



Departamento de Programación  
Facultad de Informática  
Universidad Nacional del Comahue



# Programación Concurrente



*Presentación y Repaso*

# Presentación

## Programación concurrente (PC)

Area: Programación Especializada

### Correlativas:

- Introducción a la Programación
- Programación Orientada a Objetos

### Docentes: Docentes:

- Silvia Amaro,
- Nadina Martínez Carod,
- Marcela Leiva,
- Pablo Quiroga

### Horarios: 4 hs semanales

- Teoría: **Viernes 10:30 – 12:30 hs.**
- Práctica: **Viernes 16 -18 hs.**
- Consultas:
  - Silvia: lunes 14- 15 hs.**
  - Nadina: martes 11- 12 hs**

# Condiciones de Acreditación

- Un examen parcial a mitad del cuatrimestre, con su correspondiente recuperatorio
- Un trabajo práctico obligatorio con entrega a fin de cuatrimestre, y defensa
- Otras actividades individuales y grupales (cuestionarios - desarrollo de ejercicios – etc)
- Promoción: próximamente

# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Vectores
- Wrappers
- Excepciones

# Ejemplo: Clase Persona

## Persona

- nombre

### Constructoras

+ Persona()  
+ Persona(String n)

### Observadoras

+ getNombre():String  
+ getDatos():void

### Modificadoras

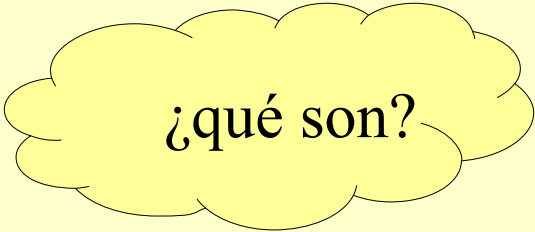
+ setNombre(String n) void

### Propias del tipo

+ mismoNombre(Persona p): boolean

# Ejemplo: Clase Persona

```
public class Persona {  
    private String nombre;  
    public Persona( ) {  
        this.nombre = null;  
    }  
    public Persona(String n) {  
        this.nombre = n;  
    }  
    public void setNombre(String nuevoNombre) {  
        this.nombre = nuevoNombre;  
    }  
    public String getNombre( ) {  
        return this.nombre;  
    }  
    public String getDatos( ) {  
        return ("Nombre: " + this.getNombre());  
    }  
    public boolean mismoNombre(Persona otraPersona) {  
        return  
(this.getNombre().equalsIgnoreCase(otraPersona.getNombre()));  
    }  
}
```



¿qué son?

## Diagrama UML de la clase derivada Estudiante

### Persona

```
- nombre
+ Persona()
+ Persona(String n)
+ getNombre():String
+ getDatos():String
+ setNombre(String n) void
+ mismoNombre(Persona p): boolean
```



es una

### Estudiante

```
- legajo // no mutable
```

#### Constructoras

```
+ Estudiante(int l)
+ Estudiante(int l, String n)
```

#### Observadoras

```
+ getLegajo():int
+ getDatos(): String
```

#### Modificadoras

```
// modifica nombre desde la clase base
```

#### Propias del tipo

```
+ esIgual(Estudiante e): boolean
+ aCadena(): String
```

# Redefiniendo constructores

```
public class Estudiante extends Persona {  
    private int legajo;  
  
    public Estudiante(int leg)  
    {  
        super ();  
        this.legajo = leg;  
    }  
}
```

- Palabra clave **extends** crea la clase derivada desde la clase base, usando herencia
- La clase Estudiante establece dos constructores, uno donde se inicializa al atributo legajo con el argumento leg
  - **super** es la primera acción en un constructor de una clase derivada.
  - Si no estuviese, Java lo incluye automáticamente
  - **super()** invoca al constructor por defecto de la clase base



# Redefiniendo constructores

```
public class Estudiante extends Persona {  
    private int legajo;  
    public Estudiante(int leg, String nom)  
    {  
        super(nom) ;  
        this.legajo = leg;  
    }  
}
```

- Este constructor pasa el parámetro `nom` al constructor de la clase base
- Utiliza el segundo parámetro para inicializar la variable de instancia que no está en la clase base.

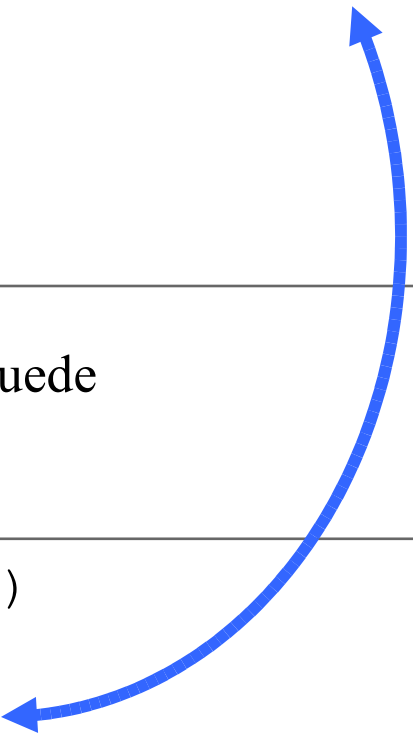
# Redefiniendo constructores

- La clase `Estudiante` tiene un constructor con dos parámetros: `String` para el atributo `nombre` y el atributo `legajo` de tipo `int`

```
public Estudiante(int legajoNuevo, String
nombreNuevo)
{
    super(nombreNuevo);
    this.legajo = legajoNuevo;
}
```

- El otro constructor dentro de `Estudiante` puede ser escrito invocando al constructor con dos argumentos dentro de la misma clase

```
public Estudiante(int leg)
{
    this(leg, null);
}
```



# Agregando atributos

```
private int legajo;
```

- En la clase Estudiante se agrega el atributo **legajo**
- Estudiante tiene **dos** atributos:
  - El atributo **legajo** (propio)
  - El atributo **nombre** (que heredó desde Persona)

```
public class Estudiante extends Persona {
    private int legajo;
    public Estudiante(int leg) {
        super( );
        this.legajo = leg;
    }
    public Estudiante(int leg, String nombreInicial) {
        super(nombreInicial);
        this.legajo = leg;
    }
    public int getLegajo() {
        return this.legajo;
    }
    public void setLegajo(int legajoNuevo) {
        this.legajo = legajoNuevo;
    }
    ...
}
```

```
public String getDatos() {  
    return("Nombre: " + this.getNombre( )  
           + "\nLegajo: "+ this.getLegajo());  
}  
  
public boolean esIgual(Estudiante otroEstudiante) {  
    return (this.mismoNombre(otroEstudiante)  
           && (this.getLegajo() == otroEstudiante.getLegajo()));  
}  
  
}
```

# Probando la clase Estudiante

```
public class TestEstudiante
{
    public static void main(String[] args)
    {
        Estudiante s = new Estudiante(123);

        s.setNombre("Juan Pablo");
        ....
    }
}
```

# La clase Object

- Java tiene una clase base de todas las clases
  - Object es ancestro de todas las clases
  - Cada clase es un descendiente de Object
  - Se puede escribir código de métodos con parámetros de tipo Object que puede ser reemplazado por cualquier clase.
- La clase Object tiene métodos que heredan todas las clases, por ejemplo equals y toString
- Cuando se necesita recuperar un objeto específico, se utiliza casting

# Otro ejemplo: Clase Empleado

```
public class Empleado extends Persona
```

```
{
```

```
    private int nroEmpleado;
```

```
    public Empleado()
```

```
{
```

```
        super();
```

```
        nroEmpleado= 0;
```

```
}
```

```
...
```

Utilizar palabra clave **extends**

Primer acción del constructor

**super()** llama al constructor por defecto de la clase padre

Persona



Empleado



# Constructor clase Empleado

```
public class Empleado extends Persona  
{
```

```
.....
```

```
public Empleado(String nuevoNombre, int nuevoNroEmpleado)
```

```
{
```

```
    super(nuevoNombre);
```

```
    nroEmpleado = nuevoNroEmpleado;
```

```
}
```

```
...
```

Pasa el parámetro nuevoNombre al constructor de la clase base

Usa un segundo parámetro para inicializar la variable de instancia que no está en la clase base

Persona

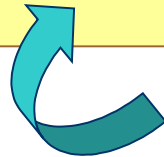


Empleado

# Constructor usando *this*

Empleado tiene un constructor con dos parámetros: *String* para el atributo nombre e *int* para el atributo número de empleado

```
public Empleado (String nuevoNombre, int nuevoNroEmpleado)
{
    super(nuevoNombre);
    nroEmpleado = nuevoNroEmpleado;
}
```



Otro constructor dentro de Empleado que llama al constructor con dos argumentos nombreInicial (String) y 0 (int),

```
public Empleado (String nombreInicial)
{
    this(nombreInicial, 0);
}
```

# Resumen de Constructores

- Los constructores pueden llamar a **otros constructores**
- Se debe utilizar **super** para invocar a un constructor de la clase padre
- Se debe utilizar **this** para invocar a un constructor dentro de la clase
- Cualquiera de las dos opciones debe ser la **primera acción** realizada por el constructor
- Si se quiere invocar a ambos se debe utilizar **this** para llamar a un constructor con **super**

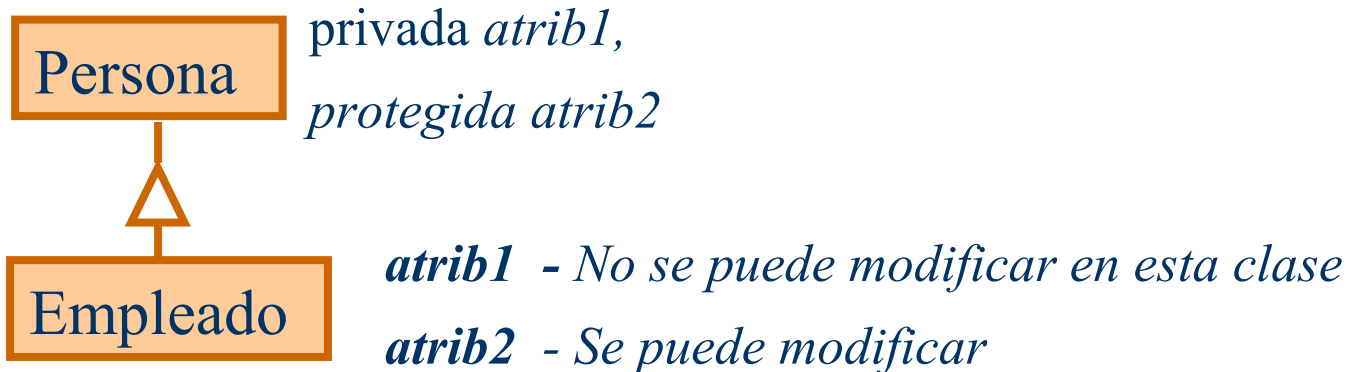
# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Vectores
- Wrappers
- Excepciones

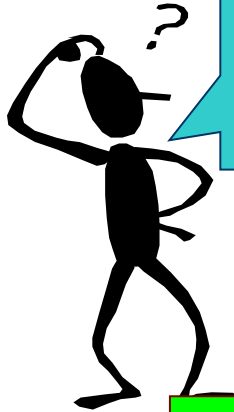


# Visibilidad

- Variables de instancia
  - Públicas --- acceso fuera del ámbito de la clase
  - Privadas --- acceso sólo dentro de la clase
  - Protegidas --- acceso dentro de la clase y sub-clases



# Visibilidad



*Qué variables se utilizan en las subclases*

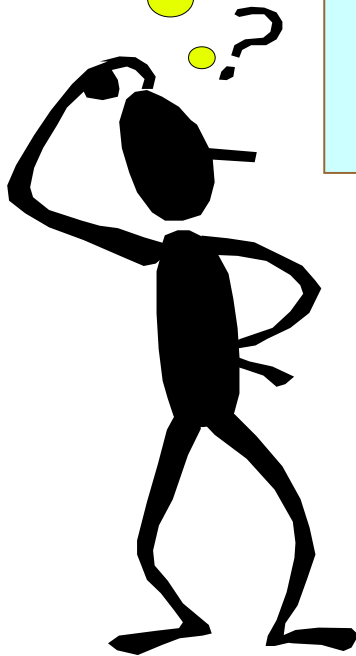
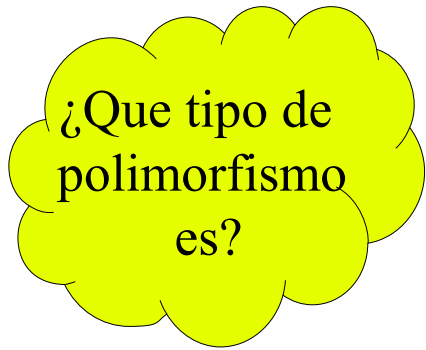
Las variables privadas no están disponibles en las subclases.

Las variables protegidas **están** disponibles en las subclases.



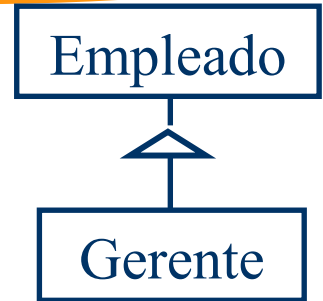
Los métodos privados no son heredados!

# Clase Empleado y Gerente

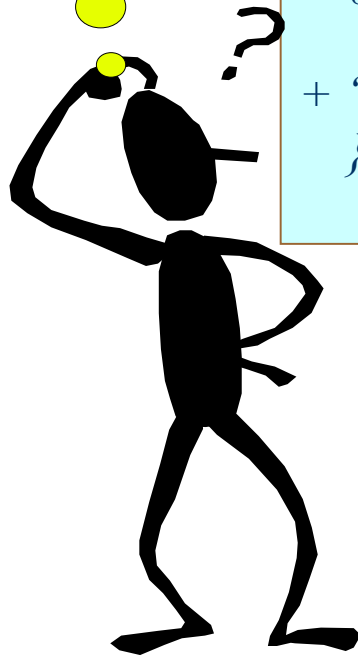
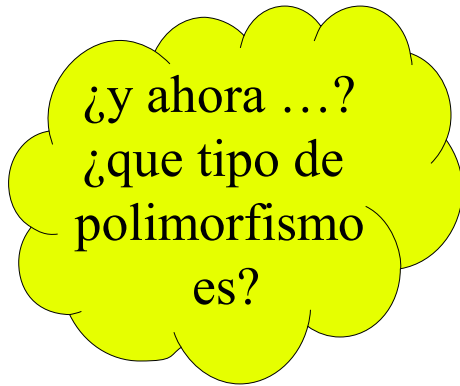


```
public class Empleado
{
    private String nombre;
    private Double salario;
    ....
    public String getDatos()
    {
        return "Nombre " + nombre +
            "\n" + "Salario: " + salario;
    }....
```

```
public class Gerente extends Empleado
    private String depto;
    ....
    public String getDatos()
    {
        return super.getDatos +
            "\n Gerente de : " + depto;
    }...
```

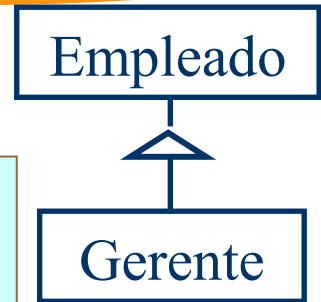


## Otra implementación de Empleado y Gerente



```
public class Empleado
{
    protected String nombre;
    protected double salario;
    ....
    public getDatos()
    {
        return "Nombre " + nombre + "\n"
        + "Salario: " + salario;
    }....
```

```
public class Gerente extends Empleado
    private String depto;
    ....
    public getDatos()
    {
        return "Nombre " + nombre + "\n" +
        "Gerente de : " + depto;
    }...
```





## Redefinicion/sobreescritura

Los métodos redefinidos no pueden ser menos accesibles

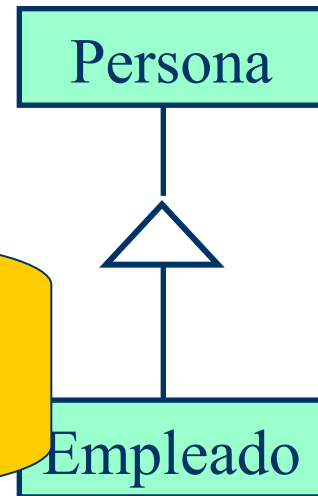


```
public class Padre{  
    public void haceAlgo1() {}  
}  
  
public class Hijo extends Padre {  
    private void haceAlgo1() {} // ILEGAL  
}  
  
public class UsarAmbos {  
    private void haceAlgo1() {}  
    Padre p1 = new Padre();  
    Padre p2 = new Hijo();  
  
    p1.haceAlgo1();  
    p2.haceAlgo1();  
}
```

# Tipos en Java

```
Empleado jefe = new Empleado();  
Persona p = jefe;
```

Se puede asignar un objeto instancia de una subclase a cualquier variable del tipo de la superclase



Empleado es miembro de ambas clases: Persona y Empleado  
El tipo de Empleado es tanto Empleado como Persona

```
Persona p = new Persona();
```

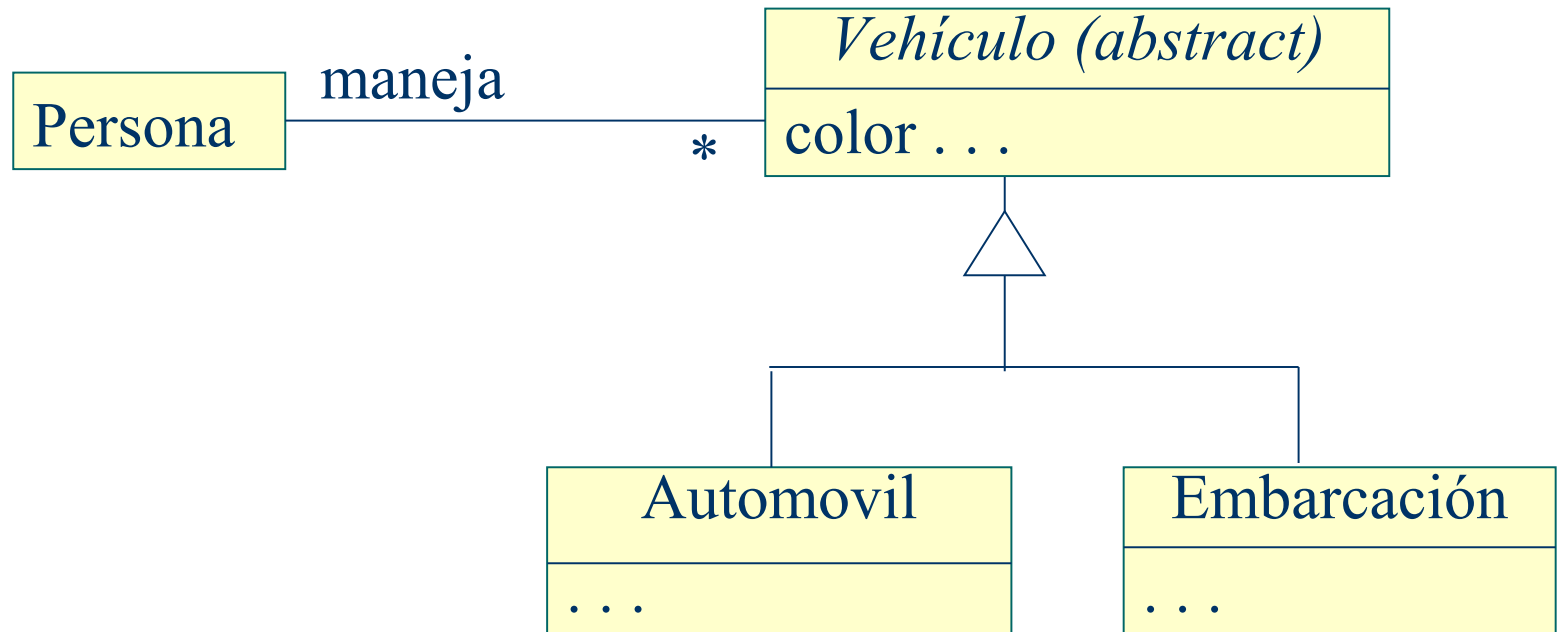
~~**Empleado jefe = p;**~~

No se puede asignar a una variable del tipo de la subclase un objeto instancia de la superclase

# Resumen

- Una clase derivada hereda métodos de la clase base y (a través de algunos de ellos) accede a sus atributos
- Una clase derivada puede tener atributos y métodos adicionales
- El constructor de una clase derivada debe invocar al constructor de la clase base
- Si una clase redefine un método de la clase base, la versión en la clase derivada reemplaza la de la clase base
- Las variables de instancia y los métodos privados de una clase base no pueden ser accedidos directamente en la clase derivada
- Si A es una clase derivada de la clase B, entonces A es miembro de ambas clases, A y B

# Clases abstractas



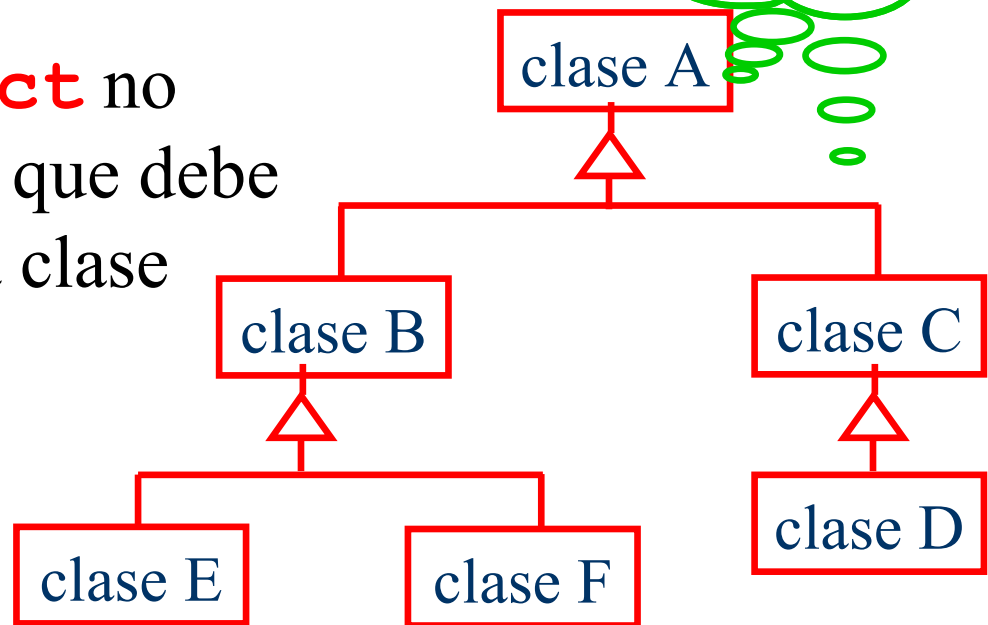
**Los objetos reales que la persona conduce son instancias de una de las subclases concretas**

# Clases Abstractas

- En el ejemplo, la clase Vehículo no fue pensada para instanciar objetos de clase Vehículo sino como clase base para otras clases derivadas.
- Hay métodos que deben estar definidos en Vehículo (de manera abstracta) pero sólo se ejecutan los de las clases derivadas
- Podemos definir un método **abstract** sin necesidad de desarrollarlo.

# Clases Abstractas (II)

- Un método **abstract** no puede ser usado sino que debe ser redefinido en una clase derivada.



- Si una clase tiene al menos un método **abstract** definido, la clase no puede estar instanciada

# Clases Abstracta – Clase Concreta

- No puede tener instancias

- Describe atributos y comportamiento **común** a sus subclases

- Puede tener **métodos abstractos**



- Puede tener instancias

- Todos los **métodos** están **implementados**. Puede tener implementaciones diferentes en sus subclases

- No puede tener ningún **método abstracto**.

# Resumen clase abstracta y concreta

- Las *clases abstractas* no pueden ser instanciadas.
- Si una clase tiene al menos un *método abstracto* entonces es abstracta.
- En una jerarquía de clases tipo árbol una clase hoja no puede ser una *clase abstracta*.
- En Java una clase abstracta se reconoce por la palabra clave *abstract*.
- Una clase puede ser abstracta si la unión de sus subclases conforman el universo.



# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- Vectores
- Wrappers
- Excepciones

# Interfaces en Java

- Una interfaz en Java es una clase abstracta pura, donde todos los métodos son **abstract** (ninguno está implementado)
- También puede contener atributos pero siempre **static** y **final**

# Interfaces en Java (II)

- Para crear una interfaz se utiliza la palabra clave **interface** en lugar de **class** y todos sus métodos son **public**.
- Para indicar que una clase implementa los métodos de una interfaz se utiliza la palabra clave **implements** (puede implementar más de una)

# Ejemplo interface

Definir la  
interfaz

```
public interface MiInterfaz {  
    /* ejemplo de interfaz */  
  
    public boolean metodo1 (int i);  
    public void método2();  
}
```

Utilizar la  
interfaz

```
public class MiClase implements MiInterfaz {  
    /* la clase debe implementar todos los  
    métodos definidos en la interfaz*/  
  
    ....  
    public boolean metodo1 (int i){  
        return true  
    };  
    public void método2(){  
        .....  
    };  
}
```

# Resumen de Conceptos

- Separa los conceptos de *subclase y subtipo*.
- Todas las clases son derivadas de una clase raiz, si no hay clase padre explicitada se utiliza *Object*.
- Si bien Java soporta *herencia simple*, también soporta múltiples interfaces es por esto que en algunas bibliografías aparece como herencia múltiple (aunque no la tiene).
- Una clase puede extender *múltiples interfaces*.  
Ej. `class graphicalObject implements Storable, Graphical {...};`
- Utiliza las palabras claves *abstract* (para clase abstracta), *final* (indica que una clase no puede tener subclases).

# Temario

- Herencia en Java
- Visibilidad en Java
- Interfaces
- **Vectores**
- Wrappers
- Excepciones

# Java – Colecciones

**Array - (arreglos)**

En qué se parece a un objeto?



Cuando es usado como argumento de un método, pasa por referencia

Pero, cuando se pasa como argumento un solo elemento del arreglo, se comporta de acuerdo al tipo del mismo

En qué se diferencia de un objeto?



Los métodos son invocados con una notación de subíndice especial

No aceptan herencia

# Estructuras Dinámicas

*Una estructura es dinámica cuando su tamaño puede crecer y decrecer en tiempo de ejecución*

- Los vectores de Java son similares a los arreglos pero son flexibles.
- Las listas enlazadas son estructuras dinámicas



# Arreglos vs. Vectores

## *Arreglos*

*No pueden cambiar de tamaño*

*Puede cambiar el valor de un elemento*

*El índice va desde 0 hasta tamaño-del-arreglo*

*Se declara de un tipo en especial  
Ej: `String[] a;` `int[] v;` etc*

## *Vectores*

*Cambian dinámicamente de tamaño*

*Puede añadir y/o eliminar elementos*

*El índice va desde 0 hasta tamaño-del-vector*

*No se declara de un tipo en particular. Cada elemento del vector debe ser una referencia a un objeto*

Los vectores no son una parte de Java, están en la librería util (paquete estándar de Java)  
Al inicio de la clase que desea utilizar un vector, debemos importar el paquete  
**java.util.\***

# Uso de Vectores

Importamos el paquete *java.util* para que la clase *Vector* sea reconocida por el compilador

```
import java.util.*;  
  
public class UsaVector  
{  
    public static void main (String[] arg)  
    {  
        Vector v = new Vector();  
  
        // otras sentencias  
    }  
}
```

# Vectores: constructores

- Constructor vacío

***Vector v = new Vector();***

La dimensión inicial es 10. Si se pasa de ese elemento la dimensión del vector se duplica

- Recibe la dimensión inicial

***Vector v = new Vector(20);***

Si se pasa de la dimensión inicial, la dimensión del vector se duplica

- Recibe la dimensión inicial y cuanto crecerá si se pasa dicha dimensión.

***Vector v = new Vector(20, 5);***

Si se pasa la dimensión del vector crece en 5

# Capacidad Inicial vs. Eficiencia

- Elegir la capacidad inicial de un vector es un problema clásico de análisis de ingeniería:
  - demasiado grande gasta espacio en memoria de más
  - demasiado pequeño retarda la ejecución  
(es muy costoso re-dimensionar vectores dinámicamente)
- Solución:
  - Optimizar una a expensas de la otra
  - Buscar un equilibrio
    - Elegir un tamaño ni demasiado grande ni demasiado pequeño

# Sintaxis de Vector

Si **a** es un arreglo de `String`

```
String[] a = new String[5]
```

```
a[1] = "Hola!";
```

```
String temp = a[1];
```

Si **v** es un vector (de `Object`)

```
Vector v = new Vector(5);
```

```
v.setElementAt("Hola!", 1);
```

```
String temp =  
    (String)v.elementAt(1);
```

## Asignar

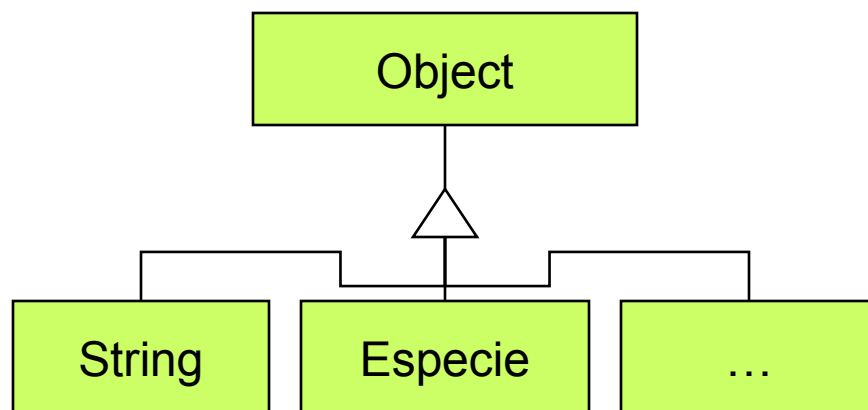
En lugar del índice entre corchetes y `=`, utiliza el método **setElementAt** con dos argumentos (valor e índice)

## Recuperar

Utiliza el método **elementAt** para recuperar el valor del elemento en una posición dada

# El tipo Object

- En Java, todas las clases predefinidas y las definidas por nosotros “son hijas” de la clase Object
- En un vector se pueden poner elementos de cualquier tipo clase
- Cuando se recuperan elementos del Vector se obtienen elementos de tipo Object



# Tipo Base de Vector

- El siguiente código produce un error:

```
Vector v = new Vector()  
String saludo = "Hola!";  
v.addElement(saludo);  
System.out.println("Longitud: "  
    + (v.elementAt(0)).length());
```

La clase Object  
no tiene un  
método llamado  
length

- El método `length`, es de la clase `String` pero Java ve el tipo de `v.elementAt(0)` como un `Object`, no `String`

# Tipo Base de Vector

- Cuando recuperamos un elemento del Vector hay que “castearlo” al tipo que insertamos

- Se puede hacer directamente

```
System.out.println ("Longitud: " +  
    ((String) v.elementAt(0)).length());
```

*Una vez aplicado el cast, el resultado es de tipo String*

- O podemos guardarlo en una variable auxiliar

```
String aux = (String) v.elementAt(0);  
System.out.println ("Longitud: " +  
    aux.length());
```



# Problemas del Vector

- El siguiente código produce un error:

```
Vector v = new Vector();  
String saludo = "Hola!";  
v.addElement(saludo);
```

```
System.out.println("Longitud: “ + (v.elementAt(0)).length());
```

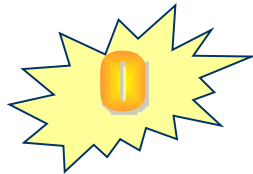
- El método *length*, es de la clase String pero Java ve el tipo de *v.elementAt(0)* como un *Object*, no *String*
- Cuando recuperamos un elemento del Vector hay que “**castearlo**” al tipo que insertamos

# Castear un elemento

```
Vector v = new Vector();  
String saludo = "Hola!";  
v.addElement(saludo);
```

```
System.out.println("Longitud: " + (v.elementAt(0)).length());
```

```
System.out.println("Longitud: " + ((String) v.elementAt(0)).length());
```



*Una vez aplicado el cast, el resultado es de tipo String*

```
String aux = (String) v.elementAt(0);  
System.out.println ("Longitud: " + aux.length());
```

# Tipo Base de Vector

- El tipo base de un Vector es siempre `Object`
  - los elementos pueden ser de cualquier tipo clase (no primitivo)
  - incluso, pueden ser de tipos clase *diferentes*
  - Buena práctica de programación
    - Es mejor tener todos los elementos del mismo tipo clase en un vector.
    - Si tenemos una mezcla de clases, que no corresponden a una jerarquía, en el mismo vector, no sabemos a qué hay que “castear” cada uno
- ¿Qué podemos hacer para almacenar elementos de tipo primitivos en un vector? (Ej: Vector de enteros, Vector de char, etc)
  - Clases Wrapper

# Métodos de Vector

- *constructores*
- métodos para trabajar como los arreglos  
*(setElementAt, elementAt)*
- métodos para agregar elementos *(addElement, insertElementAt)*
- métodos para remover elementos *(remove)*
- métodos de búsqueda *(indexOf)*
- métodos para trabajar con el tamaño y la capacidad del vector  
*(size, isEmpty)*
- un método para copiar el vector completo *(clone)*

# Métodos *addElement* y *setElement*

- *addElement* se usa para colocar los valores iniciales en los elementos del vector, el valor se agrega al final del vector e incrementa la longitud del vector en 1
- *setElementAt* asigna un valor en una posición dada, pero sólo si ha sido previamente asignado un valor con *addElement*
- Es decir
  - *addElement (elemento)* crea el elemento la primera vez
  - *setElementAt (elemento, posición)* lo modifica después

# Métodos de inserción y borrado

- *addElement* agrega un elemento al final
- *insertElementAt* agrega un elemento en cualquier posición

*v.insertElementAt(elemento, indice);*

- Si índice es igual al tamaño, entonces elemento será insertado al final (idem *addElement*).
- Si índice es más grande que el tamaño, obtendríamos un error en tiempo de ejecución (*ArrayIndexOutOfBoundsException*)
- Todos los elementos después de índice tendrán su posición incrementada en 1
- *removeElementAt* elimina el objeto en una posición dada del vector

# Método insertElementAt

0	uno
1	dos
2	tres
3	cuatro
4	cinco



0	uno
1	dos
2	tres
3	<b>nuevo cuatro</b>
4	cuatro
5	cinco
6	<b>seis</b>

Dado el vector anterior

- `v.insertElementAt("nuevo cuatro", 3);` agrega un elemento en la posición 3 y "corre" un lugar para adelante los de las posiciones 3 y 4
- `v.insertElementAt("seis", 6);` agrega al final
- `v.insertElementAt("****", 8);` da error de límite excedido

# Método removeElementAt

- removeElementAt elimina el objeto en una posición dada del vector

0	uno
1	dos
2	tres
3	cuatro
4	cinco

- Dado el vector anterior

- removeElementAt (2)  
elimina el elemento en la  
posición 2 (“tres”) y corre los  
de las posiciones superiores  
(3 y 4) un lugar para atrás

0	uno
1	dos
2	cuatro
3	cinco



# Método indexOf

- `indexOf` es equivalente al método de búsqueda de `String`
- El argumento que hay que enviar es el objeto buscado

- Dado el vector anterior

0	uno
1	dos
2	tres
3	cuatro
4	cinco

- `indexOf("dos")` devuelve 1
- `indexOf("XXX")` devuelve -1

# Otros métodos de Vector

- `size()` devuelve la cantidad de elementos en el vector

- Ej: `v.size()` devuelve 5

- `isEmpty()` devuelve true si la cantidad de elementos es 0

- Ej: `v.isEmpty()` devuelve false

Sea el vector v

0	uno
1	dos
2	tres
3	cuatro
4	cinco

- `clear()` vacía el vector (remueve todos los elementos)
  - Ej: `v.clear()` deja el vector v vacío

# Arreglos vs. Vectores

<u>Arreglos</u>	<u>Vectores</u>
<p><i>Negativo</i></p> <ul style="list-style-type: none"><li>● El tamaño se fija cuando se declara</li><li>● Almacenamiento ineficiente: puede usar un arreglo parcialmente lleno, pero el espacio reservado es el tamaño completo.</li><li>● Si uno o más valores necesitan ser agregados al pasar el tamaño máximo, el arreglo necesita ser re-declarado</li></ul>	<p><i>Positivo</i></p> <ul style="list-style-type: none"><li>● El tamaño no es fijo</li><li>● Almacenamiento eficiente: un vector guarda en memoria <i>aproximadamente</i> el espacio que necesita</li><li>● Si uno o más valores necesitan ser agregados pasando el tamaño máximo, el tamaño del vector se incrementa automáticamente</li></ul>
<p><i>Positivo</i></p> <ul style="list-style-type: none"><li>● Ejecución eficiente (más rápido)</li><li>● Permite elementos de tipos primitivos</li></ul>	<p><i>Negativo</i></p> <ul style="list-style-type: none"><li>● Menos eficiente en la ejecución</li><li>● Los elementos deben ser de tipos clase (no tipos primitivos)</li></ul>

# Clonación

- El método clone se utiliza para hacer una copia de un vector pero su tipo de retorno es Object, no Vector por lo que hay que castearlo

```
Vector v = new Vector(10);
```

```
Vector otroV;
```

~~otroV = v;~~



Hace a otroV otro nombre para el vector v (hay una sola copia del vector y ahora tiene dos nombres)

```
Vector otroV = (Vector)v.clone();
```

castear



Crea una copia con nombre otroV

# Otros métodos

- size() devuelve la cantidad de elementos en el vector
  - *v.size()*
- isEmpty() devuelve true si la cantidad de elementos es mayor que 0
  - *v.isEmpty()*
- clear() vacía el vector (remueve todos los elementos)
  - *v.clear()* deja el vector v vacío
- indexOf() es equivalente al método de búsqueda de String, el argumento que hay que enviar es el objeto buscado
  - *v.indexOf(elemento)* deja el número de elemento o -1

# Tamaño Vs. Capacidad

- Debemos tener en claro la diferencia entre *capacidad* y *tamaño* de un vector:
  - ***capacidad*** es el tamaño del vector declarado
    - el número máximo actual de elementos
  - ***tamaño*** es el número actual de elementos que están siendo usados
    - el número de elementos que contienen valores válidos, no basura
    - recordemos que los vectores agregan valores en índices sucesivos
- Los bucles que recuperan los elementos del vector deben estar limitados por el valor del tamaño, no de la capacidad, para evitar leer los valores basura

# Características de los Vectores

- Un vector incrementa automáticamente su tamaño si sus elementos van más allá de su capacidad actual
- Pero no decrementa automáticamente su tamaño si sus elementos son eliminados
- El método *trimToSize* encoge la capacidad de un vector a su tamaño actual por lo que no quedará espacio extra
  - El espacio se reduce al que está siendo usado
- Usar *trimToSize* cuando el vector no necesitará capacidad extra más tarde aumenta la eficiencia en general

# Como iterar sobre un Vector

- ... continuará (esperar hasta la proxima semana)
- Opcion
  - INVESTIGAR
    - Distinto tipo de colecciones en Java
    - Iteradores
    - Genéricos