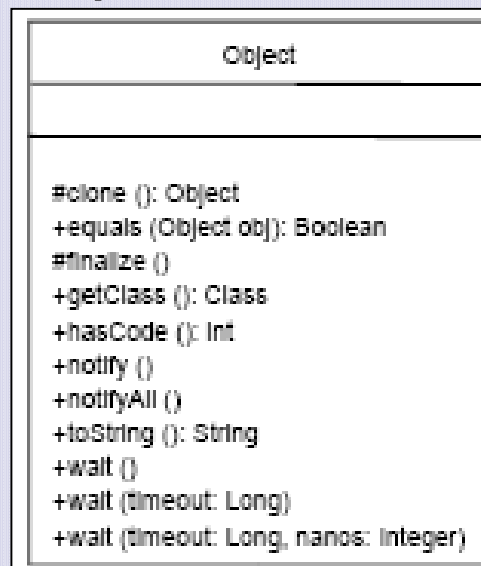
    public void desactivar () {
        System.out.println ("Error: llamada a operación privada");
        System.getRuntime ().exit (1);
    }

    public String leerMsg () { return msg; }

    public void ejecutarTarea () {
        System.out.println ("ATENCIÓN AVISO: " + msg);
        desactivar ();
    }
}
```

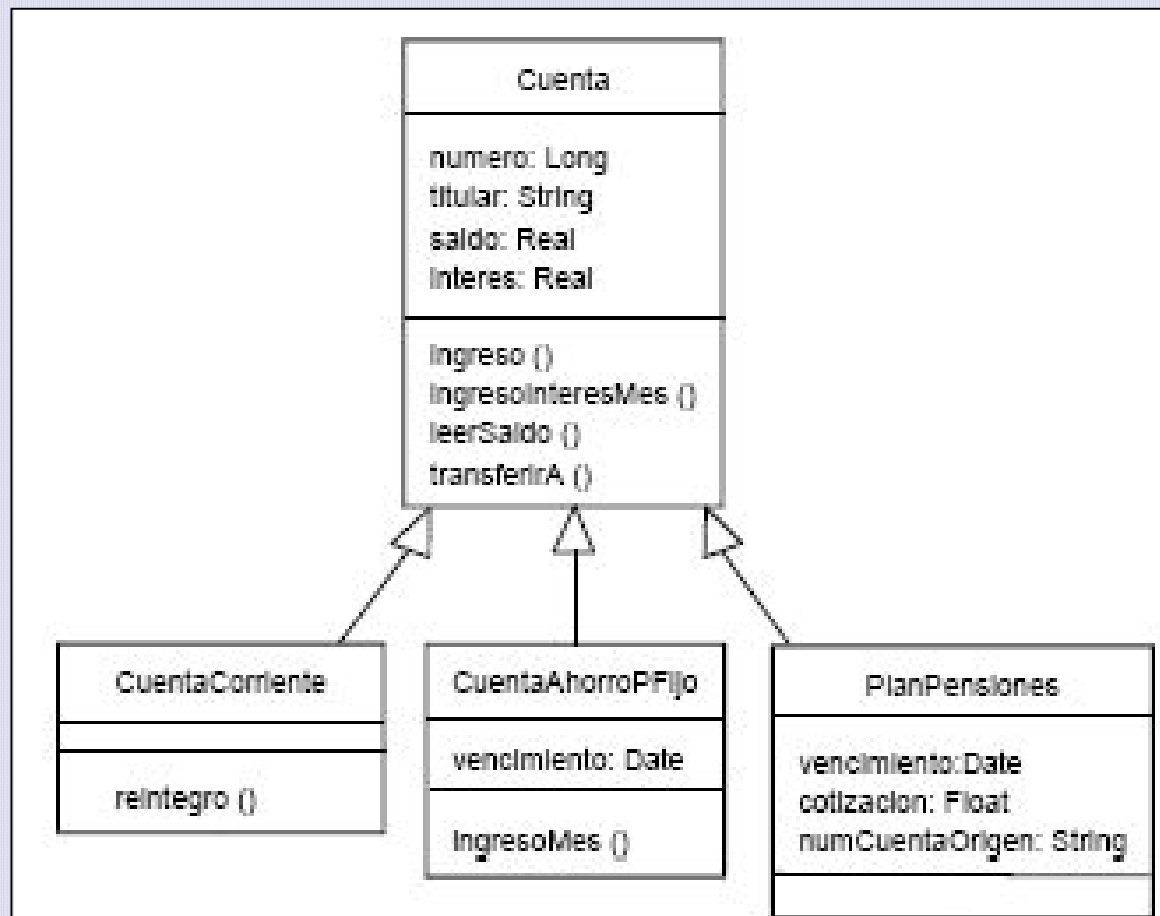
■ Herencia: Object

- Todos las clases en Java heredan en última instancia de Object. Incluso si creamos una clase independiente, Java la hace heredar implícitamente de *Object*.



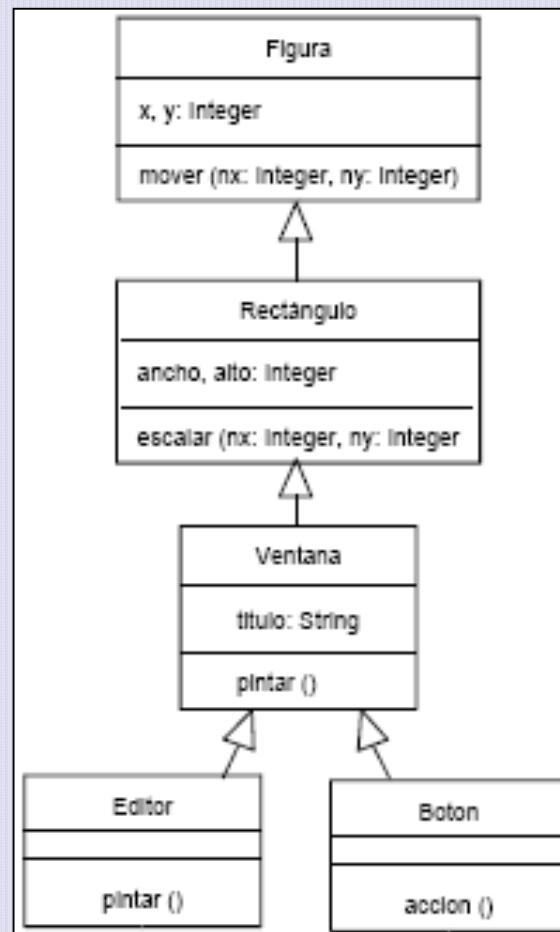
■ Ejemplos Herencia

Distintos tipos de cuentas bancarias



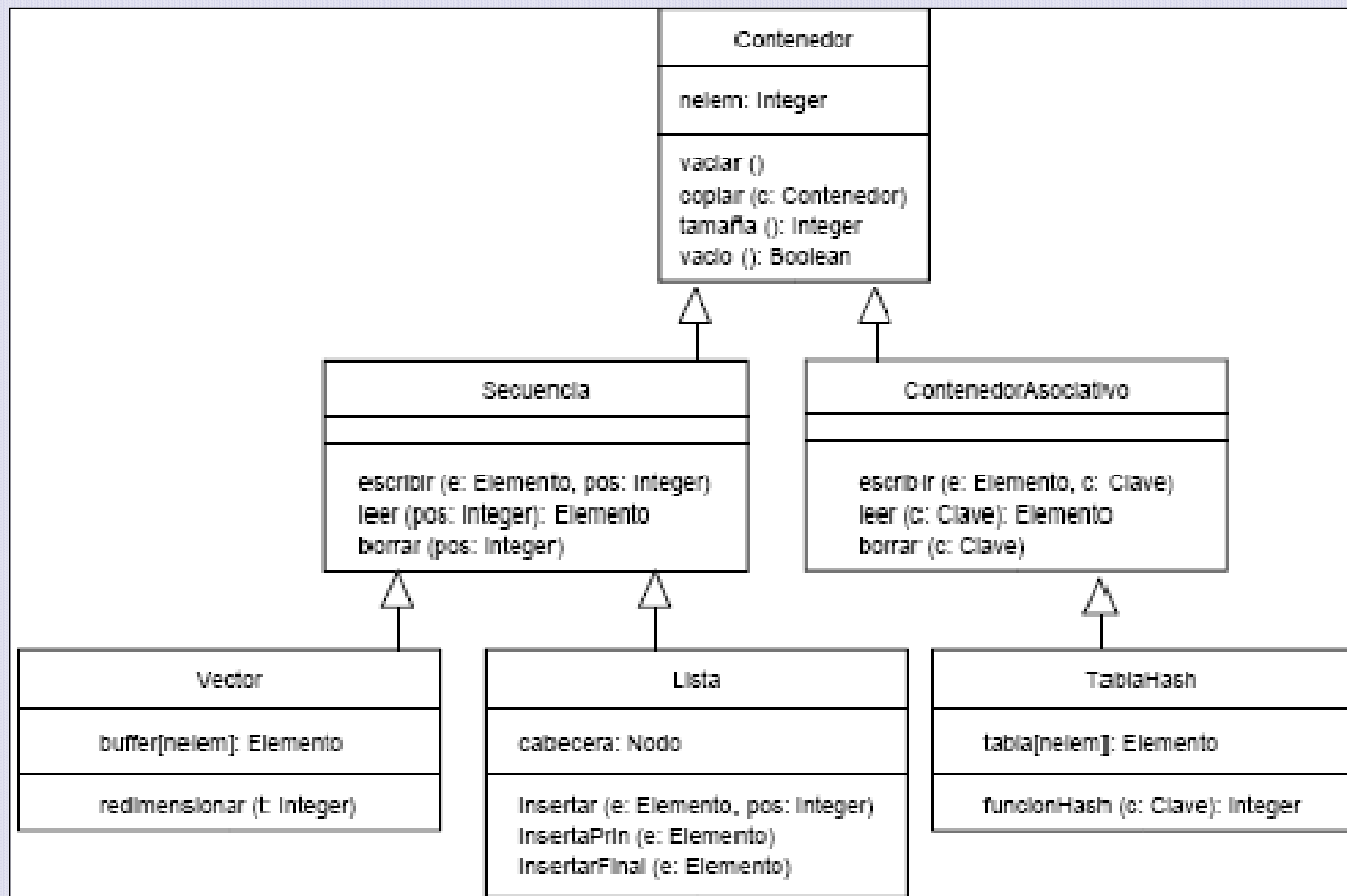
■ Ejemplos Herencia

Elementos de una interfaz de usuario



■ Ejemplos Herencia

Estructuras de datos





Conclusiones

1. Programación Orientada a Objetos
2. Introducción y Sintaxis Java
3. Sentencias Control Flujo
4. POO en Java
5. **Relaciones entre Objetos**
6. Polimorfismo, abstracción e interfaces
7. Excepciones
8. Conceptos avanzados

- Relación de Composición
- Relación de asociación
- Relación de Uso
- Herencia
 - Tipos de herencia
 - Sintaxis
 - Constructores = Super
 - Redefinición de métodos