Programación Orientada a Objetos

Material de Estudio

Nuria Medina Medina, José Parets Llorca, María del Mar Abad Grau, Fernando Molina Ortiz, María José Rodríguez Fórtiz

Universidad de Granada

ETSIIT Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Departamento de Lenguajes y Sistemas Informáticos

La **Programación Orientada a Objetos** es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas de computadora.

Sin embargo, durante el estudio de este libro le recomendamos que estire el paradigma orientado a objetos, que lo expanda al máximo, que lo exprima, en definitiva que trate de aplicarlo también en su vida cotidiana.

Observe el universo que le rodea consciente de que todo lo que puede percibir a través de sus sentidos son objetos. Analice, también, el universo que tiene dentro y comprenda que cualquier imagen que puede formar en su mente es también un objeto.

Cualquier cosa, idea o incluso persona transfórmelo en un objeto, véalo como un objeto. La silla donde se sienta mientras lee estas líneas, el bolígrafo con el que escribe, el profesor que le explica en clase, el compañero que tiene sentado a su lado en el aula, todo son objetos.

Una vez que todo ante usted se haya puesto el traje de objeto, compare unos objetos con otros y extraiga la esencia de cada objeto: ¿qué tienen en común todas las sillas?, ¿qué tienen en común todas las personas?, bombardéese con preguntas de este tipo y en base a sus respuestas agrupe los objetos en clases, ah… y póngales nombre a las clases.

Ya conocidas las clases, analice qué responsabilidades tiene cada clase, esto es fundamental. ¿Qué responsabilidades tiene usted mismo como estudiante de Programación Orientada a Objetos? Puede empezar por ahí...

Para cada responsabilidad defina un método, una forma concreta de realizar esa responsabilidad. Observe que un estudiante de programación orientada a objetos no resuelve de igual forma la responsabilidad de programar que un estudiante de programación declarativa. ¿Por qué? Porque pertenecen a clases diferentes.

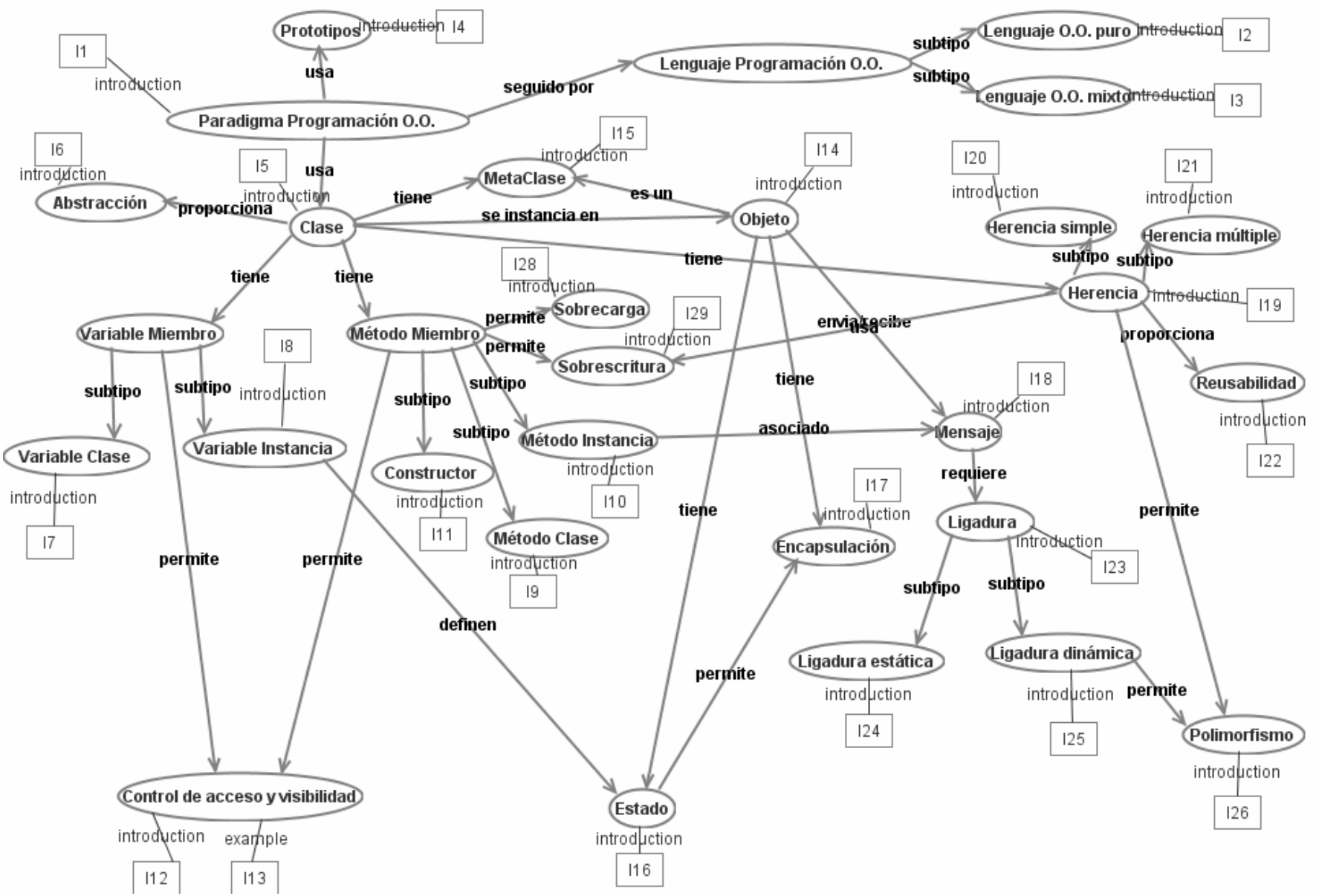
Al concretar los métodos se dará cuenta de que el objeto necesita manejar cierta información para poder ejecutar el método. Pregúntese si esa información pertenece al propio objeto o a un objeto diferente. En el primer caso habrá identificado atributos del objeto, en el segundo caso habrá detectado la necesidad de colaborar con otros objetos.

Es aquí donde el universo se integra y se torna un TODO. Los objetos colaboran entre sí, ¡esa es la clave! Ésta es la dinamo que no permite al engranaje detenerse: objetos que trabajan juntos para conseguir un objetivo.

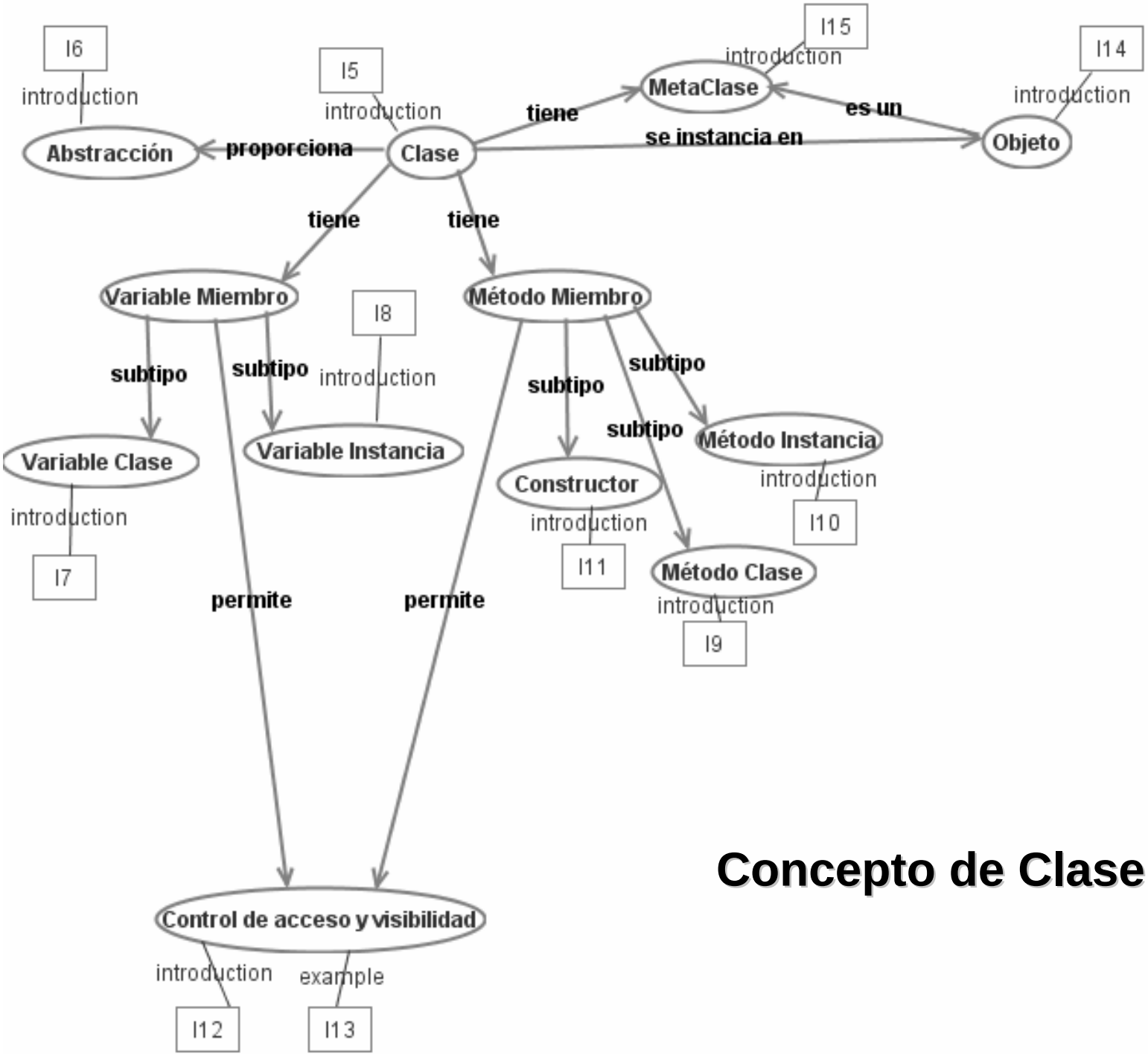
Después habrá mucho más: herencia, polimorfismo, interfaces, encapsulación, etc.

Pero esto es sólo el prefacio del libro, para conocer el resto tiene todo un curso por delante. Aprovéchelo!

Conceptos de Programación Orientada a Objetos

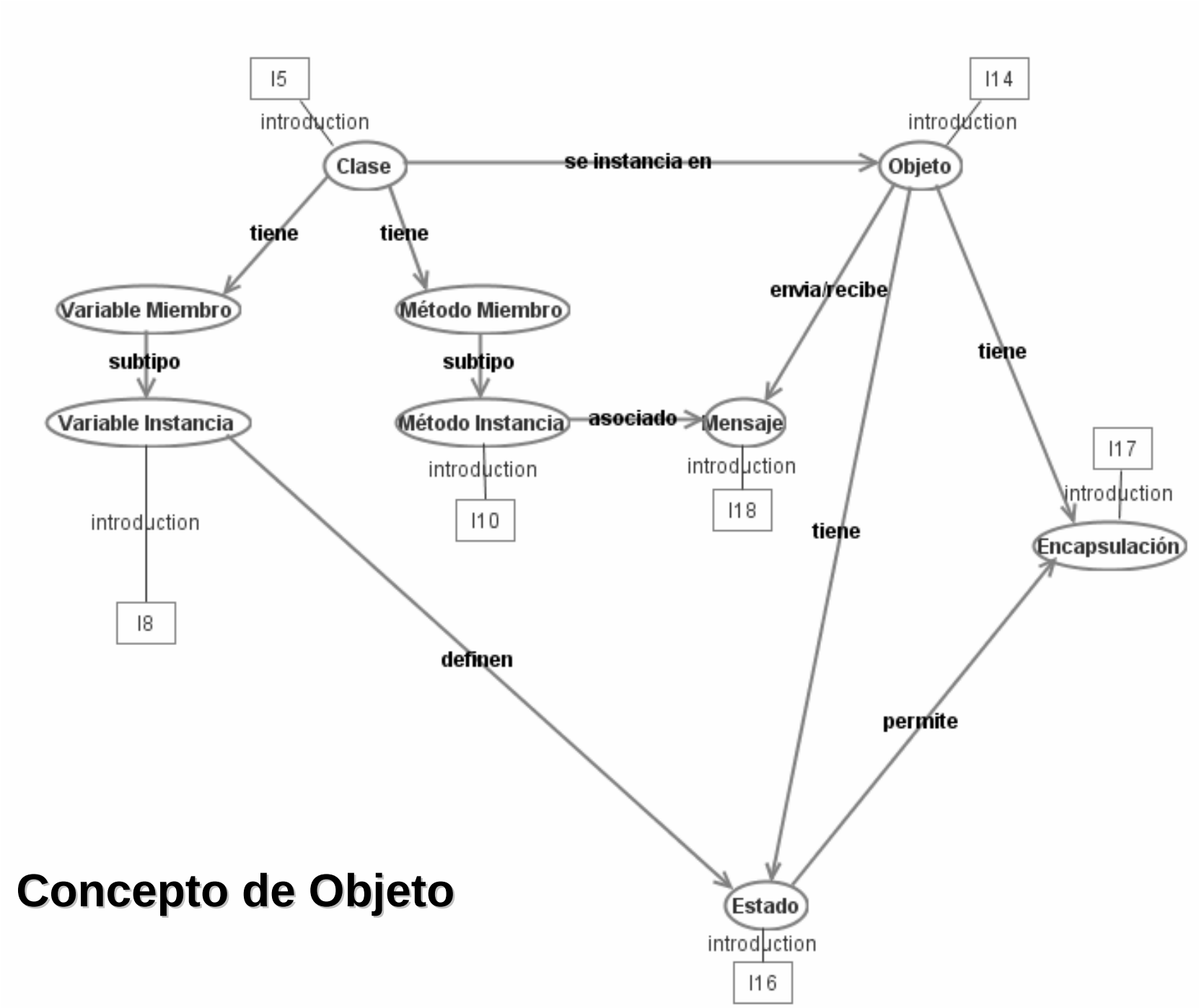


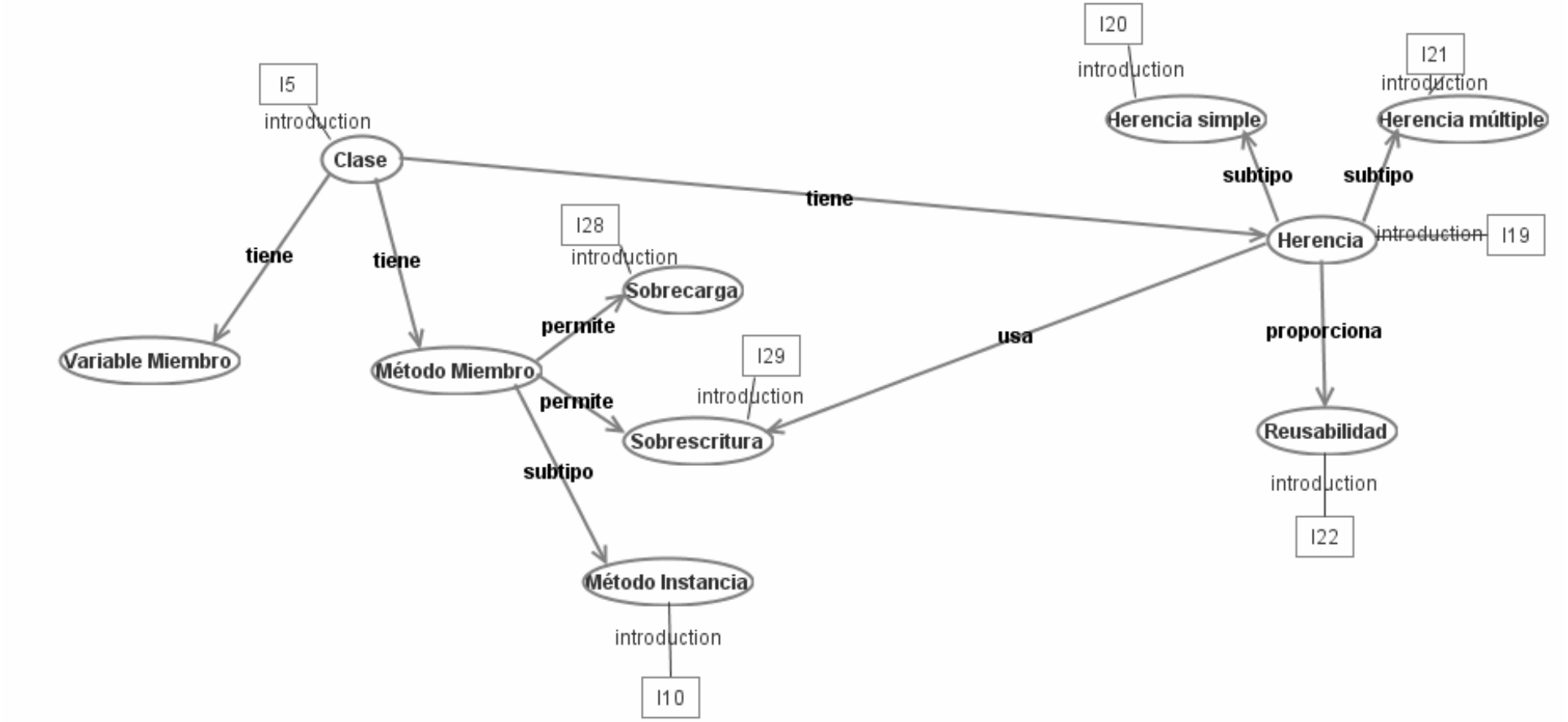
Mapa Conceptual de Programación Orientada a Objetos



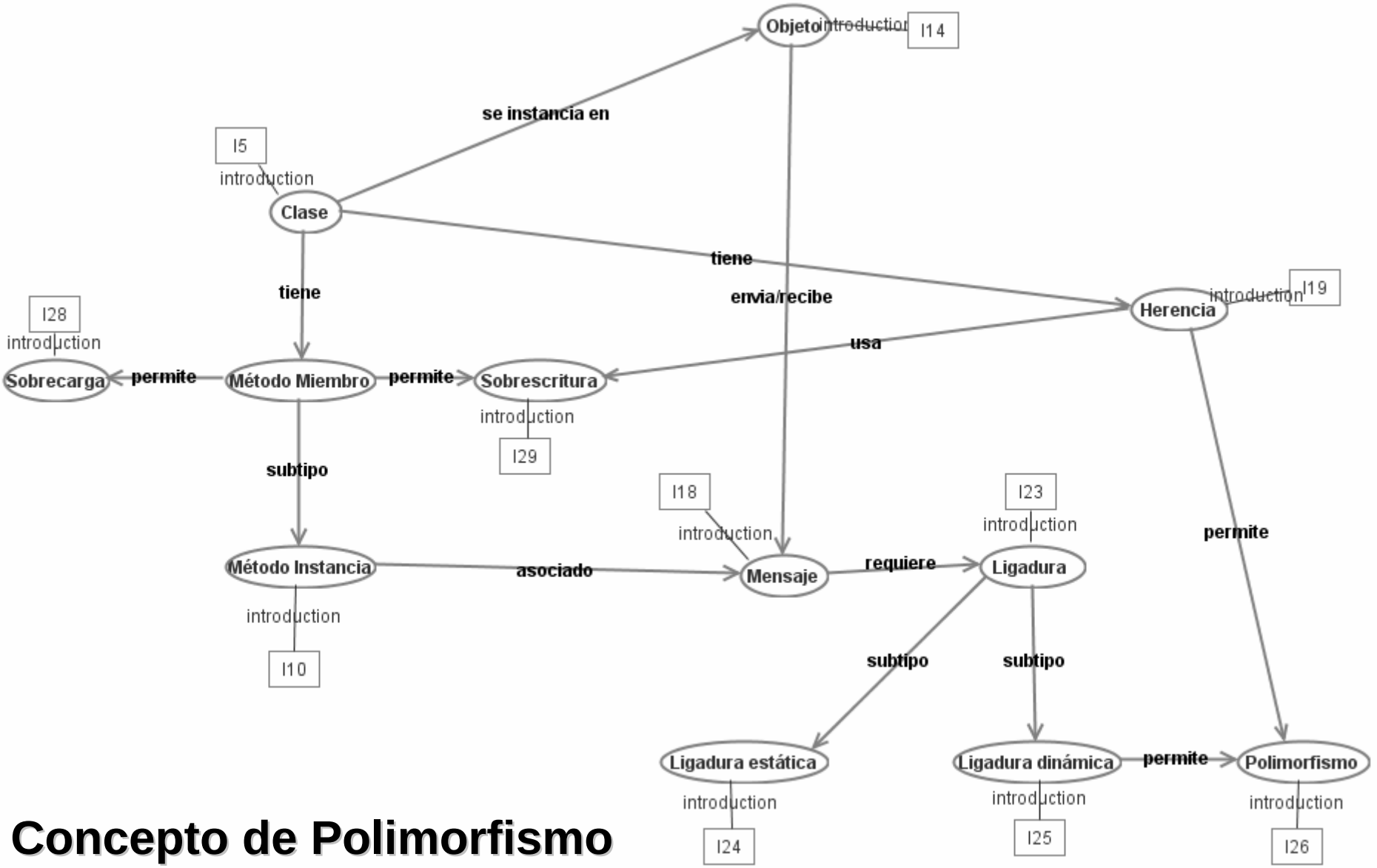
Concepto de Clase

Concepto de Objeto





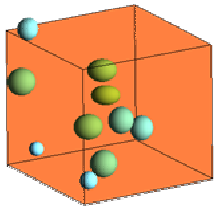
Concepto de Herencia



Concepto de Polimorfismo

Tabla de Contenidos

Tema 1



Lección 1

Concepto de Programación y Diseño Dirigido a Objetos

Un Nuevo Paradigma de Programación

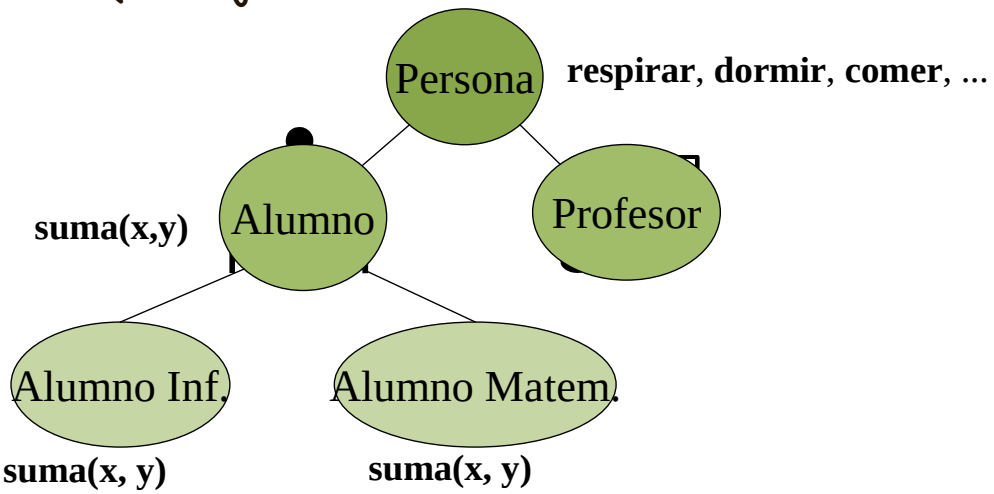
- ¿Si sé programar en C, sé programar en C++?
- Paradigmas de programación
  - ¿Qué es un paradigma?
  - Paradigmas de programación
- Resolución de problemas como en la vida real

Historia de la PDO

Años	Datos	Programas	Lenguajes
1960 - 1968	Datos	Programas	Fortran, Cobol
1967 - 1975	Estructuras de datos	Estructura de programas Programación modular	Pascal
1975 - 198...	Tipos Abstractos de Datos (TAD)		CLU, ADA, Modula 2
1978 - 199...	P.O.O. (TAD + Clases de Simula)		Smalltalk, C++, Eiffel, Java

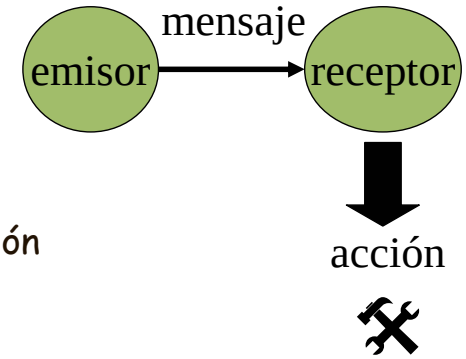
Metáfora de la P.D.O

- Programar la actividad de este aula
  - ¿Qué objetos intervienen?



Principales Conceptos

- Objeto
  - Mensaje
    - Argumentos
  - Método
  - Ocultación de información
- Clases e Instancias
- Jerarquía de Clases
  - Herencia
  - Adición
  - Refinamiento y redefinición o reemplazo



Objetos

- Protocolo o Interfaz
- Comportamiento
- Estado

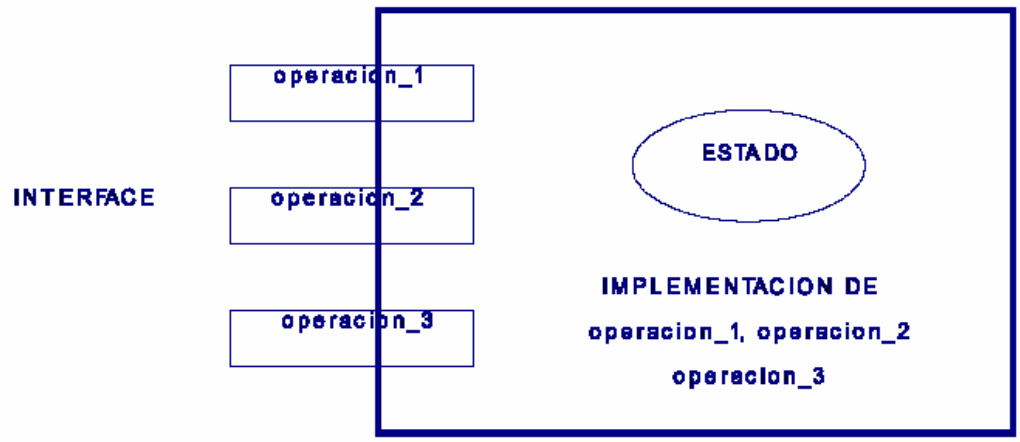
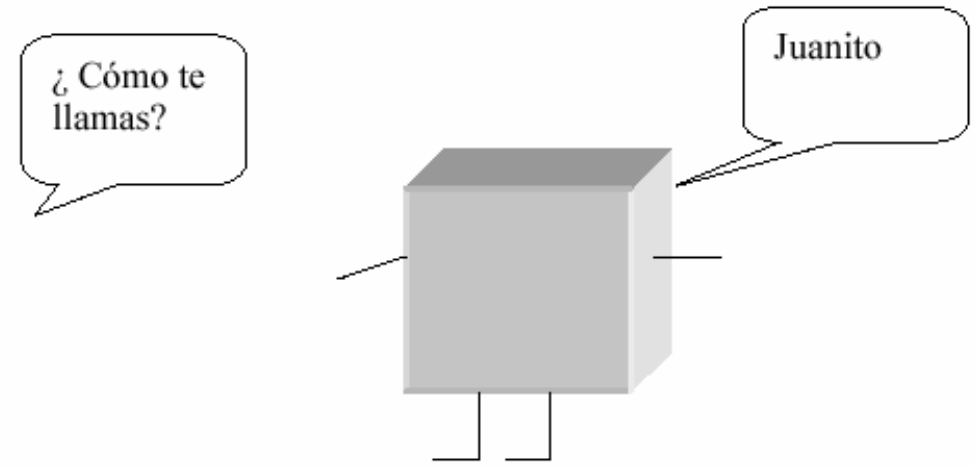


Figura 1.- Un objeto

Ocultación de Información



Mensajes

- Receptor, Petición y Argumentos

  David salta

  Ana pinta "Las Meninas"

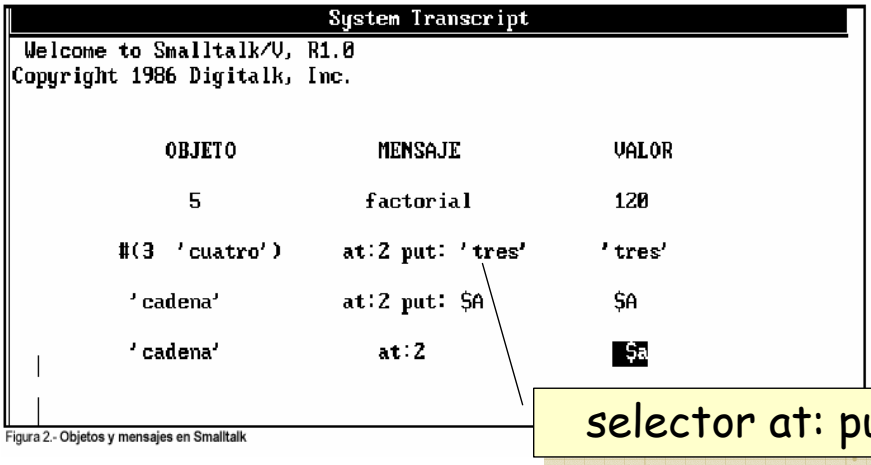


Figura 2.- Objetos y mensajes en Smalltalk

Mensaje vs Llamada a Procedimiento

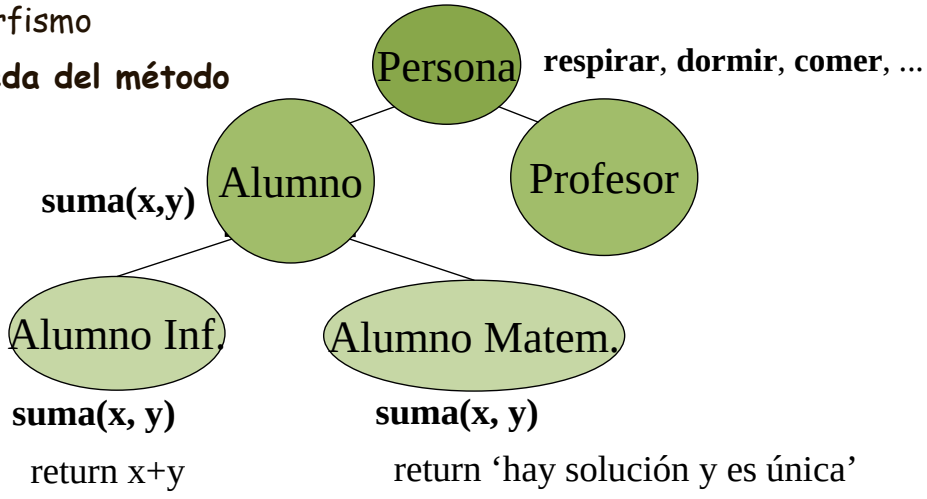
5 factorial. ⟺ *factorial(5);*

- Receptor
- Polimorfismo
- Búsqueda del método

Mensaje vs Llamada a Procedimiento

5 factorial. ⟺ *factorial(5);*

- Receptor
- Polimorfismo
- **Búsqueda del método**



Clases

Definición de una clase en JAVA:

```
public class Rectangulo {
        private Point origen;
        private Point extremo;

        public void cambiaOrigen(Point nuevoOrigen){origen = nuevoOrigen};
        public int area(){...};
}
```

Definición de una clase en Smalltalk:

```
Object subclass: #Rectangulo
 instanceVariableNames: 'origen extremo'
 classVariableNames: ''
 poolDictionaries: ''
```

Mensaje a su superclase

Instancias

Creación de objetos en JAVA:

```
Rectangulo miRectangulo;
...
miRectangulo = new Rectangulo( );
```

Sin declaración de tipo

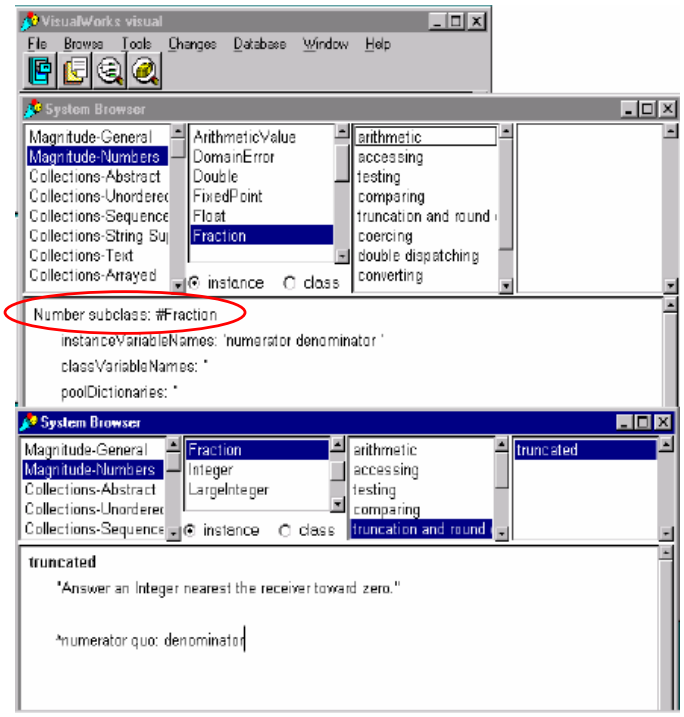
Creación de objetos en Smalltalk

```
miRectangulo := Rectangulo new.
```

Mensaje a la clase

- ¿En qué consiste Programar ?

Jerarquía de Clases



Herencia

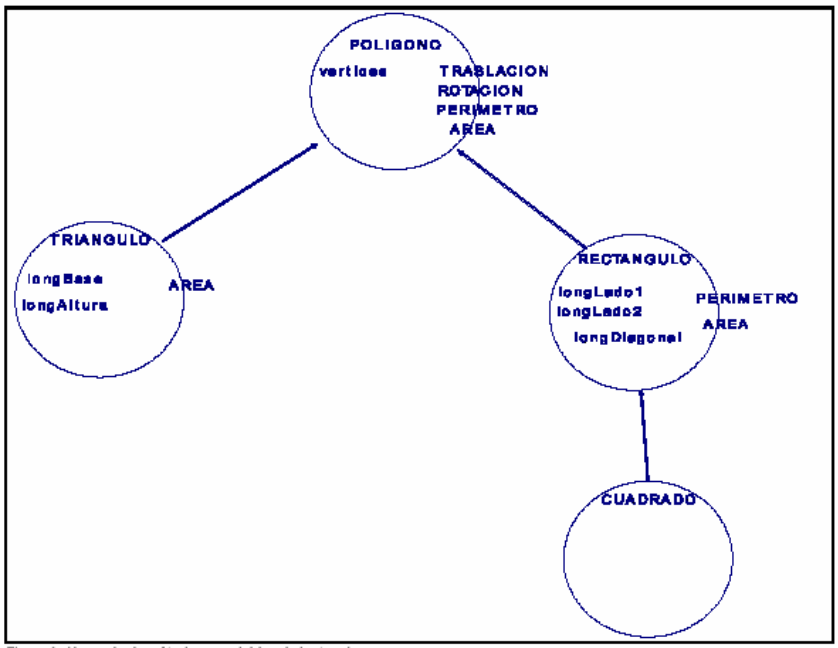


Figura 4.- Herencia de métodos y variables de instancia

Polimorfismo y enlace dinámico o tardío

El tiempo de LIGADURA o ENLACE es el momento en que se resuelven las llamadas a procedimientos y las referencias a los datos

- Estática: compilación/encuadernación

```
PROGRAM principal
 BEGIN
  ...
  rutina(a,b)
  ...
 END
```

```
PROCEDURE rutina(a,b:integer)
   BEGIN
      …
   END
```

- Dinámica: ejecución

Polimorfismo y ligadura dinámica: ejemplo

Persona:

suegra "Devuelve el nombre de la suegra"

estadoSuegra "viva/muerta"

encargarFlores

self estadoSuegra='viva'

ifTrue: [florista:= FloristaNormal new]

ifFalse: [florista:=FloristaFuneraria new].

ramo:=florista prepararFlores.

florista entregar: ramo a: (self suegra)

Polimorfismo y ligadura dinámica: ejemplo

Florista:

prepararFlores

entregar: unRamo a: unNombre

FloristaNormal (subclase florista)

prepararFlores: "prepara un ramo de rosas"

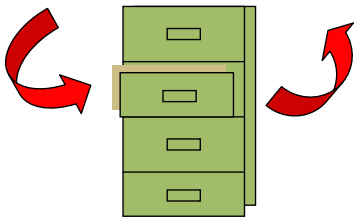
FloristaFuneraria (subclase florista)

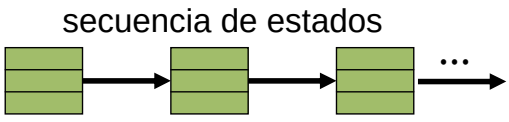
prepararFlores: "prepara una corona de flores"

```
unaPersona:= Persona new.
unaPersona encargarFlores
```

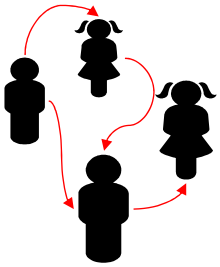

Modelos de Computación

- Tradicional (modelo de casillero)



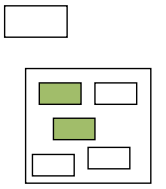


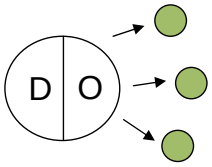
- Orientado a objetos (modelo de simulación)
  - Lenguajes mixtos (C++, Java)
  - Lenguajes puros (Smalltalk)

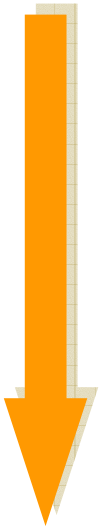


Mecanismos de Abstracción

- Objetivo: Reducir la complejidad
  - **Procedimientos**
    - Reutilización de código
  - **Módulos**
    - Ocultación de información
  - **TDA**
    - **D**atos + **O**peraciones
    - Permite instanciación
  - **Objetos**
    - Paso de mensajes
    - Sobrecarga
    - Herencia
    - Polimorfismo







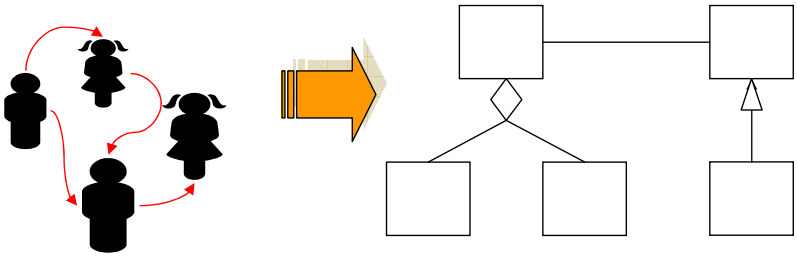
Software Reutilizable

- Componentes reutilizables

- Componentes software reutilizables
  - Dificultad en la programación convencional
  - Aislar la información esencial y genérica

Diseño Dirigido por Responsabilidades

- La responsabilidad implica un grado de independencia → Reutilización
- Consejos de diseño
  1. Pensar en la organización de un grupo de individuos



  2. Establecer el responsable de cada acción

Diseño Dirigido por Responsabilidades

- Cuidar el vocabulario del diseño (nombres de clases, variables, métodos, etc.):
  - Corto
  - Significativo
  - Pronunciable
  - Mayúscula al inicio de cada nueva palabra
  - Evitar abreviaturas
  - No usar dígitos (5 ó S, 0 ó O?)
  - Variables booleanas claramente interpretables

Diseño Dirigido por Responsabilidades

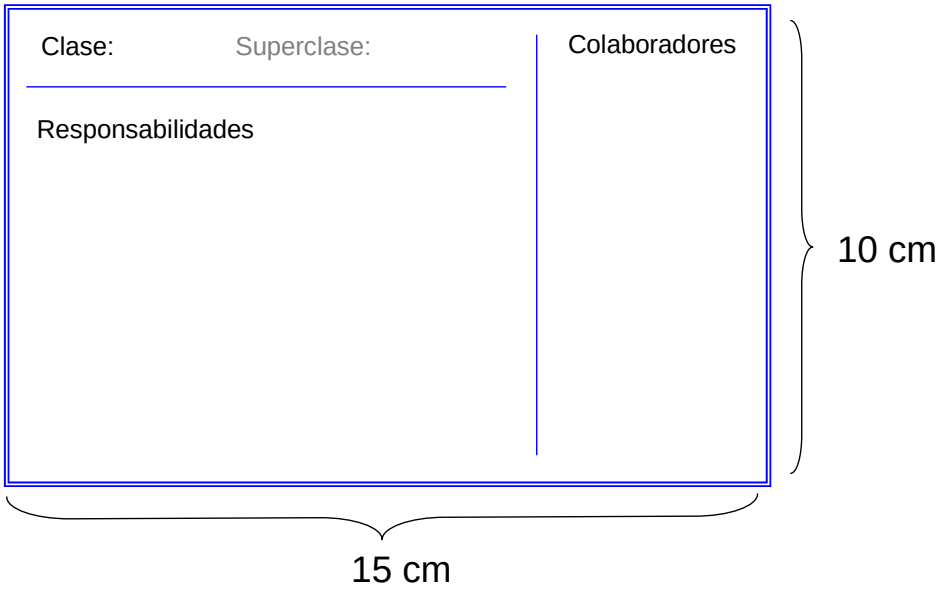
- Expresar las responsabilidades con frases cortas
  - Un verbo activo
  - Especificar qué, no cómo
- Identificar los colaboradores de cada clase
  - Clases que le prestan servicios
  - Quizá también las clases a las que presta servicios

Errores de Diseño

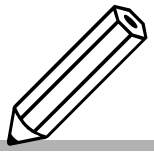
- Clases con demasiada responsabilidad
- Clases sin responsabilidades
- Responsabilidades que no se usan
- Responsabilidades desconectadas
- Responsabilidades repetidas
- Uso inadecuado de la herencia
- Violación en la encapsulación

Tarjeta CRC

Clase – Responsabilidades - Colaboradores



Haz tu Diseño

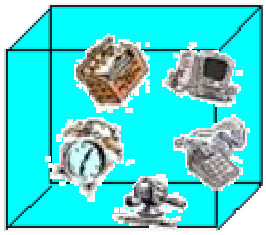


- En una **empresa** hay tres tipos de empleados:
  - **Empleados por horas** que cobran en función de las horas trabajadas durante el mes correspondiente.
  - **Empleados a comisión** que cobran un porcentaje del importe total vendido en ese mes.
  - **Empleados asalariados** que cobran un sueldo fijo al mes.
- Existen empleados asalariados especiales llamados **directores** que dirigen y organizan a otros empleados.
  - Un director no puede ser dirigido por otro director.
  - Un empleado no puede tener más de un director (puede no tener ninguno).
  - Un director puede tener más de un empleado subordinado.
- Objetivo: Gestionar la organización de los empleados y calcular la nómina de todos los empleados cada mes.

Haz tu Diseño

Clase:	Superclase:	Colaboradores
Responsabilidades		

Tema 1



Lección 2

Conceptos Básicos de los Lenguajes Dirigidos a Objetos

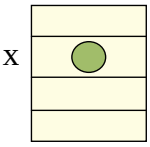
Conceptos

1. Asignación de tipos
2. Tiempo de enlace
3. Asignación de memoria

1. Asignación de tipos

- Estática: no se comprueba el tipo
- Estática fuerte: se comprueba el tipo
- Dinámica: se acepta cualquier tipo

Asignación estática y dinámica de tipos

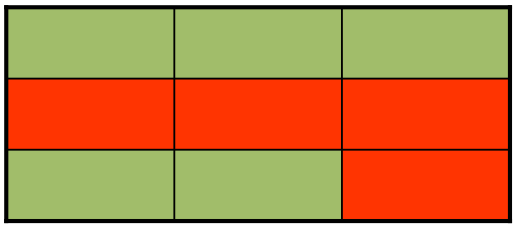


Variable

- Identificador (nombre)
- Valor (objeto)
- Tipo

Asignación

Estática Fuerte Dinámica



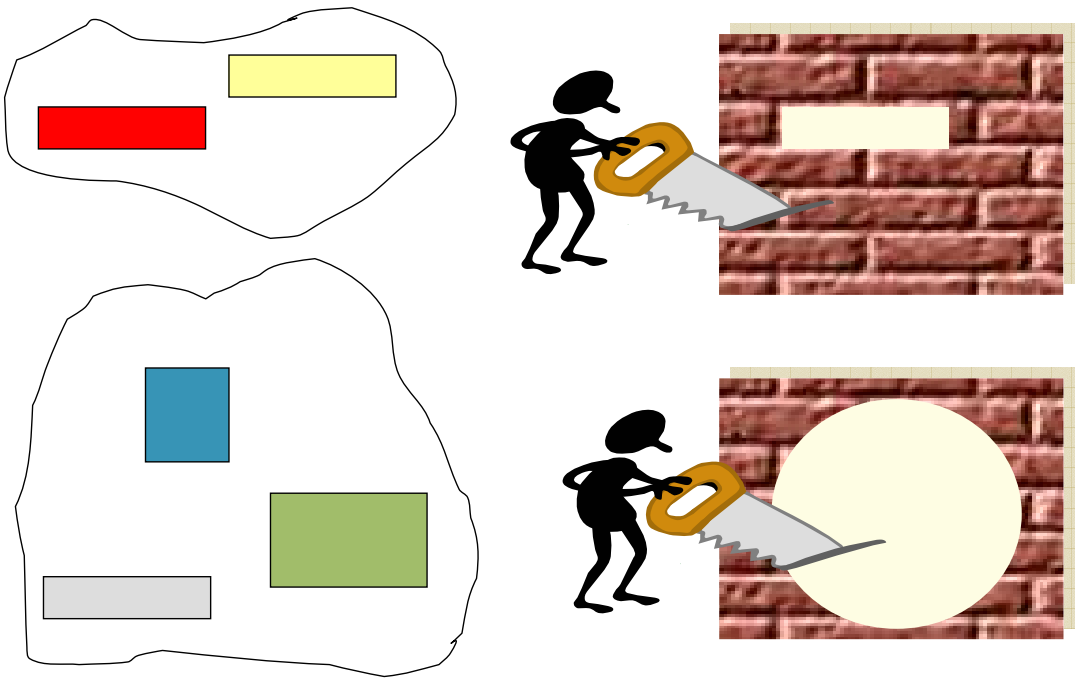
Cambia

No cambia

Asignación de tipos

- Estática
  - Pascal: estructuras CASE
  - C, C++: uniones (estructuras UNION)
- Estática fuerte
  - Pascal, C++: resto de tipos
  - Java: todos los tipos
- Asignación dinámica
  - SmallTalk

Incompatibilidad de tipos básicos



Incompatibilidad de tipos básicos en lenguajes con asignación fuerte de tipos

- Fuentes de error
  - Asignación errónea: detectada en compilación

    int n; n=3.5; // error: pérdida de precisión
  - Mal uso del casting:

    int n; n=(int) 3.5;

Ejemplo en asignación estática 'suave'

La union (C/C++) es una estructura en la que todos campos comparten las mismas posiciones de memoria.

```
union Salario {
                int mensual;
                float comision;
                };
Salario salarioEmpleado;
salarioEmpleado.mensual=1000;
salarioEmpleado.comision=0.5;
cout << salarioEmpleado.mensual;// 1056964608
```

Ejemplo en asignación estática 'suave'

La union (C/C++) es una estructura en la que todos campos comparten las mismas posiciones de memoria.

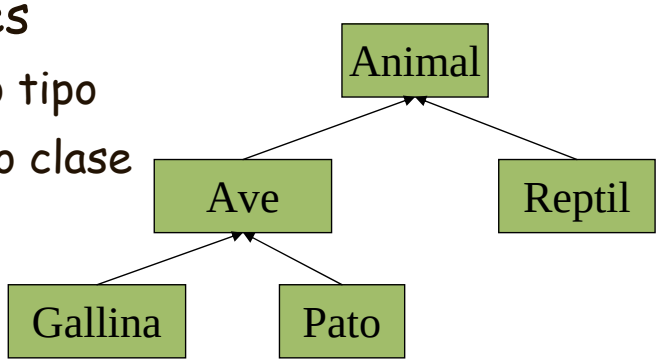
```
union Salario {
                int mensual;
                float comision;
                };
Salario salarioEmpleado;
salarioEmpleado.mensual=1000;
cout << salarioEmpleado.comision;// 1.4013e-42
```

Asignación estática de tipos en LOO: tipo y clase

- Tipo.- Dicta los objetos que podrá contener una variable. Otros nombres son:
  - Tipo de la variable
  - Tipo estático
- Clase.- Da la interpretación a los objetos. Otros nombres son:
  - Clase del valor
  - Tipo dinámico de la variable

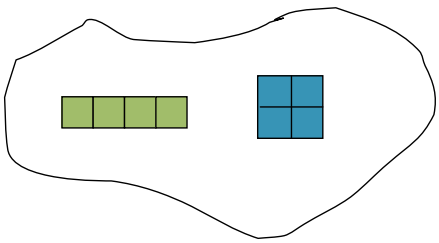
Asignación estática de tipos no básicos en LOO: tipo y clase

- Una variable puede referenciar a cualquier objeto de la clase declarada o de sus subclases
  - Tipo Estático o tipo
  - Tipo Dinámico o clase

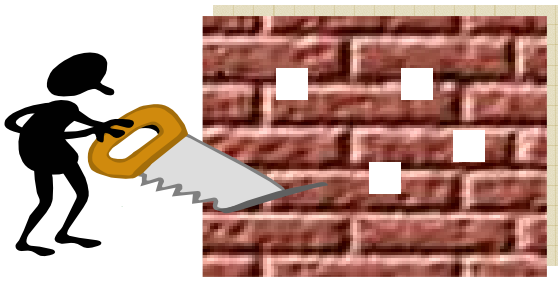



```
Animal x;
x = new Animal();
x = new Reptil();
x = new Gallina();
```

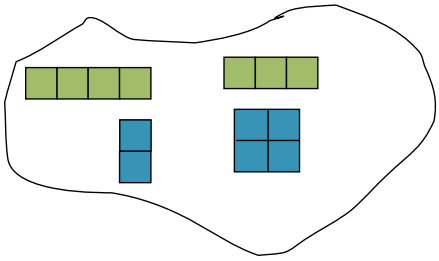

El Problema del Contenedor en los tipos (clases estáticas) en LOO con asignación estática de tipos

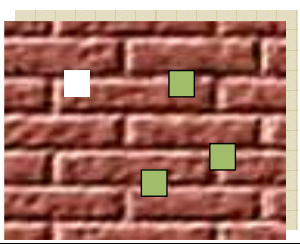


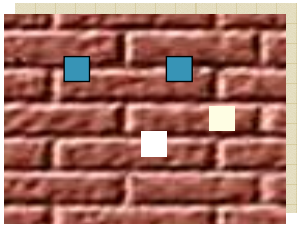
Pedazos ladrillo= métodos de la clase del valor



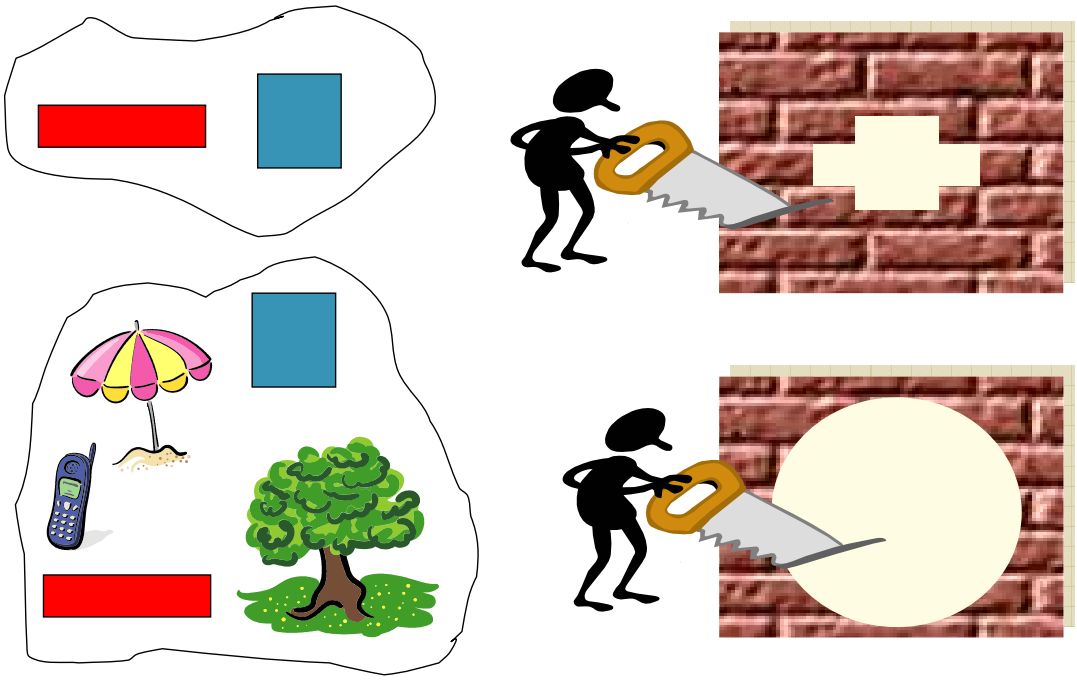
Errores:







El problema de la incompatibilidad de tipos en las clases



El problema de la incompatibilidad de tipos en las clases

```
Empleado empleado;
empleado=new EmpleadoAsalariado("Juan", 65426758, 1200);

EmpleadoAsalariado empleadoAsalariado;
empleadoAsalariado=new Empleado("Ana", 54416351);
// error de compilación: incompatibilidad de tipos
```

El Problema del Contenedor

- En LOO, los contenedores se suelen declarar como Object para poder ser usados de forma genérica (reusabilidad).
- Así, los objetos de cualquier clase pueden formar parte de ellos.
- Problema: Error al determinar la clase de un objeto del contenedor
  - Asignación estática de tipos: Errores de compilación/ejecución
  - Asignación dinámica de tipos: Error de ejecución

El Problema del Contenedor con asignación estática de tipos

```
ArrayList empresa = new ArrayList();
empresa.add(new EmpleadoAsalariado("Juan", 65426758, 1200));
empresa.add(new EmpleadoAComision("Ana", 54416351, 0.3));
Iterator itEmpr = empresa.Iterator();
while(itEmpr.hasNext()) {
    Object e =  itEmpr.next();
    String n = e.dimeNombre();
    float s = e.calculaSueldo();
    float c = e.dimeComision();
}
```

Posible solución: uso del casting

```
ArrayList empresa = new ArrayList();
empresa.add(new EmpleadoAsalariado("Juan", 65426758, 1200));
empresa.add(new EmpleadoAComision("Ana", 54416351, 0.3));
Iterator itEmpr = empresa.Iterator();
while(itEmpr.hasNext()) {
    Object e =  itEmpr.next();
    String n = ((Empleado) e).dimeNombre();
    float s = ((Empleado) e).calculaSueldo();
    float c =  ((Empleado) e).dimeComision();
}
```


El Problema del Contenedor con asignación estática de tipos

```
ArrayList empresa = new ArrayList();
empresa.add(new EmpleadoAsalariado("Juan", 65426758, 1200));
empresa.add(new EmpleadoAComision("Ana", 54416351, 0.3));
Iterator itEmpr = empresa.Iterator();
while(itEmpr.hasNext()) {
    Object e =  itEmpr.next();
    String n = ((Empleado) e).dimeNombre();
    float s = ((Empleado) e).calculaSueldo();
    if (e instanceof EmpleadoAComision)
            float c =  ((EmpleadoAComision) e).dimeComision();
}
```

El Problema del Contenedor con asignación estática de tipos

```
ArrayList empresa = new ArrayList();
empresa.add(new EmpleadoAsalariado("Juan", 65426758, 1200));
empresa.add(new EmpleadoAComision("Ana", 54416351, 0.3));
Iterator itEmpr = empresa.Iterator();
while(itEmpr.hasNext()) {
    Empleado e =  (Empleado) itEmpr.next();
    String n = e.dimeNombre();
    float s = e.calculaSueldo();
    if (e instanceof EmpleadoAComision)
            float c =  ((EmpleadoAComision) e).dimeComision();
}
```


Solución alternativa: uso de genéricos

```
ArrayList<Empleado> empresa = new ArrayList<Empleado>();
empresa.add(new EmpleadoAsalariado("Juan", 65426758, 1200));
empresa.add(new EmpleadoAComision("Ana", 54416351, 0.3));
Iterator<Empleado> itEmpr = empresa.Iterator();
while(itEmpr.hasNext()) {
    Empleado e =  itEmpr.next();
    String n = e.dimeNombre();
    float s = e.calculaSueldo();
    ¿EmpleadoAComision?

}
```

El Problema del Contenedor con asignación estática de tipos

```
ArrayList<Empleado> empresa = new ArrayList<Empleado> ();
empresa.add(new EmpleadoAsalariado("Juan", 65426758, 1200));
empresa.add(new EmpleadoAComision("Ana", 54416351, 0.3));
Iterator<Empleado> itEmpr = empresa.Iterator();
while(itEmpr.hasNext()) {
    Empleado e =  itEmpr.next();
    String n = e.dimeNombre();
    float s = e.calculaSueldo();
    if (e instanceof EmpleadoAComision)
            float c =  ((EmpleadoAComision) e).dimeComision();
}
```


Eficiencia vs Flexibilidad en la asignación de tipos

Estática: ↑ Eficiencia, ↓ Flexibilidad

Dinámica: ↑ Flexibilidad, ↓ Eficiencia

Eficiencia vs Flexibilidad en la asignación de tipos

- Asignación estática de tipos
  - Conocer el objeto que sale del contenedor
  - Almacenamiento y generación de código
  - Detectar errores de tipos en compilación
- Asignación dinámica de tipos
  - Aumenta el costo de paso de mensajes (requiere ligadura dinámica)
  - Errores de tipos no aparecen hasta tiempo de ejecución
  - Simplifica la construcción de ED genéricas

2. Tiempo de enlace

- Temprano o estático
- Tardío o dinámico

Tiempo de Enlace

- Momento en el que se determina el significado exacto de una construcción
  - Tipo de una variable
  - Clase de un objeto
  - Método que responde un mensaje

Enlace Estático y Dinámico

- **Estático:** El enlace del método al mensaje se basa en el tipo estático de la variable. Ej. C++
- **Dinámico:** El enlace del método al mensaje se basa en el tipo dinámico de la variable. Ej. Java y C++ con métodos "virtual"
- ¿y SmallTalk? Tiene asignación dinámica de tipos y por tanto enlace dinámico

El Problema del Contenedor con asignación estática. Errores en tiempo de ejecución

```
ArrayList empresa = new ArrayList();
empresa.add(new Pera());
empresa.add(new EmpleadoAComision("Ana", 54416351, 0.3));
Iterator itEmpr = empresa.Iterator();
while(itEmpr.hasNext()) {
    Object e =  itEmpr.next();
    String n = ((Empleado) e).dimeNombre(); //Error en ejecución:
    float s = ((Empleado) e).calculaSueldo(); //casting incompatible
    if ( e instanceof EmpleadoAComision)
            float c =  ((EmpleadoAComision) e).dimeComision();
}
```

El Problema del Contenedor con asignación dinámica

```
empresa := OrderedCollection new.
empresa add: (Pera new).
empresa add: (EmpleadoAComision
                    newNombre:"Ana"
                    dni: 54416351
                    comision: 0.3).
empresa do:[:empleado |
    n := empleado dimeNombre; 'Error en ejecución: mensaje no válido'
    s := empleado calculaSueldo; 'Error en ejecución'
    (c instanceOf #EmpleadoAComision) ifTrue:[
        c := empleado dimeComision.] 'con tipos dinámicos no hay
                                      casting'
]
```

Eficiencia vs Flexibilidad en el tiempo de enlace

Estático: ↑ Eficiencia, ↓ Flexibilidad

Dinámico: ↑ Flexibilidad, ↓ Eficiencia

Eficiencia vs Flexibilidad en el tiempo de enlace

- Enlace estático de métodos
  - Paso de mensajes implantado como llamadas a procedimientos
- Enlace dinámico de métodos
  - Acoplar en tiempo de ejecución método y mensaje
  - Pueden ocurrir errores en tiempo de ejecución
  - Facilita el desarrollo de componentes sw reutilizables

Ligadura/enlace en los Lenguajes de Programación

- Lenguajes tradicionales
  - Ligadura estática: (compilación/encuadernación)
- Lenguajes orientados a objetos
  - Siempre dinámica (Smalltalk)
  - Lo elige el programador (C++)
  - Depende de si se trata de una clase o un tipo básico (Java)

Ligadura y asignación de tipos en los Lenguajes de Programación

	Asignación de Tipos	Ligadura de Métodos
Smalltalk	Dinámica	Dinámica
Java	Estática	Dinámica para objetos
C++	Estática	Estática por defecto

Asignación de tipos/tiempo de enlace

Tiempo de enlace	Asignación de tipos: Estática/ Estática fuerte	Asignación de tipos: Dinámica
Enlace Estático	Pascal, C++, Java (tipos básicos)	
Enlace dinámico	Java (clases), C++ (*virtual*)	**Small Talk**

Ejemplo Ligadura Estática

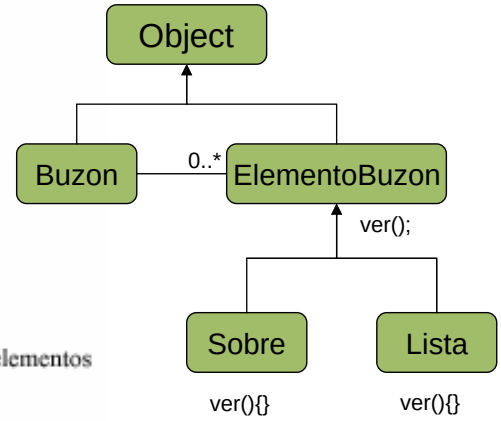
- Ejercicio:
  - Tenemos un buzón de correo electrónico donde se reciben varios tipos de elementos: carta, lista, informe
  - Se pretende visualizar cada uno de los elementos
  - Usar cualquier lenguaje tradicional para dar la solución

Ejemplo Ligadura Estática

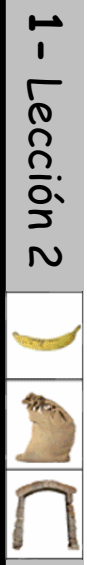
```
Elementos = (Sobre, Lista, Informe,...)

ElementoBuzon : record
                    Nombre:...;
                    Dirección:...;
                    CASE tipo_elemento: elementos
                        Sobre: ...;
                        Lista: ...;
                        Informe: ...;

Procedure ver_elemento (elemento: ElementoBuzon);
    ....
    CASE elemento.tipo_elemento
        Sobre: ver_sobre (elemento);
        Lista: ver_lista (elemento);
        Informe: ver_informe (elemento);
        ....
    END CASE
```

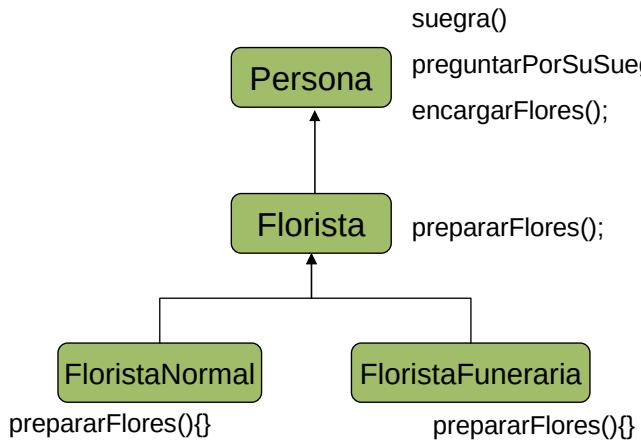

Ejemplo(I) Ligadura Dinámica

```
Clase ElementoBuzon: hereda de Object

    Public void ver ()

Clase Sobre: hereda de ElementoBuzon
    Public void ver()
        {muestra un sobre}

Clase Lista: hereda de ElementoBuzon
    Public void ver()
        {muestra una lista}


Clase Buzon: hereda de Object

    Protected ColeccionDeElementoBuzon elementos

    Public void ver()
    {....
        Para cada Item de elementos
            Item.ver()
        FinHacer
    }
```



Ejemplo(II) Ligadura Dinámica





Ejemplo(II) Ligadura Dinámica

Persona:

suegra "Devuelve el nombre de la suegra"

estadoSuegra "viva/muerta"

encargarFlores

```
self estadoSuegra='viva'
ifTrue: [florista:= FloristaNormal new]
ifFlase: [florista:=FloristaFuneraria new].
ramo:=florista prepararFlores.
florista entregar: ramo a: (self suegra)
```

Ejemplo(II) Ligadura Dinámica

Florista:

prepararFlores

entregar: unRamo a: unNombre

FloristaNormal (subclase florista)

prepararFlores: "prepara un ramo de rosas"

FloristaFuneraria (subclase florista)

prepararFlores: "prepara una corona de flores"

```
unaPersona:= Persona new.
unaPersona encargarFlores
```


Ejemplo(II) Ligadura Dinámica

Persona:

suegra "Devuelve el nombre de la suegra"

estadoSuegra "viva/muerta"

encargarFlores

```
self estadoSuegra='viva'
ifTrue: [florista:= FloristaNormal new]
ifFlase: [florista:=FloristaFuneraria new].
ramo:=florista prepararFlores.
florista entregar: ramo a: (self suegra)
```

Variable polimórfica

Ligadura dinámica

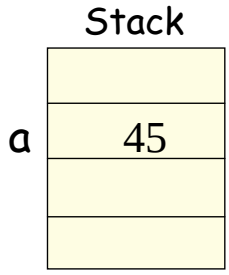
3. Asignación de memoria

- Estática: pila (stack)
- Dinámica: montón/montículo (heap)

Stack vs Heap

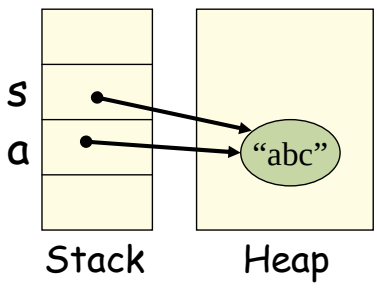
- Variable automática

```
{
int a = 45;
... a ...
}
```



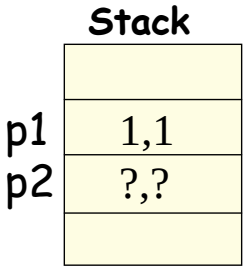
- Variable dinámica

```
String s = new String("abc");
{
 String a;
 a = s;
  ... a...
}
```



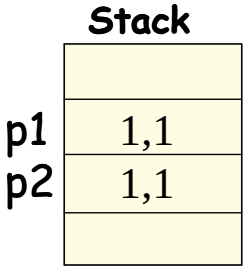
Stack vs Heap

```
{
Punto2D p1;
Punto2D p2;
p1.fijarX(1);
p1.fijarY(1);




}
```



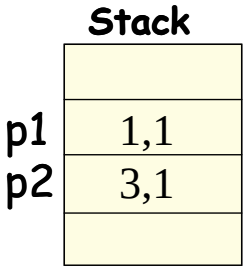
Stack vs Heap

```
{
Punto2D p1;
Punto2D p2;
p1.fijarX(1);
p1.fijarY(1);

p2 = p1;




}
```



Stack vs Heap

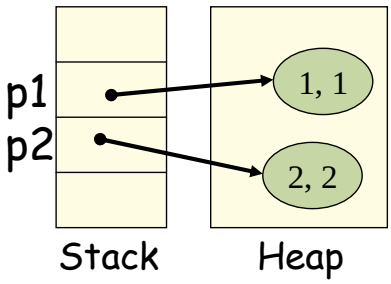
```
{
Punto2D p1;
Punto2D p2;
p1.fijarX(1);
p1.fijarY(1);

p2 = p1;

p2.fijarX(3);

¿p1==p2?

}
```




```
p2.fijarX(1);
p2.fijarX(1);

¿p1==p2?
```

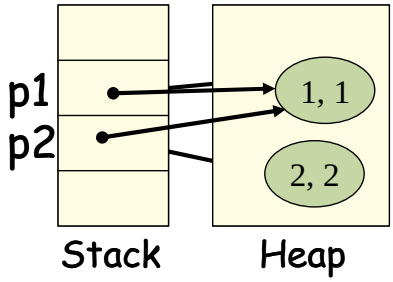

Stack vs Heap

```
JAVA
{
Punto2D p1 = new Punto2D(1,1);
Punto2D p2 = new Punto2D(2,2);




}
```



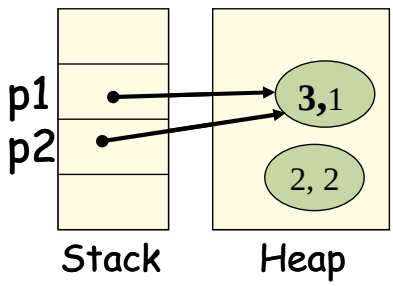
Stack vs Heap

```
JAVA
{
Punto2D p1 = new Punto2D(1,1);
Punto2D p2 = new Punto2D(2,2);

p2 = p1;




}
```



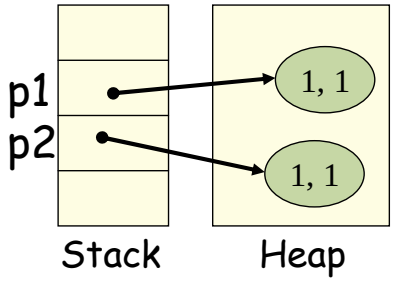
Stack vs Heap

```
JAVA
{
Punto2D p1 = new Punto2D(1,1);
Punto2D p2 = new Punto2D(2,2);

p2 = p1;

p2.fijarX(3);

¿p1==p2?


}
```



Stack vs Heap

```
JAVA
{
Punto2D p1 = new Punto2D(1,1);
Punto2D p2 = new Punto2D(1,1);

¿p1==p2?
}
```



Stack vs Heap

```
Estilo C++
{
Punto2D* p1 = new
    Punto2D(1,1);
Punto2D* p2 =new
    Punto2D(2,2);

p2 = p1;

*p2.fijarX(3);

¿p1==p2?;

delete p1;
delete p2; // error de
    ejecución, ¿todo borrado?
}
```

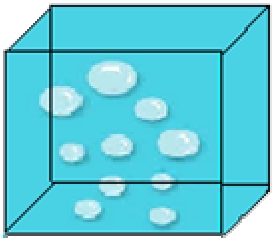
Stack vs Heap

- Recuperación de memoria
  - Liberación explícita
  - Recolector de basura
- Vida de los valores
- Apuntadores
  - Explícito
  - Todo son punteros
- Creación inmutable

Stack vs Heap

- Creación/Destrucción
- Asignación
- Comparación
  - Stack
  - Heap. ¿Comparar valor?
- Copia
  - Stack
  - Heap. ¿Copiar valor?

Tema 2

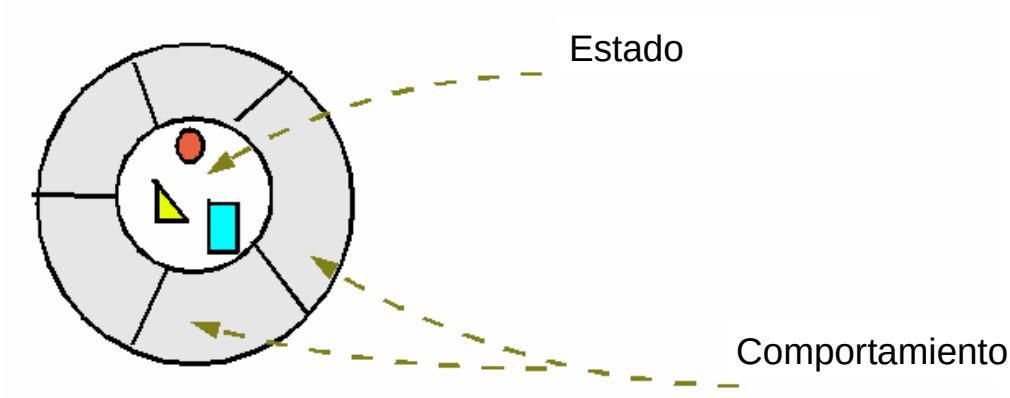


Lección 1

Objetos y Mensajes

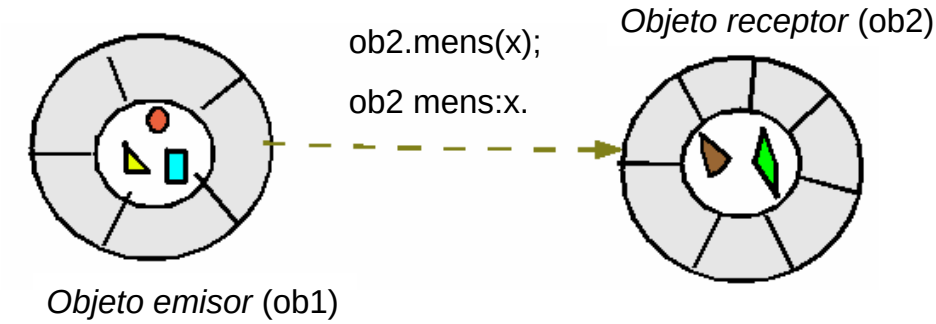
Identidad, Comportamiento y Estado

- Un objeto es una **entidad software** que posee **datos** y tiene capacidad de **procesamiento**



Identidad, Comportamiento y Estado

- El comportamiento viene determinado por los mensajes que acepta el objeto



Identidad, Comportamiento y Estado

- Ejemplo en Java:

```
Rectangulo a=new Rectangulo();
a.rotar(5);
```

- Ejemplo en C++:

```
Rectangulo a;
a.rotar(5);
```

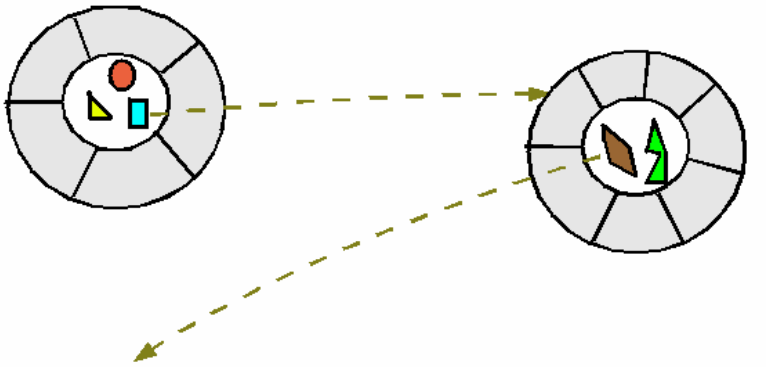
```
Rectangulo *a=new Rectangulo();
*a.rotar(5); //a->rotar(5)
```

- Ejemplo en SmallTalk:

```
a:=Rectangulo new.
a rotar: 5
```

Identidad, Comportamiento y Estado

- El **estado** es el conjunto de datos del objeto
  - El estado de un objeto puede estar definido por el estado de otros objetos que lo componen



Identidad, Comportamiento y Estado

- La **identidad** de un objeto permite diferenciarlo del resto
  - Depende de su propia existencia
  - Es inherente al modelo. ¿Y en el modelo relacional?

Estado ≠ Identidad

¿Iguales o el mismo?

Identidad, Comportamiento y Estado

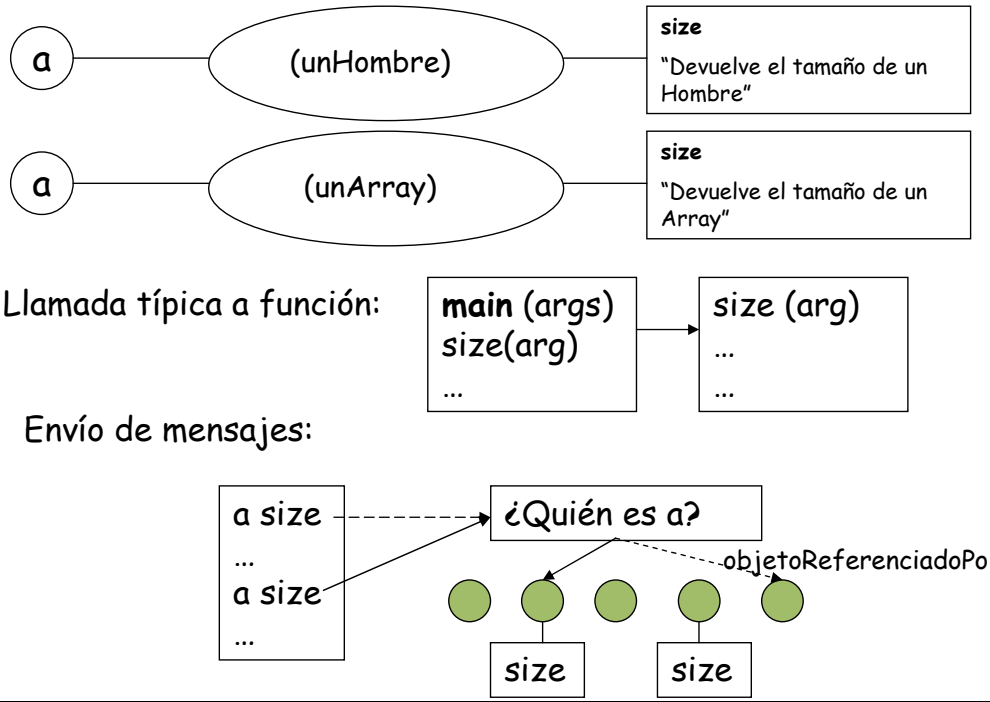
Estado ≠ Identidad

	Java	ST
	Rectangulo a=new Rectangulo(1,1,30,20); Rectangulo b=new Rectangulo(1,1,30,20);	a:=Rectangulo newx1:1 y1:1 x2:30 y2:20 b:=Rectangulo newx1:1 y1:1 x2:30 y2:20
Identidad	a==b a!=b	a==b a~~b
Estado	a.equalsTo(b)	a=b a~=b

Creación e Inicialización de Objetos

- Creación del objeto
  - Asignar memoria
  - Ligar el espacio a un nombre
- Inicialización del objeto
  - Condiciones iniciales para su manipulación
    - Inicialización del área de datos
  - Encapsulación: ocultar inicialización

Mecanismo de envío de mensajes



Paso de Mensajes – Object Pascal

- Notación convencional de puntos

  objeto-receptor.selector-de-mensaje

- Seudovariable self
- Ejemplos:

  x.moverA(45,46);

  x.volver;

Paso de Mensajes – C++

- Exige paréntesis en la invocación de una función miembro

```
objeto-receptor.funcion-miembro(...)
```

- Seudovariable this
  - Apuntador al objeto
    - this → palo();
  - A ausencia de receptor se sobreentiende el actual
    - palo();

Paso de Mensajes – Smalltalk

- El punto se sustituye con un espacio y para los argumentos se utilizan palabras clave

Unario: objeto-rec. selector-mensaje **#(4 5 'a') size**
Binario: objeto-rec.r selector1o2CaracteresNoAlfanumericos arg1 **4+3**
Palabra clave: objeto-rec. palabra-clave1: arg1 palabra-clave2: arg2 ... **3 compara: 7**

- Prioridad de ordenamiento:
  - Unarios
  - Binarios
  - Palabras clave

3 compara:4 + #(4 5 'a') size

3

7

false

- Seudovariable self

Paso de Mensajes – Objective C

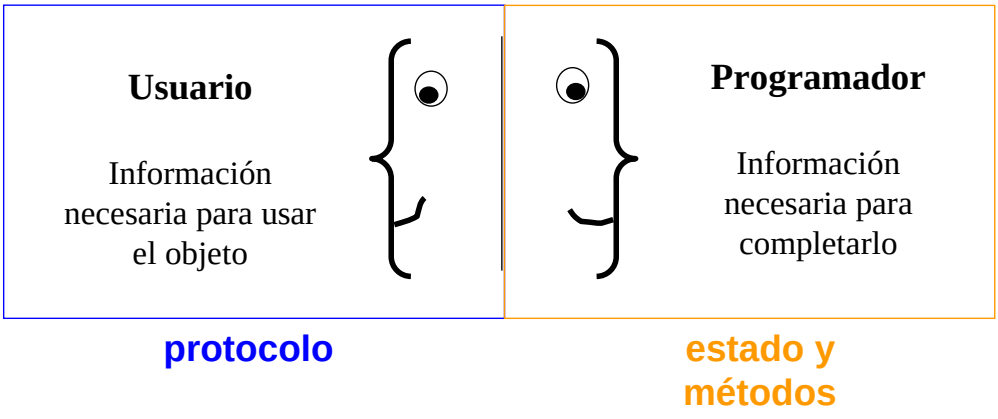
- Expresiones de mensaje

  [objeto-receptor selector-mensaje]

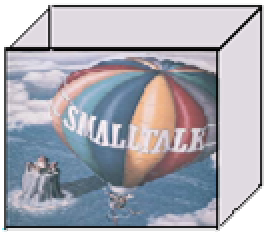
  - Ejemplo: x = [y copy];
- Los mensajes binarios no pueden redefinirse
- Variable self
  - Puede ser manipulada por el usuario

Encapsulación

- Podemos ver los objetos como ejemplos de TDA:



Tema 2

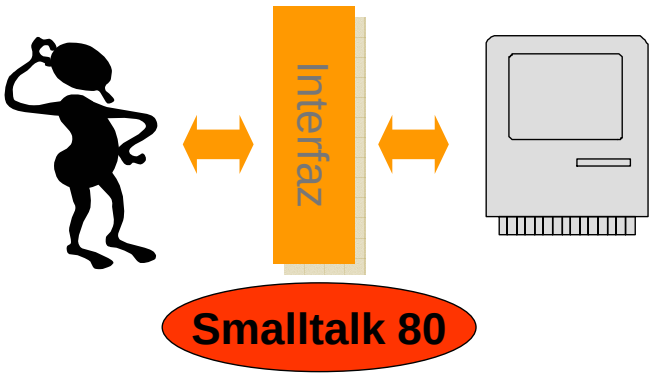


Lección 2

Objetos y Mensajes en Smalltalk

Un Poco de Historia

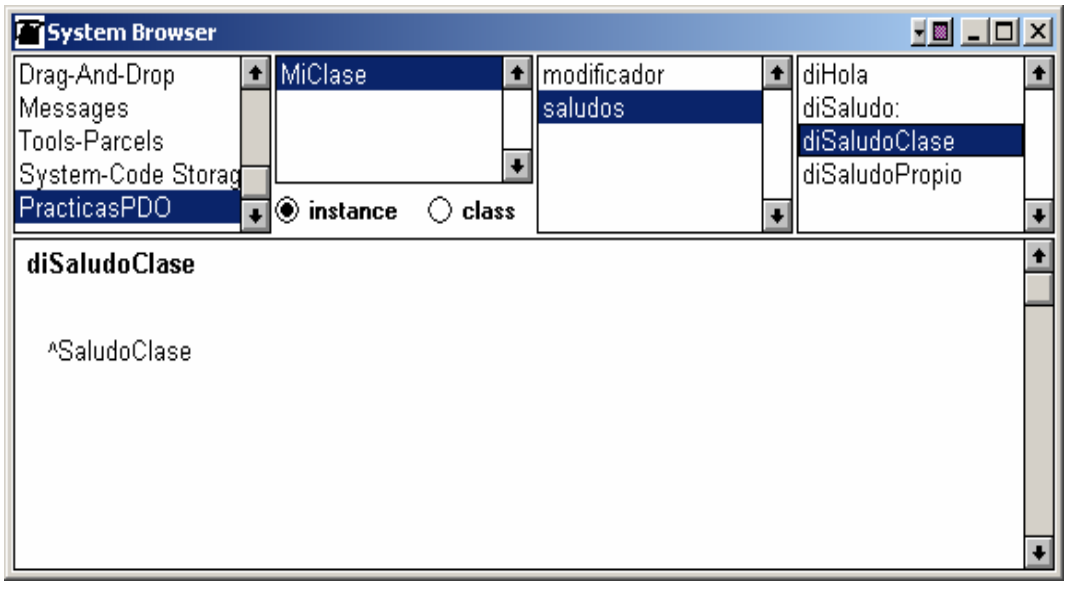
- Principios de los setenta
- Centro de Investigación XEROX (EE.UU.)
  - Investigador Alan Kay



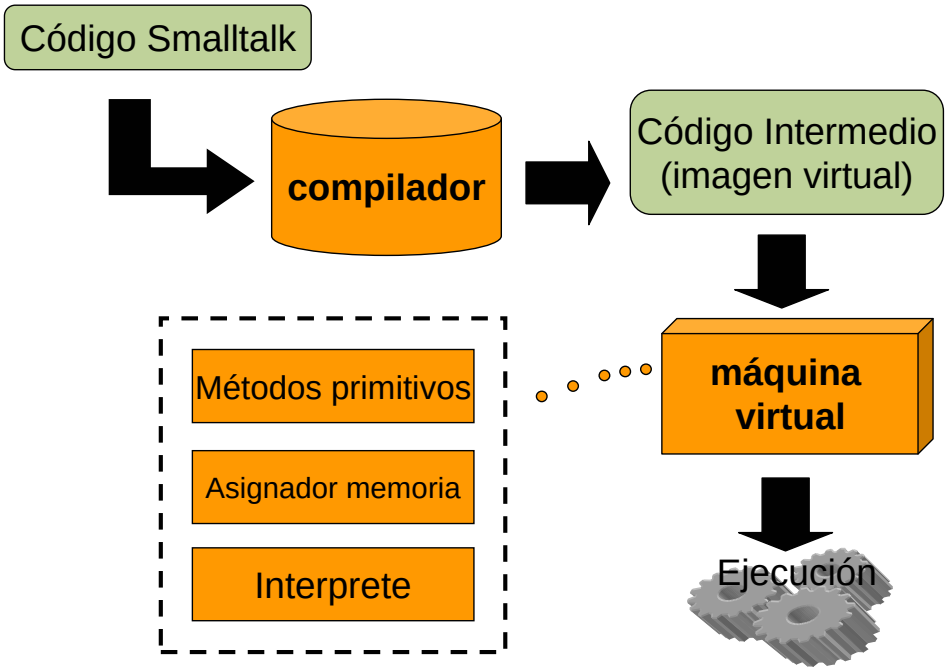
El Sistema de Programación

- Smalltalk:
  - Lenguaje de programación O.O.
  - Entorno de programación gráfico
  - Elementos preprogramados
- TODO es un OBJETO
- La programación añade funcionalidad al sistema

El Sistema de Programación



Entorno de Programación



Ficheros Smalltalk

- Máquina virtual
- Objetos predefinidos en fuente
- Objetos compilados
- Fichero de Cambios

- visual.im
- vw.exe
- visual.cha
- visual.sou

Expresiones

- Secuencia de caracteres que describe un objeto llamado valor de la expresión
- Cuatro tipos
  - Literales
  - Nombres de variables
  - Expresiones de mensaje
  - Expresiones de bloque

Literales

- Número

5	7.9	7.9e-5	2r101

- Carácter

$a	$$	$1

- Cadena

'abc'	''	'a'

- Símbolo (nombre del sistema)

#Array	#VarGlobal

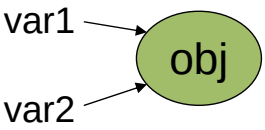
- Array de literales

#(9 'nueve' $9 #nueve #(0 'cero'))

Nombres de Variables

- Todas las variables en Smalltalk son referencias
- Una variable puede apuntar a cualquier objeto
- En la asignación hay compartición de objetos

```
var1 := expresión.
var2 := var1
```



Tipos de Variables

- Variables
- Seudovariables: no se les puede asignar valor

Tipos de Variables

- Variables
  - Privadas
  - Compartidas
- Seudovariables
  - Constantes
  - Variables

Tipos de Variables

- Variables
  - Privadas
    - Variables de instancia
    - Variables temporales
  - Compartidas
    - Variables de clase
    - Variables semiglobales
    - Variables globales
- Seudovariables
  - Constantes
  - Variables

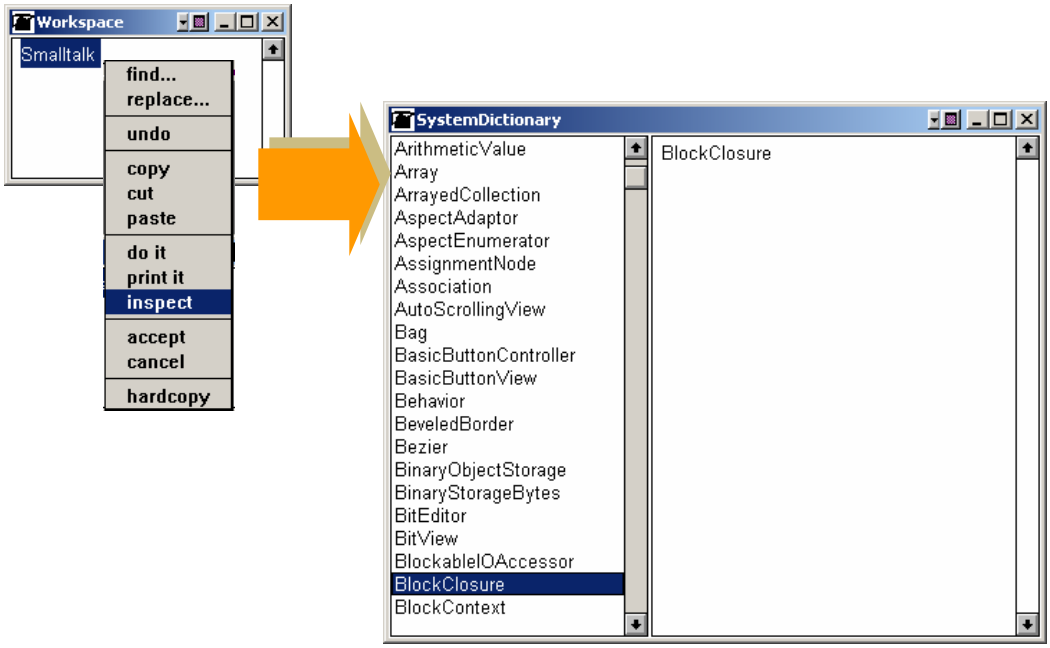
Variables Privadas

```
Empleado subclass: #EmpleadoAComision
instanceVariableNames: 'comision ventas'
classVariableNames:''
poolDictionaries:''

!EmpleadoAComision methodsFor:'modificación'
dimeSueldo
   |sueldo|
   sueldo := comision * ventas.
   ^sueldo
```

Variables Compartidas

```
Object subclass: #Empleado
instanceVariableNames: 'dni nombre'
classVariableNames:'PorcentajeRetencion'
poolDictionaries:'MiPoolDiccionario'

Smalltalk at:#MiPoolDiccionario put:Dictionary new.
MiPoolDiccionario at:#VarSemiGlobal put:valor

Smalltalk at:#VarGlobal put:valor
```

Variables Compartidas

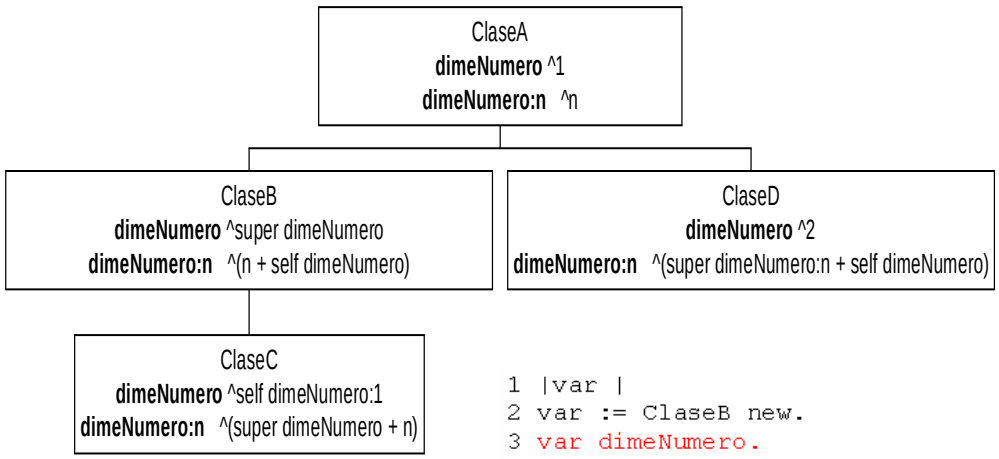


Seudovariables

- Constantes
  - nil
  - true, false
- Variables
  - self
  - super
  - Los argumentos en el patrón de un mensaje. Ej: **index** y **character** en

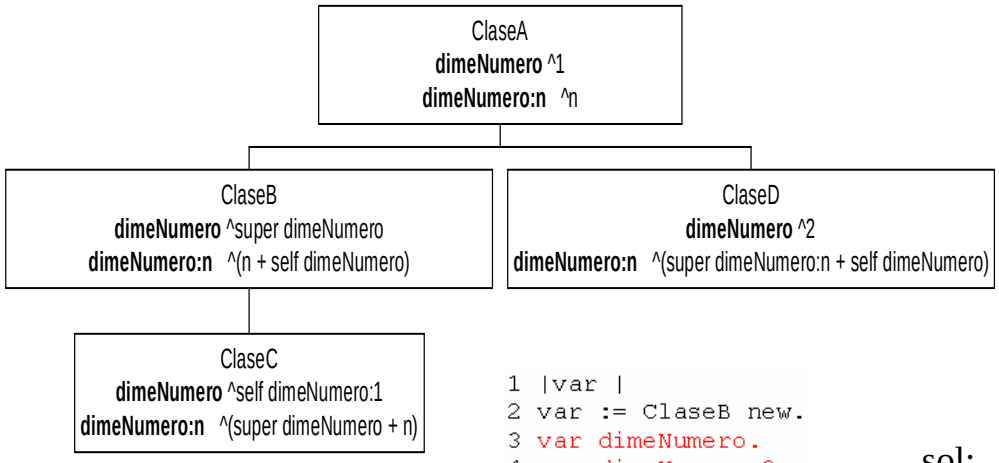
```
!Text methodsFor: 'accessing'!

at: index put: character
    "Store the argument aCharacter in the field of the receiver indicated
    by the index.  Answer aCharacter.  Fail if the index is not an
    Integer or is out of bounds, or if the argument is not a Character. "

    ^super at: index put: character! !
```

Seudovariables




```
1 |var |
2 var := ClaseB new.
3 var dimeNumero.
4 var dimeNumero:3.
5 var := ClaseC new.
6 var dimeNumero.
7 var dimeNumero:3.
8 var := ClaseD new.
9 var dimeNumero.
10 var dimeNumero:3.
```


Seudovariables




```
1 |var |
2 var := ClaseB new.
3 var dimeNumero.
4 var dimeNumero:3.
5 var := ClaseC new.
6 var dimeNumero.
7 var dimeNumero:3.
8 var := ClaseD new.
9 var dimeNumero.
10 var dimeNumero:3.
```

sol:

5

Expresiones de Mensaje

- Efecto y Resultado
  - Lo que ocurre (estado del objeto)
  - El valor devuelto
    - Por defecto el objeto receptor

	Efecto	Modifica	Resultado
cuentaCliente saldo	Dice el saldo de la cuenta	Nada	El saldo
cuentaCliente deposito: 500	Incrementar saldo de la cuenta	La cuenta	La cuenta

Expresiones de Mensaje

- **Unarias** *receptor selector*
  - 3.1419 sin
  - cantidad sqrt
  - #(1 2 3) size
- **Con palabras clave**

receptor palabraClave1: arg1 palabraClave2: arg2 ...

  - 28 gcd: 12
- **Binarias**

receptor símboloEspecial argumento

  - 3 + 7
  - 'hola','mundo'

Expresiones de Mensaje

- Orden de Evaluación
  - de izquierda a derecha
  - 1º) Paréntesis
  - 2º) Unarias
  - 3º) Binarias
  - 4º) Palabras clave
- Las expresiones de mensaje con palabras clave que son argumentos de otra expresión con palabras clave, deben estar entre paréntesis

	Resultados:
4 + 7 * 5 + 2	57
10 raisedTo: 2 + 4 sqrt	10000
edades at: 'Juan' put: (edades at:'Pedro')	
edades at:'Juan' put: edades at:'Pedro'	at:put:at: mensaje no entendido

Secuencias de Expresiones de Mensaje

- Separador de expresiones de mensaje: .
  - Para un mismo objeto: ;

```
vector at: 1 put: 7.
vector at: 2 put: 45 .
vector at: 3 put: 23.
```

```
vector at: 1 put: 7 ;  at: 2 put: 45 ; at: 3 put: 23
```

Expresiones de Bloque

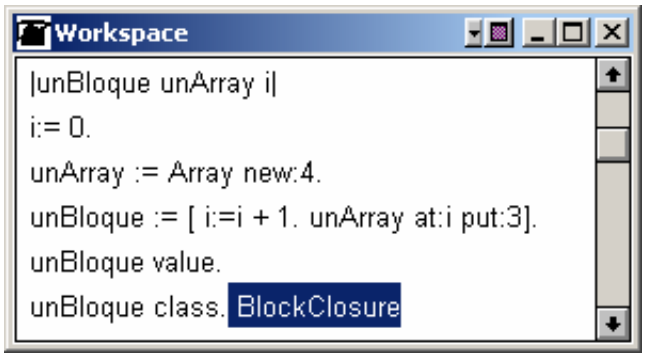
- Concepto de bloque
  - **Secuencia aplazada de acciones**
  - Objeto de la clase *BlockClosure*
- Sintaxis del bloque
  - Secuencia de expresiones encerradas entre corchetes

  [expr1. expr2. ... exprN]
- Evaluación del bloque
  - Mensaje value
  - Resultado: Valor de la última expresión

Expresiones de Bloque

- Ejemplo:

```
|unBloque unArray i|
i:= 0.
unArray := Array new:4.
unBloque := [ i:=i + 1. unArray at:i put:3].
unBloque value.
```



Expresiones de Bloque con Argumentos

- Sintaxis del bloque con argumentos

[:arg1 :arg2 ...| expresiones que pueden usar arg1 arg2...]

- Evaluación del bloque con argumentos

value:argActual1 value:argActual2 ...

- Ejemplo:

```
|blq|
blq := [:a :b | (a,b) size].
blq value:'hola' value:'amigo'
```

Expresiones de Bloque: Estructuras de Control

- Estructuras de Decisión Condicional: se implementan mediante los siguientes métodos:
  - ifTrue:
  - ifFalse:
  - ifTrue: ifFalse:
  - ifFalse: ifTrue:

  Receptor: objeto de las clases True o False
  Argumento-s: objeto de la clase BlockClosure

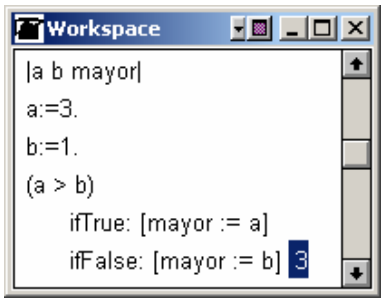
- Estructuras de Iteración Condicional: se implementan mediante los siguientes métodos:
  - whileTrue
  - whileFalse
  - whileTrue:
  - whileFalse:

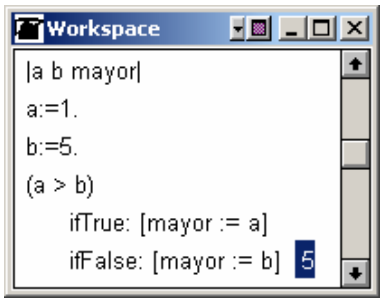
  Argumento: objeto de la clase BlockClosure

  Receptor: objeto de la clase BlockClosure

Expresiones de Bloque: Estructuras de Control

- Ejemplo de estructuras de decisión:





Expresiones de Bloque: Estructuras de decisión condicional

Objeto receptor	Mensaje	Argumentos	Efecto	Valor devuelto
false	ifTrue:	[bloque]		
false	ifFalse:	[bloque]		
false	ifTrue:ifFalse:	[bloqueTrue] [bloqueFalse]		
false	ifFalse:ifTrue:	[bloqueFalse] [bloqueTrue]		
true	ifTrue:	[bloque]		
true	ifFalse:	[bloque]		
true	ifTrue:ifFalse:	[bloqueTrue] [bloqueFalse]		
true	ifFalse:ifTrue:	[bloqueFalse] [bloqueTrue]		

Expresiones de Bloque: Estructuras de Control

- Ejemplo de estructura de iteración condicional:

```
| vector buscado encontrado i |
vector := #(1 2 3 4 5).
buscado := 3.
i := 1.
encontrado := false.

[encontrado not] whileTrue:[
     ((vector at:i) = buscado)
          ifTrue:[encontrado := true ].
     i := i+1.
].
```


Expresiones de Bloque: Estructuras de Control

[bloqueReceptor] *whileTrue*: [bloqueArgumento]
[bloqueReceptor] *whileFalse*: [bloqueArgumento]
[bloqueReceptor] *whileTrue*
[bloqueReceptor] *whileFalse*

whileTrue: bloqueArgumento

whileTrue

whileFalse

whileFalse: bloqueArgumento

Expresiones de Bloque: Iteradores

- Iteradores
  - timesRepeat:

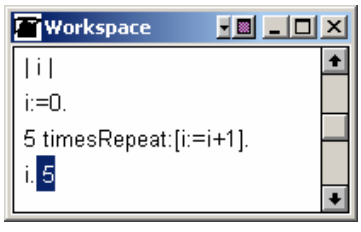
Receptor: objeto de la clase Integer
Argumento: objeto de la clase BlockClosure

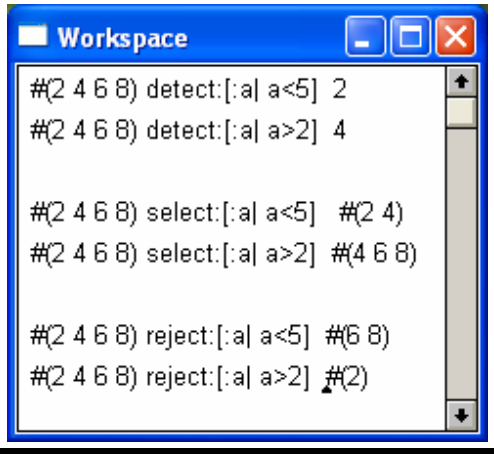
- Iteradores de colección
  - do:
  - collect:
  - detect:
  - select:
  - reject:

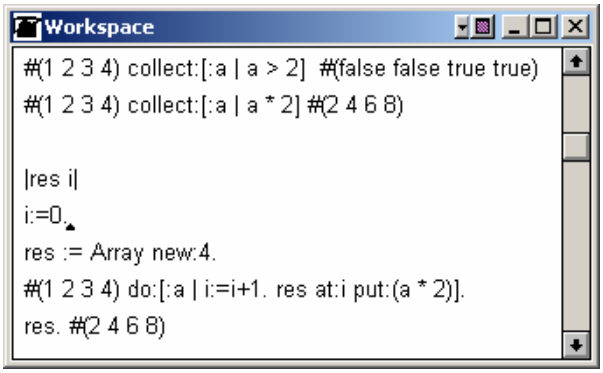
Argumento: El resultado de su evaluación debe ser un objeto de la clase True o False

Receptor: objeto de la clase Collection o subclases
Argumento: objeto de la clase BlockClosure con argumento

Expresiones de Bloque: Iteradores

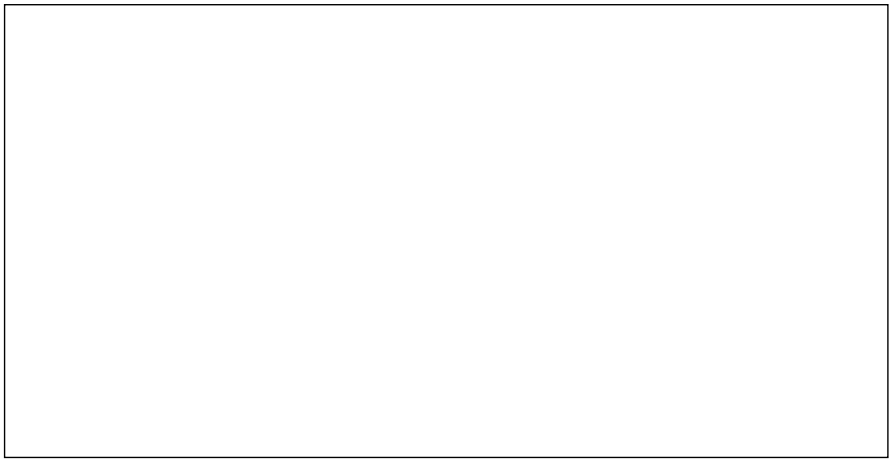






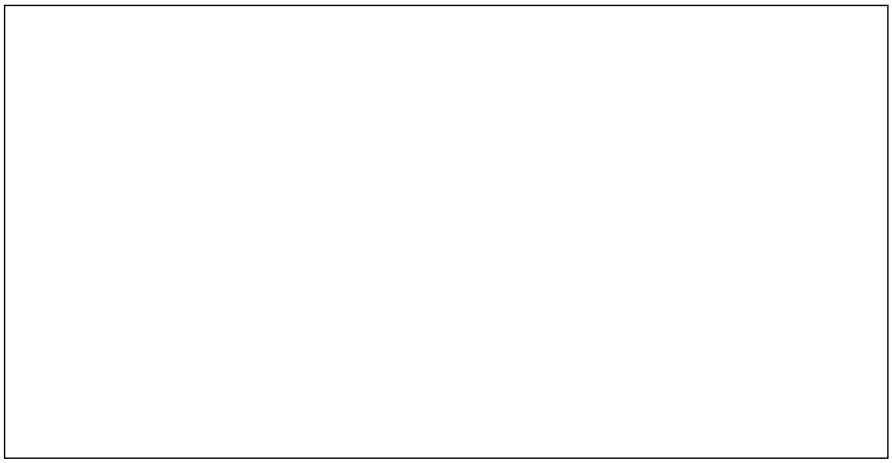
Ejercicios (I)

- Implementar el método do:



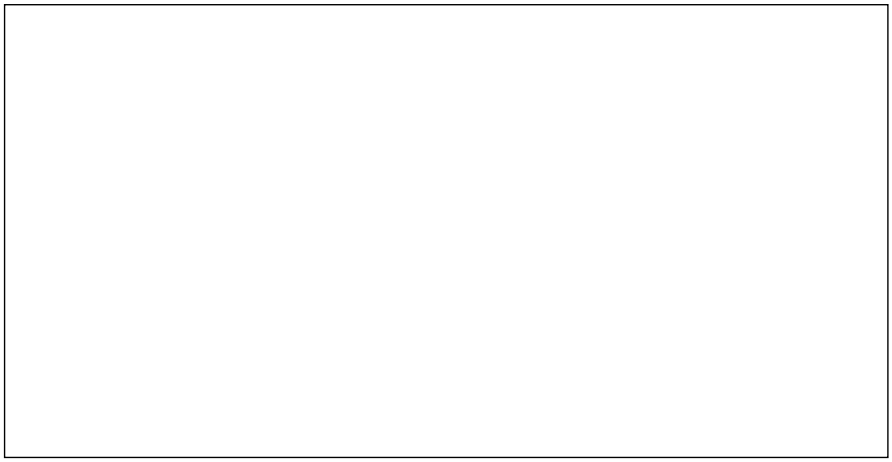
Ejercicios (II)

- Implementar el método detect:



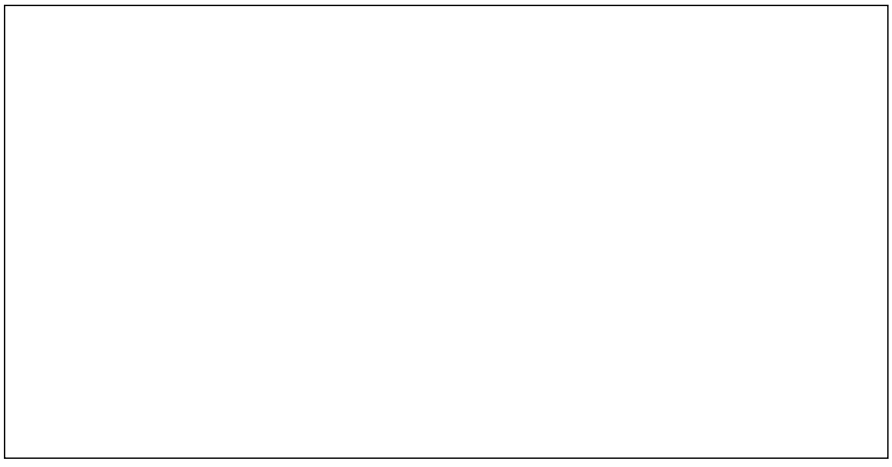
Ejercicios (III)

- Implementar el método collect:



Ejercicios (IV)

- Implementar el método select:



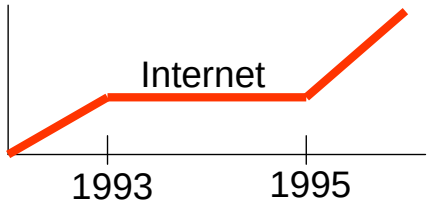
Tema 2

Lección 3

Objetos y Mensajes en Java

Un Poco de Historia

- Principios de los noventa
  - Inicialmente denominado OAK
- Proyecto Green de Sun Microsystems
  - Investigador James Gosling
- Motivaciones iniciales
  - Interfaces cómodas e intuitivas
  - Controladores electrónicos

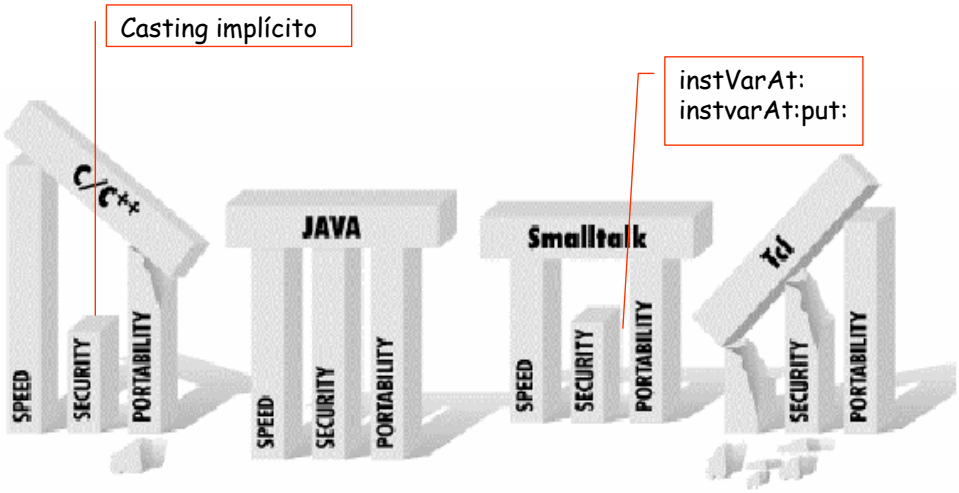


SUN vs. MICROSOFT

Características

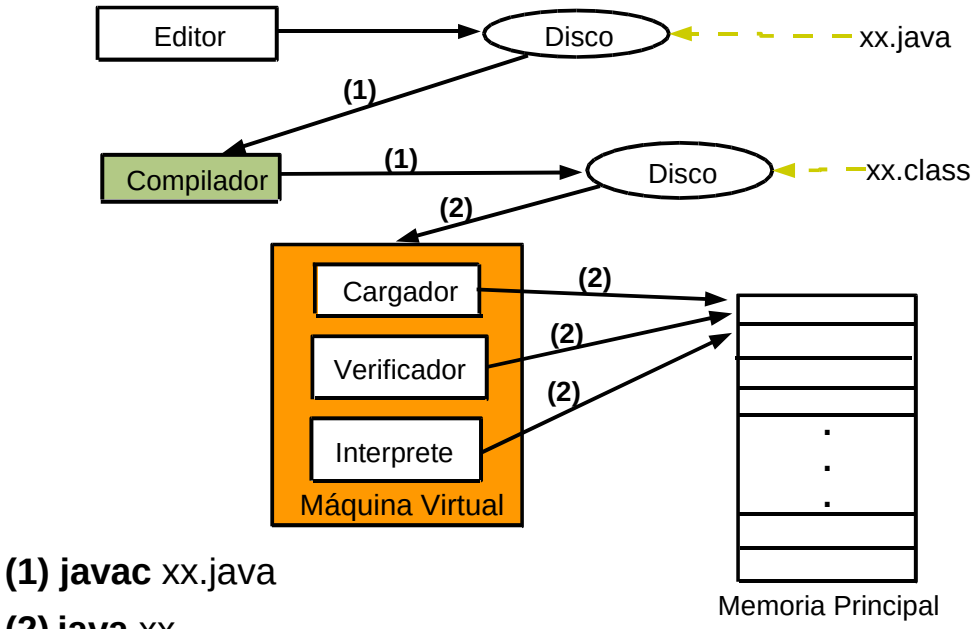
- Orientado a objetos
- Simple
- Robusto
- Seguro
- Arquitectura neutra
  - Interpretado
  - Portable
- Distribuido
- Multitarea
- Dinámico

Comparativa



Extraído: “Learning Java” Pat Niemeyer et al. O'Reilly, 2000

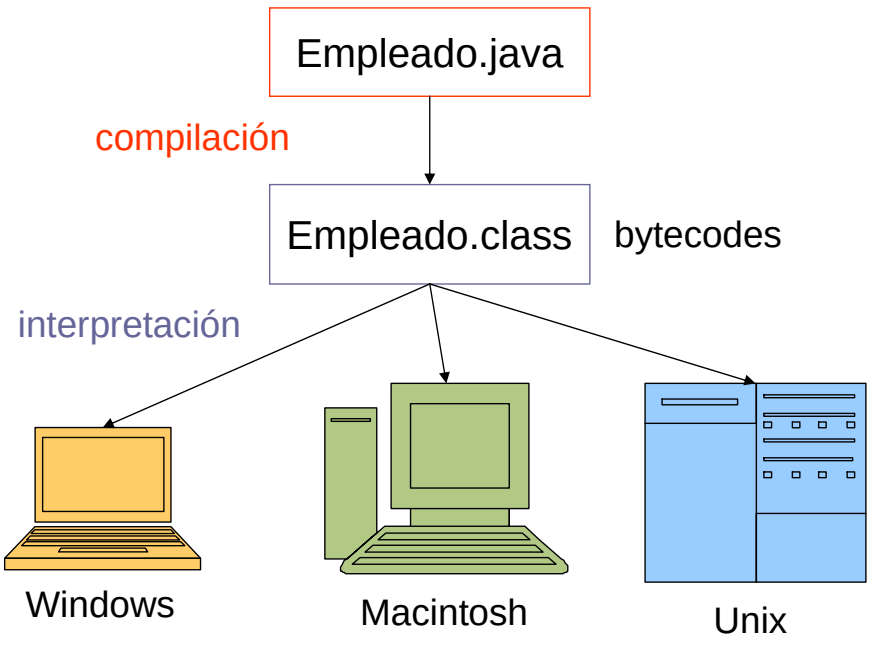
El Entorno



(1) javac xx.java

(2) java xx

La Máquina Virtual (JVM)



La Máquina Virtual (JVM)

- Principales funciones
  - Reservar espacio en memoria
  - Liberar la memoria no usada
  - Asignar variables a registros y pilas
  - Vigilar el cumplimiento de las normas de seguridad
- La máquina virtual de Java puede estar implementada en software o en hardware

Variables y Valores

- Declaración de la variable

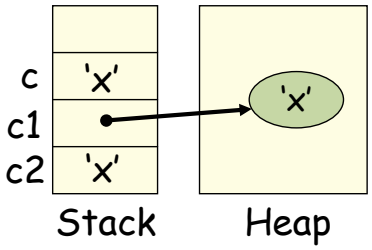
  tipo nombre_var [inicialización]
- Tipos de datos
  - Tipo primitivo → Valor puro
  - Clase → Objeto
- La variable referencia a un objeto, NO es un objeto

```
Date b = new Date();
b.toString();      //Sí funciona

Date b;
b.toString();      //No funciona
```

Tipos Primitivos

Tipo primitivo	Tamaño/formato	Descripción
boolean	---	true o false
char	16 bit (Unicode)	un carácter simple
byte	8 bit c.2	entero
short	16 bit c.2	entero
int	32 bit c.2	entero
long	64 bit c.2	entero
float	32 bit SP	punto flotante
double	64 bit DP	punto flotante

```
char c = 'x';
Character c1 = new Character(c);
char c2 = c1.charValue();
```

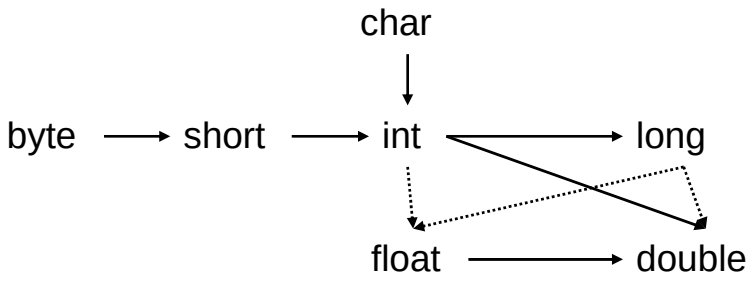


Operaciones con Tipos Primitivos

- de Incremento y Decremento
  - Prefija ++n;
  - Posfija n++;
- Relacionales y Booleanos
  - Igualdad 3==7, Desigualdad 3!=7
  - Comparación 6>5, 6>=7, ...
  - Lógicos a && b, a || b
  - Ternario (a>b) ? a : b
- De Bits
  - & (and), | (or), ^ (xor), ~(not)
  - >> (desplaza a la derecha), << (desplaza a la izquierda)

Conversiones con Tipos Primitivos

- Sin pérdida de información



- Con pérdida de información
  - Moldeados

```
double x = 9.987;
int nx = (int) x;
```


Creación e Inicialización de Objetos

- Constructores
  - Igual nombre que la clase
  - Operador new
    - Date d = new Date();
    - String s = new Date().toString();
  - Sobrecarga
    - Firma de un método
      - indexOf(int)
      - indexOf(int, int)
      - indexOf(String)
      - indexOf(String, int)
    - Resolución

Creación e Inicialización de Objetos

- Nombre de los parámetros
  - public Empleado(String **n**) { nombre = n }
  - public Empleado(String **unNombre**) { nombre = unNombre }
  - public Empleado(String **nombre**) { this.nombre = nombre }
- Constructor predeterminado
  - Inicialización por defecto
- Llamar a otro constructor
  - this (…)
  - super (…)

Creación e Inicialización de Objetos

- Inicializaciones de campo

```
class Empleado {
   private String nombre = “Juan”
}
```

- Bloques de inicialización

```
class Empleado{
   ...
   private String nombre;
   private String dni;
   { dni=“64543211”; nombre=”Juan”;}
   …
}
```

Creación e Inicialización de Objetos

- Orden en la creación de un objeto
  - Reserva de espacio en la memoria dinámica para almacenar el objeto
  - Inicialización por defecto de las variables de instancia
  - Incialización de campo y bloques de inicialización
  - Ejecución del constructor

Clases y Objetos

```
class Empleado {
private String dni;
private double sueldo;
private static double PorcentajeRetenciones = 0.5;
public Empleado (String d, double s){
   dni = d;
   sueldo = s;
}
public static void modificaPorcentajeRetenciones(double pR){
   PorcentajeRetenciones = pR;
}
…
}
```

Clases y Objetos

- La primera vez que se referencia una clase, el interprete de Java la busca y carga en memoria
- Se ejecutan una única vez todos sus inicializadores de variables de clase
- Se sigue el orden en la construcción de objetos

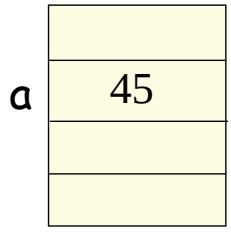
```
Empleado e = new Empleado ("Juan", "76873640");
Empleado.modificaPorcentajeRetenciones(0,8);
Empleado e = new Empleado("Ana", "49532349");
```

Se inicializa sólo una vez la variable PorcentajeRetenciones

Paso por Valor

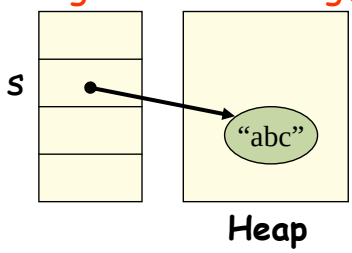
- Java realiza paso por valor para los argumentos y valores devueltos en los métodos

int a = 45;



- Paso por valor: 45
- Paso por referencia: La dirección de memoria donde se almacena la variable a

String s = new String("abc")



- Paso por valor: La dirección de memoria almacenada en la variable s (dirección del objeto)
- Paso por referencia: La dirección de memoria donde se almacena s

Clases y Objetos

```
class Empleado {
private String dni;
private double sueldo;
private static double PorcentajeRetenciones = 0.5;
public Empleado (String d, double s){
   dni = new String(d);
   sueldo = s;
}
public void dni (String d) { dni = new String(d);}
public String dni() { return new String(dni); }

public void sueldo (double s) { sueldoi = s;}
public double sueldo() {    return sueldo; }
}
```


Clases y Objetos

```
class Empleado {
private String dni;
private double sueldo;
private static double PorcentajeRetenciones = 0.5;
public Empleado (String d, double s){
   dni = d;
   sueldo = s;
}
public void dni (String d) { dni = d;}
public String dni() {    return dni; }

public void sueldo (double s) { sueldo = s;}
public double sueldo() {    return sueldo; }
}
```

Clases y Objetos

```
public void sueldoYRetenciones(double s, double pR){
  sueldo = s;
  PorcentajeRetenciones = pR;
}
public static void Retenciones (double pR){
  PorcentajeRetenciones = pR;
}
public Object clone(){
  return new Empleado(new String(dni), sueldo);
}
public boolean equals(Object o){
    Empleado e = (empleado) o;
    return (  ( (e.dni()). equals(dni) ) && (e.sueldo()==sueldo) );
}
}
```


Clases y Objetos

java.lang
Class Object

java.lang.Object

clone

```
protected Object clone()
                  throws CloneNotSupportedException
```

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object.

```
x.clone() != x
```

will be true,

```
x.clone().getClass() == x.getClass()
```

will be true,

```
x.clone().equals(x)
```

Returns:

a clone of this instance.

Throws:

`CloneNotSupportedException` - if the object's class does not support the `Cloneable` interface.

Clases y Objetos

```
Class Empleado implements Cloneable {
….
public Object clone(){
  Object e;
  try{
    e = (Empleado) super.clone();
  } catch(CloneNotSupportedException exc) {
  System.out.println("no se puede duplicar");
  }
  (String) e.dni= this.dni.clone();
  return e;
}
….
}
```


Clases y Objetos

```
class Prueba{
public static void main(String args[]){
    Empleado e = new Empleado("1", 100);
    e.sueldoYRetenciones(1000, 0.4);
    Empleado.Retenciones(0.5);
    e.Retenciones(0.4);
    Empleado e2=(Empleado) e.clone();
    boolean b1 = ( e == e2);
    boolean b2 = (e.equals(e2));
    boolean b3 = ( (e.getClass()) == (e2.getClass()) );
    modificarArg(e);
}
public static void modificarArg(Empleado e3){
    e3.sueldo(2000);
    e3.dni("2");
}
}
```

Mensajes a Objetos

- A un objeto
  - Empleado e = new Empleado();
  - double s = e.sueldo();
  - Class c = e.getClass();
    - Class sc = c.getSuperclass();
    - boolean b = c.isInstance(e);
    - Method[] m = c.getMethods();
  - Empleado e2 = (Empleado) e.clone();
- A una clase
  - String s = String.valueOf(35);
  - String s2 = Integer.toHexString(35);
  - boolean b = Character.isDigit('4');
  - boolean b2 = Character.isUpperCase('d');

Arrays

- Clase Array
  - Contenedora de objetos del mismo tipo
  - Accesibles por un índice
  - Tamaño fijo
- Construcción de objetos
  - Integer[ ] numeros = new Integer[2]
  - Con inicialización explícita
    - Integer[] numeros = {new Integer(1), new Integer(2)}

Arrays

- Array bidimensional

```
int[ ][ ] numeros = new int[2][ ];
numeros[0] = new int[4];
numeros[1] = new int[2]
```

numeros[0][0]	numeros[0][1]	numeros[0][2]	numeros[0][3]
numeros[1][0]	numeros[2][0]		

Arrays

- Array con tipo y tamaño conocido en tiempo de ejecución

```
String nombreClase;
int tam;
Class tipo = Class.forName(nombreClase);
Array.newInstance(tipo, tam);
```

Reflexión

- Funcionalidad
  - Acceso
    - numeros[3] = numeros [1]
  - Longitud
    - numeros.length

Instrucciones de Control de Flujo

Condicionales

- ***if*** *(numero > 30)*
  obj.suma(numero);
  else *{resto = obj.resta(numero);*
  return resto;};

- ***switch*** *(int ch = ……)*
  case *1 :case '1': return 'si';*
  case 2: return 'no';
  default: return 'casi';

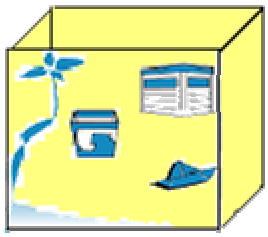
Iterativas

- *int[] valor;*
  int suma;
  for *(int i = 0; i < valor.lenght ; i++)*
  suma += valor[i];
  return suma

- ***while*** *(suma < 30) {*
  if (valor.lengh > i)
  break;
  else
  i++;
  suma += valor[i]; }

- ***do .... while(...)***

Tema 3



Lección 1

Clasificación de los Objetos

Puntos a tratar

- Métodos de Clasificación de los objetos
  - Clases y prototipos
  - Herencia versus delegación
- Concepto de clase
  - Clases-objeto
  - Clases-patrón
- Creación e inicialización de objetos: metaclases y patrones
  - Creación con clases-objeto: metaclases en Smalltalk
  - Creación con clases patrón: C++. Java
  - Una "metaclase" rara: la clase CLASS de Java

Métodos de Clasificación de Objetos

- Métodos de Clasificación de Objetos
  - Clases
  - Prototipos

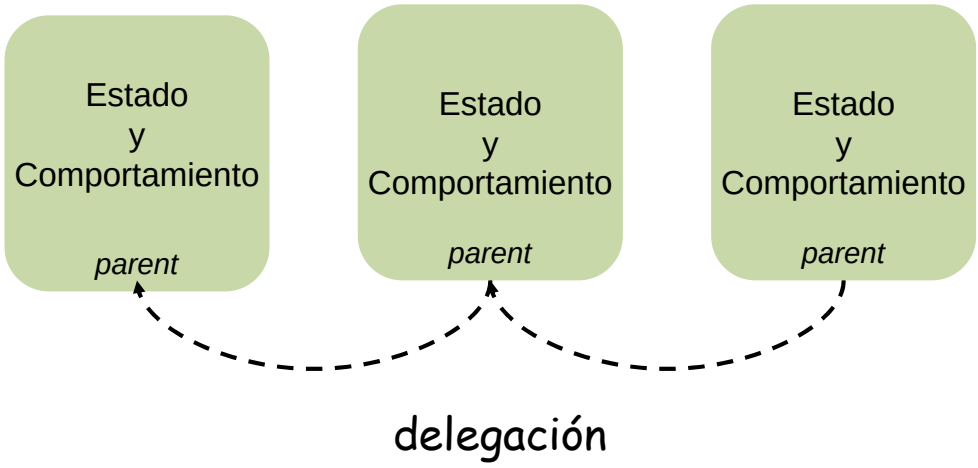
Métodos de Clasificación de Objetos

- Métodos de Clasificación de Objetos
  - Clases
    - Estructura y comportamiento en las clases
    - Instanciación de la clase
    - Lenguajes más difundidos (Java, C++, Eiffel, ...)
  - Prototipos

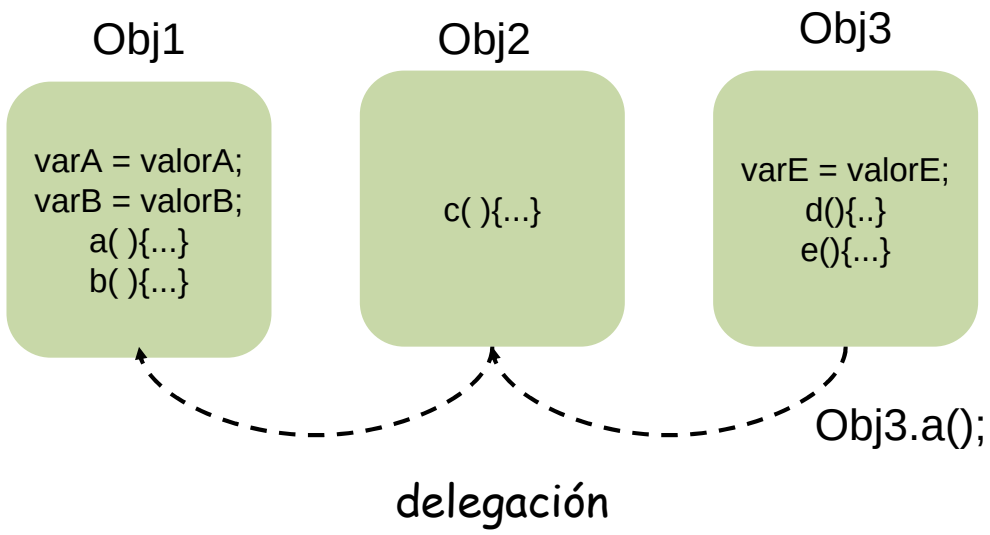
Métodos de Clasificación de Objetos

- Métodos de Clasificación de Objetos
  - Clases
  - Prototipos
    - No hay clases
    - Clonación de objetos
    - Estructura y comportamiento asociado a objetos individuales
    - Delegación y compartición
      - Puede ser dinámica
    - Más flexible (engloba el modelo de clases)
    - Lenguajes experimentales (Self, Zero, Cecil, Kevo, ...)

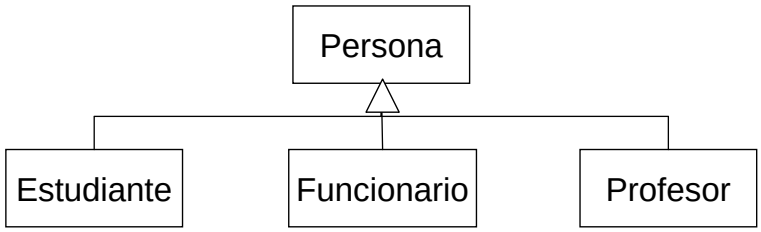
Prototipos

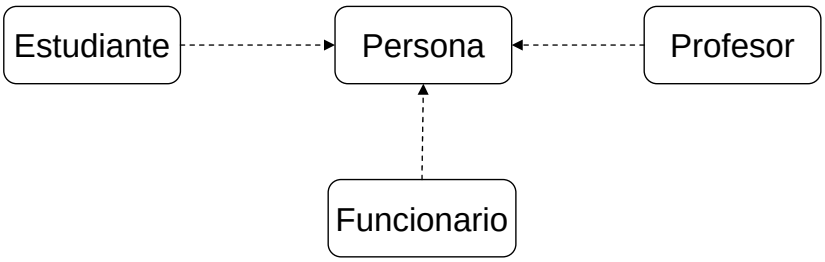


Prototipos

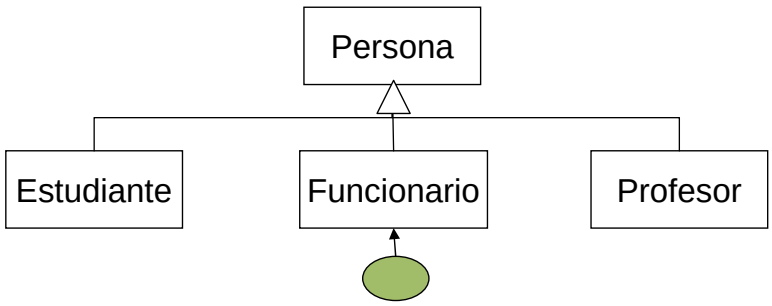


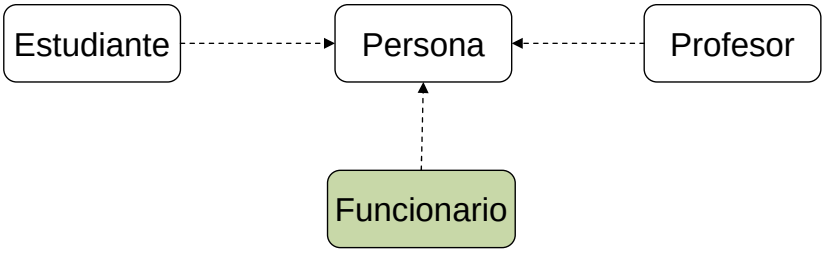
Herencia vs. Delegación



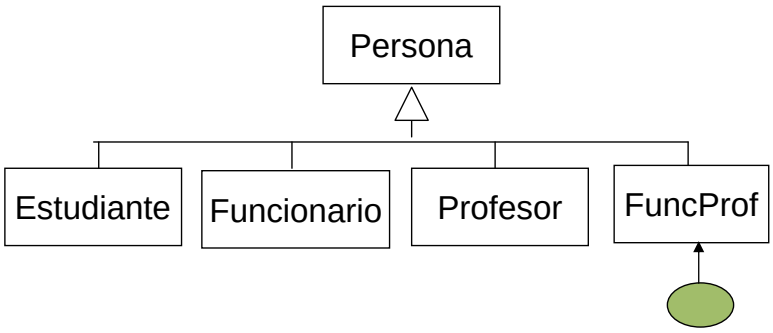


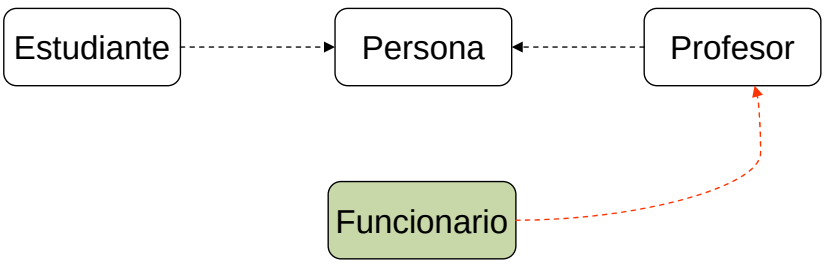
Herencia vs. Delegación



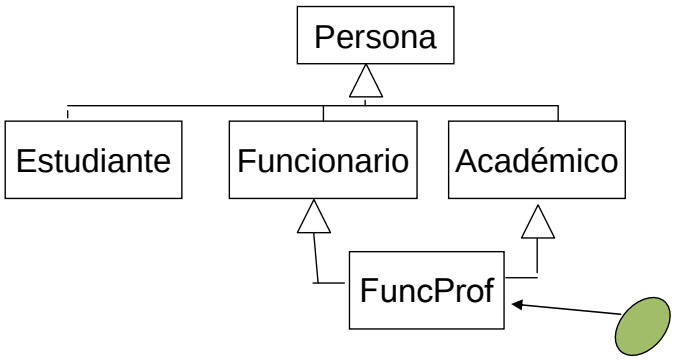


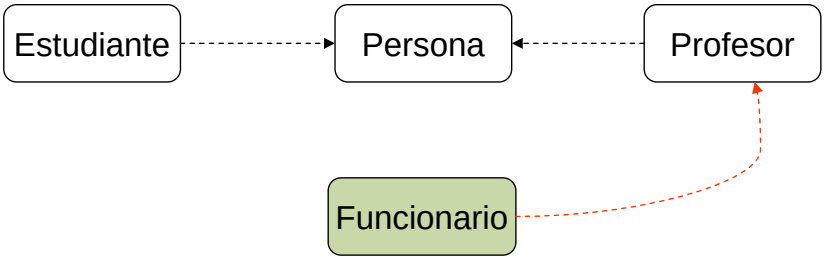
Herencia vs. Delegación





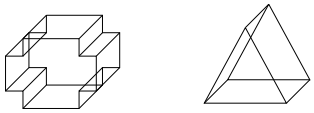
Herencia vs. Delegación



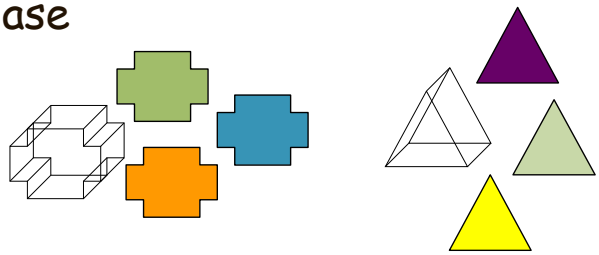


Concepto de Clase

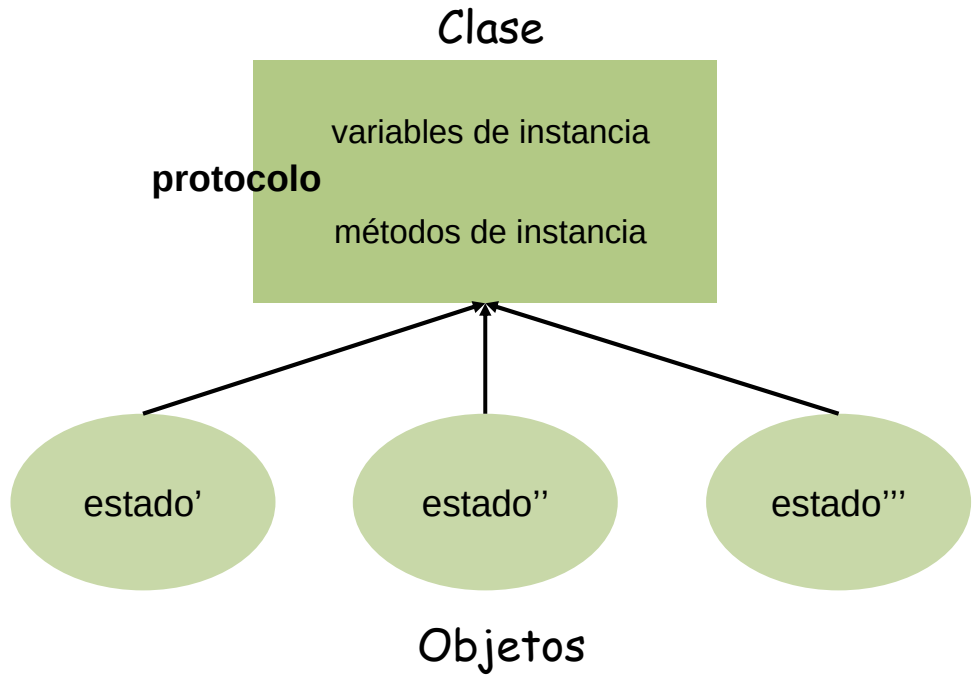
- Clase: Definición abstracta del comportamiento y estructura de un conjunto de objetos



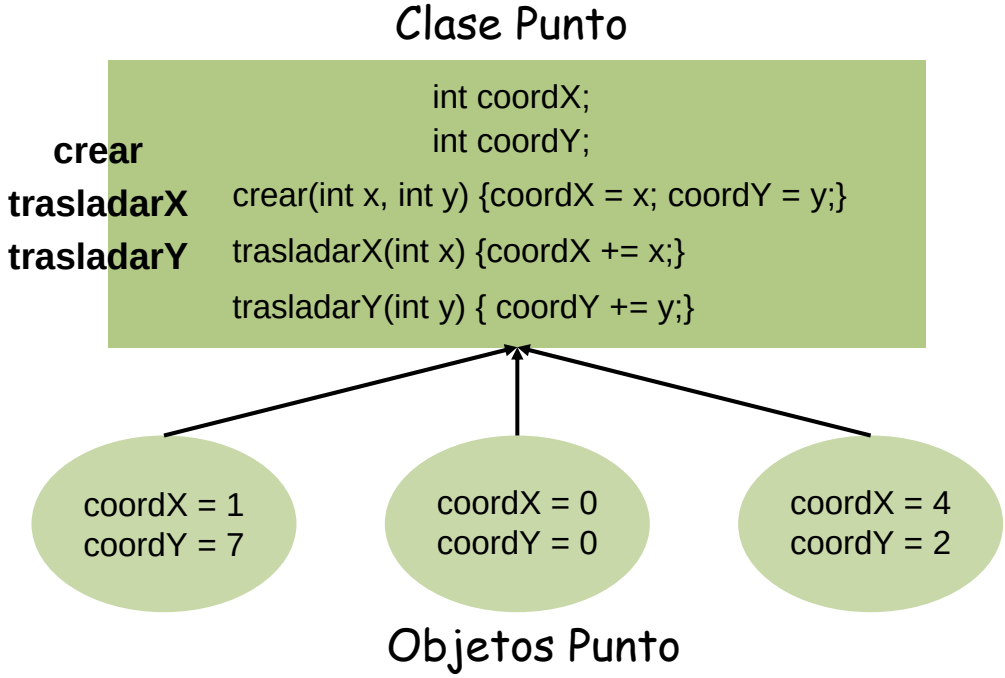
- Objeto: Ejemplo, representante o instancia de una clase



Concepto de Clase

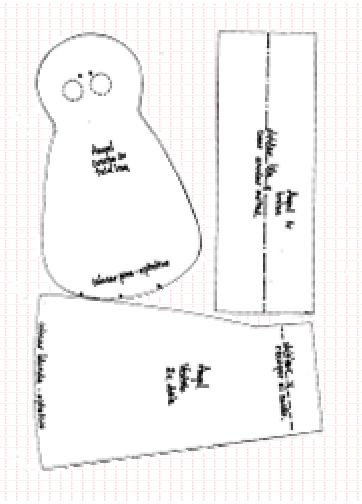


Concepto de Clase



Concepto de Clase

- Dos conceptos diferentes
  - Clases Objeto o metaclases
  - Clases Patrón (patrón fábrica)



Clases-Objeto

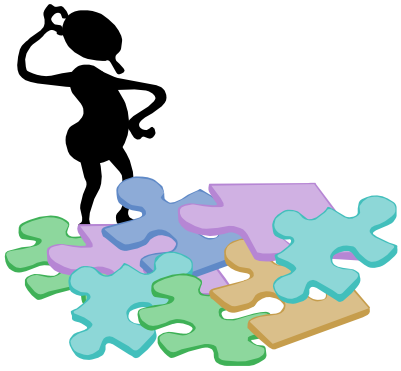
- Las clases son **Objetos**
  - Se les puede mandar mensajes
    - MÉTODOS DE CLASE

```
Object subclass: #Carta
        instanceVariableNames: 'palo numero'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'PDO'
```

Subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:

Clases-Objeto

- ¿Qué información almacena una clase-objeto en su estado?



Clases-Objeto

- ¿Qué información almacena una clase-objeto en su estado?
  - Variables de instancia (estructura, no valor)
  - Métodos de instancia
- Esta información le permite describir la estructura y comportamiento de los objetos
  - Se usa durante la construcción de éstos

Clases-Objeto

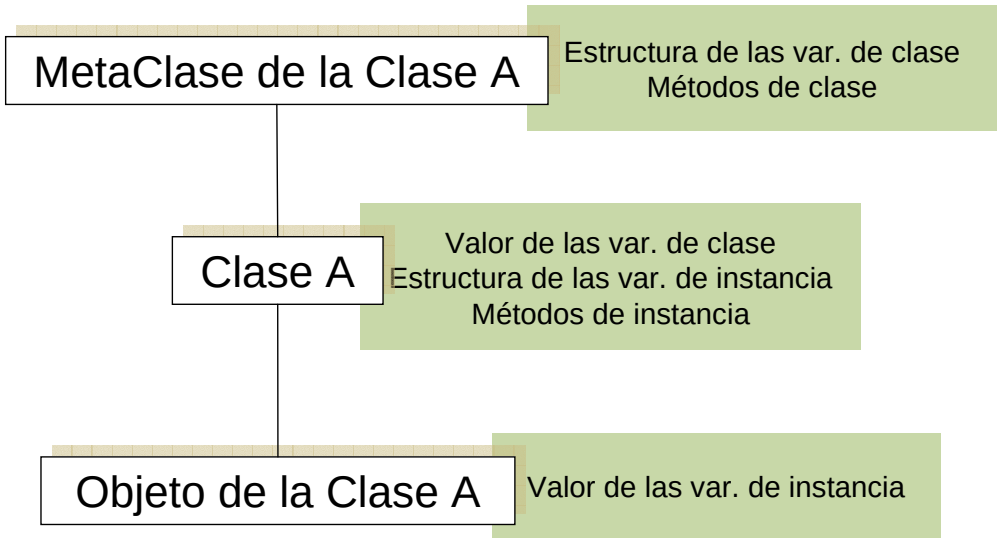
- La clase a la que pertenece una clase-objeto se llama MetaClase
- ¿Qué información almacena una MetaClase?
  - Variables de clase (estructura, no valor)
  - Métodos de clase
    - Por ejemplo, new
- Cada clase tiene su propia MetaClase

Clases-Objeto: creación

```
Object subclass: #Carta
        instanceVariableNames: 'palo numero'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'PDO'
```

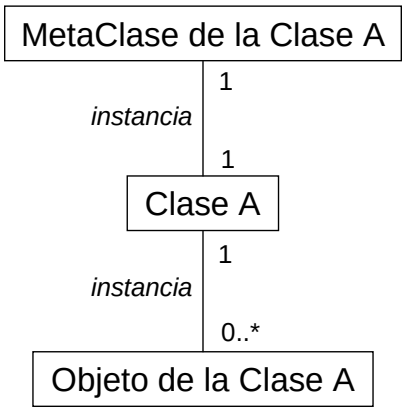
- Al ejecutarse este método se definen dos clases: *Carta* y la clase de la clase Carta (su metaclase *Carta class*). Por usarse el método de clases-objeto, se crean dos objetos que las representan:
  - La clase-objeto Carta (única instancia de Carta class)
  - La clase-objeto Carta Class, instancia de la clase Metaclass

Clases-Objeto

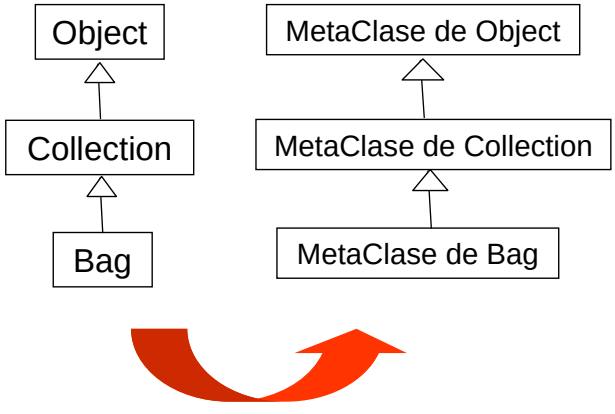


Clases-Objeto

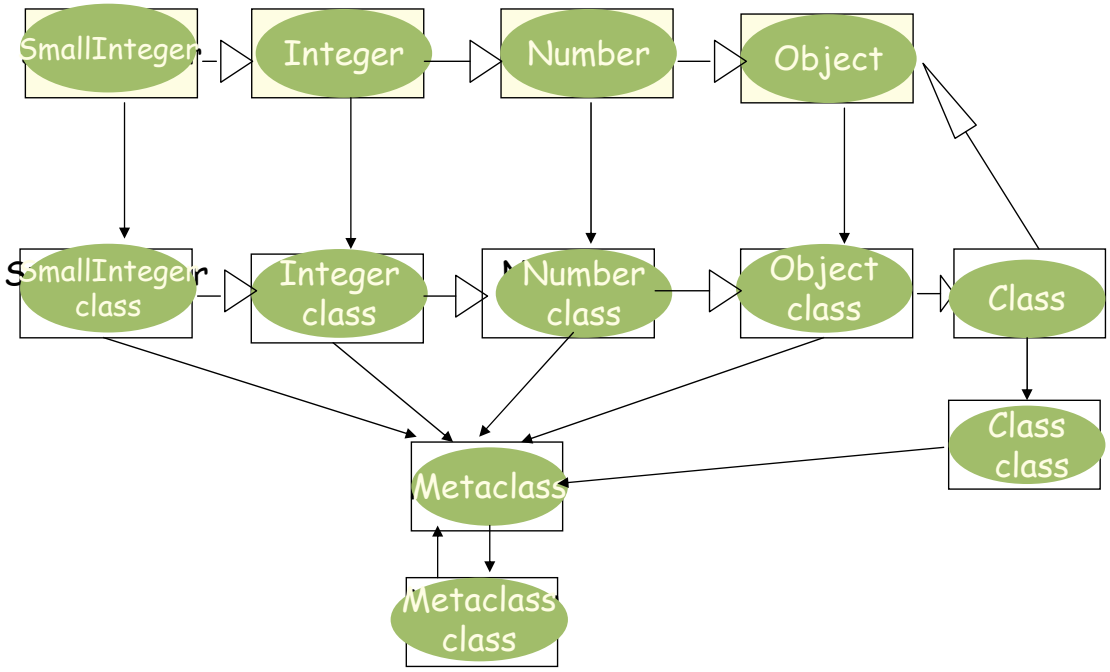
Instanciación



Herencia

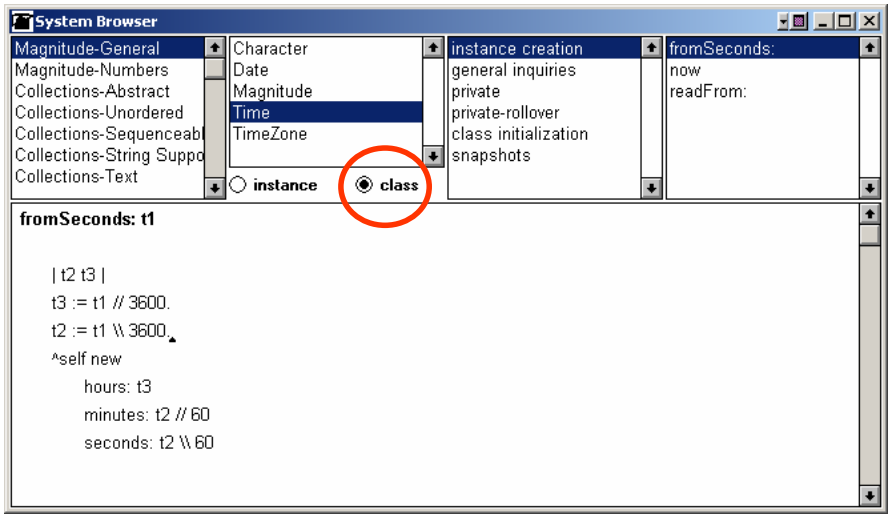


Clases-Objeto



Clases-Objeto

- Los entornos que usan MetaClases las ocultan al programador para simplificar



Clases-Patrón

- La clase es una **Definición Textual Estática**
  - Nombre de la clase
  - Declaración de las variables de instancia
  - Declaración del protocolo
  - Implementación de los métodos
- Lenguajes: Java, C++, Eiffel, ...

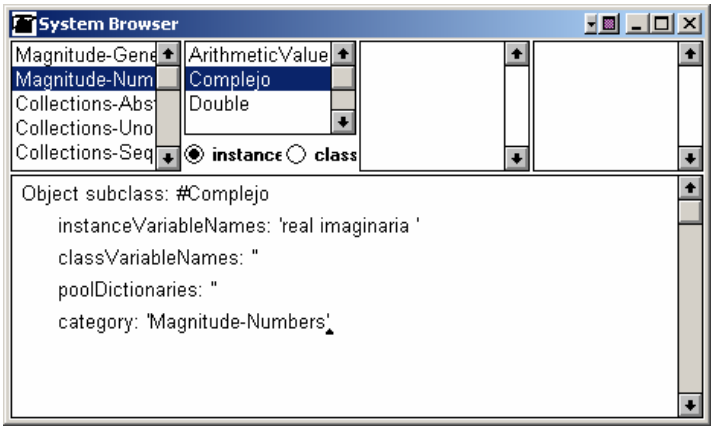
Creación e Inicialización de Objetos

- Creación e Inicialización
  - con Clases-Objeto: Métodos de clase
    - Ejemplo: Smalltalk
  - con Clases-Patrón: Constructores
    - Ejemplo: Java
- Cumplir el invariante de representación desde la creación
  - Propiedad que deben garantizar los objetos

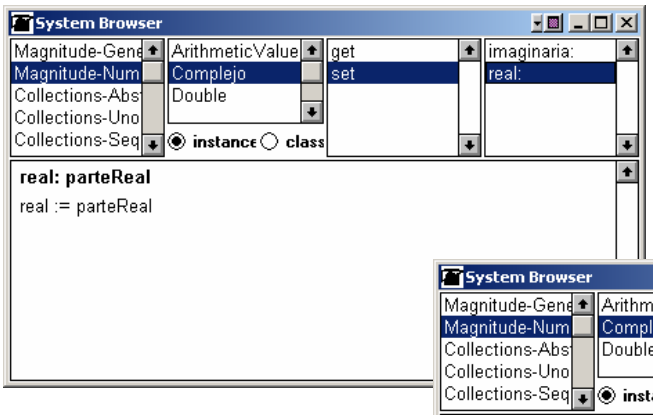
Creación con Clases-Objeto

- Los objetos se crean mandando un mensaje a la clase
  - Ese mensaje, normalmente *new*, se resuelve con un método de clase

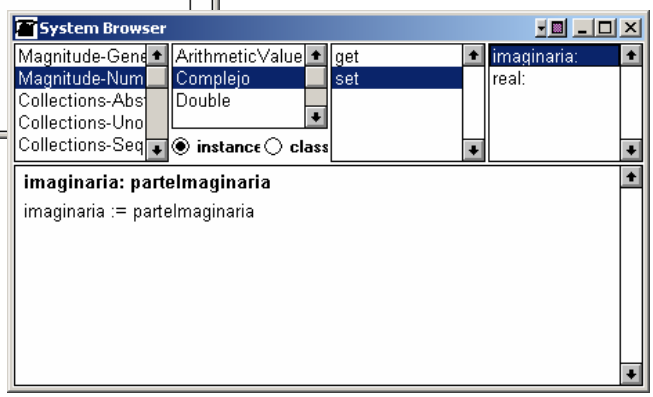
Crear un método para crear e inicializar un objeto de la clase Complejo



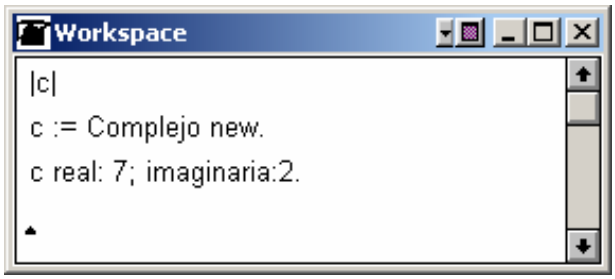
Creación con Clases-Objeto



1. Métodos de instancia para modificar las var. de instancia



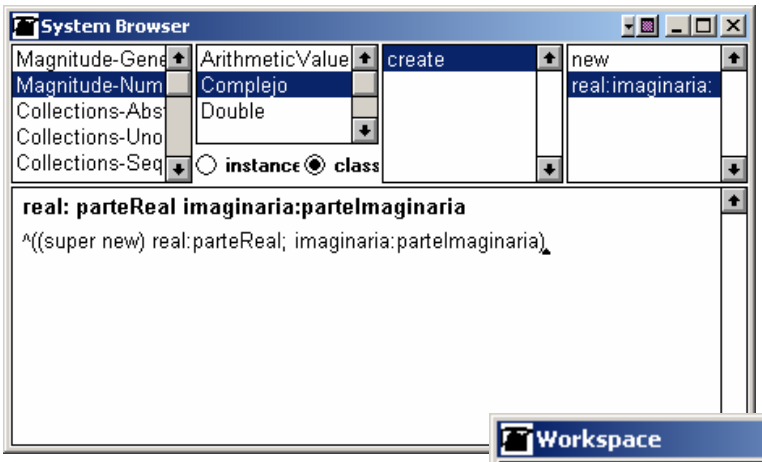
Creación con Clases-Objeto



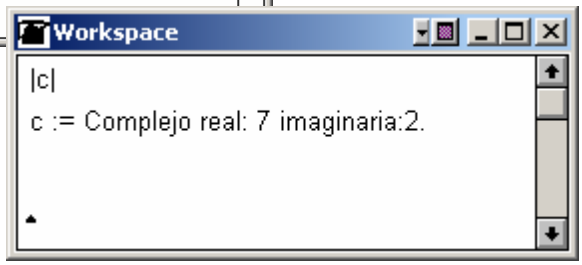
2a. Primero crear y después inicializar con métodos de instancia

- Atención: Puede no cumplirse el invariante de representación

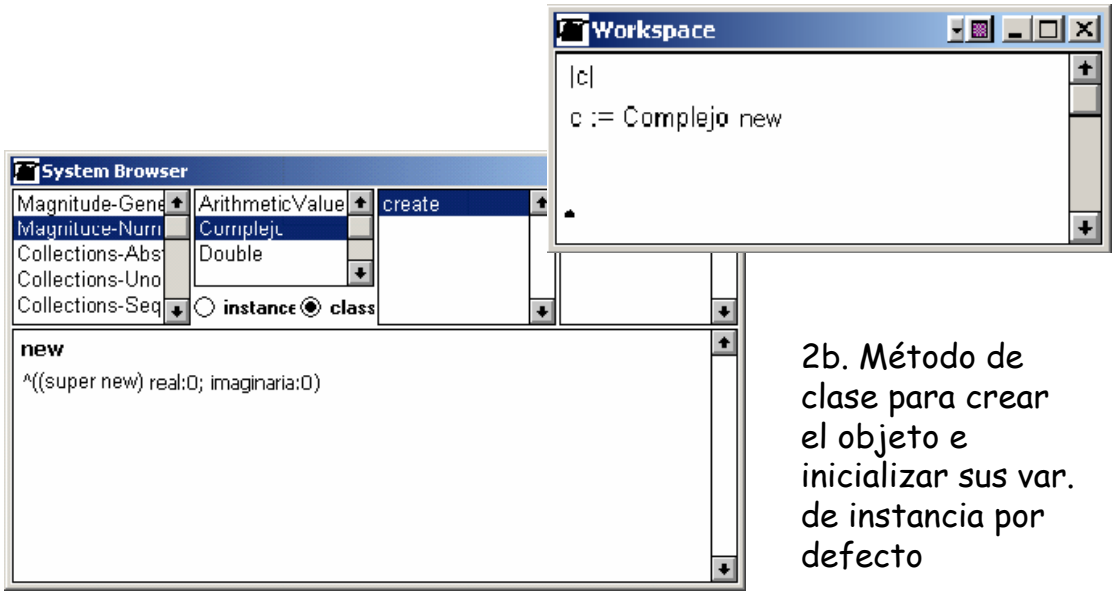
Creación con Clases-Objeto



2b. Método de clase para crear el objeto e inicializar sus var. de instancia



Creación con Clases-Objeto



2b. Método de clase para crear el objeto e inicializar sus var. de instancia por defecto

Creación con Clases-Patrón

- En C++
  - Stack

    Complejo c;
  - Heap

    *Complejo * c;*

    c = new Complejo();
- En Java

  Complejo c;

  c = new Complejo();

Creación con Clases-Patrón

```
public class Complejo {
    float real;
    float imaginaria;

   public Complejo(float real, float imaginaria) {
       this.real = real;
       this.imaginaria = imaginaria;
    }

    public Complejo() {
       this(0,0);
    }
    ...
}

Complejo c = new Complejo(5,2);
```


Uso de métodos y variables de clase con Clases-Patrón

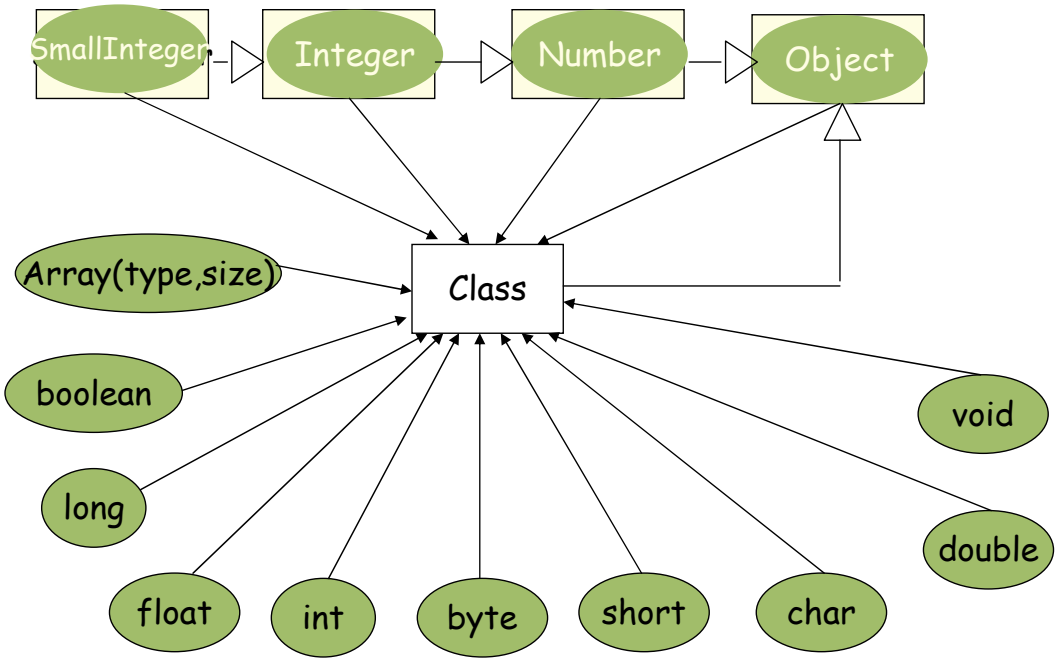
```
class claseA {
    static int  x;
    static int valorx() {return x;}
    private: int y;
};
int claseA::x=1;
claseA::valorx();
ClaseA a;
a.valorx();
```

Dos formas de acceder a un método de clase en C++

Una Metaclase rara: La clase Class en Java

- Para cada clase en Java es posible obtener un objeto de la clase Class que lo representa
  - Obtener el objeto Class
  - Usar el objeto Class
    - Nombre de la clase
    - Modificadores de la clase
    - Superclases
    - Interfaces que implementa
    - Variables miembro
    - Constructores y Métodos

Clase y objetos Class



La clase Class en Java: diferencias con el uso de objetos-clase

- No es posible usar constructores con la sintaxis para los métodos de instancia. Excepción: método newInstance. Ej. Array.newInstance(class, tam). En los lenguajes con objetos-clase los constructores no son más que un método de clase y no se requiere ninguna palabra clave para utilizarlos (como *new* en Java y C++).
- No es posible crear nuevas clases en tiempo de ejecución o añadir métodos o variables a clases existentes.

Una Metaclase rara: La clase Class en Java

- Obtener el objeto Class
  - getClass()

```
Complejo n = new Complejo(3, 1);
Class mc = n.getClass();
```

  - getSuperclass()

```
Class mc2 = mc.getSuperclass();
```

  - .class

```
Class mc = Complejo.class;
Class mc3 = java.util.ArrayList.class;
```

  - forName()

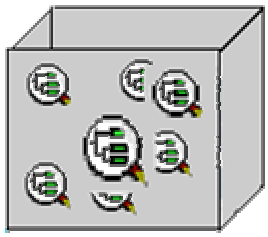
```
Class mc = Class.forName("Complejo");
```

Una Metaclase rara: La clase Class en Java

- Usar el objeto Class

```
Complejo n = new Complejo(3, 1);
Class mc = n.getClass();
String nombre = mc.getName();
Class superClase = mc.getSuperclass();
int modificadores = mc.getModifiers();
Class[] interfaces = mc.getInterfaces();
Field[] variablesMiembro = mc.getFields();
Constructor[] constructores = mc.getConstructors();
Method[] metodos = mc.getMethods();
```

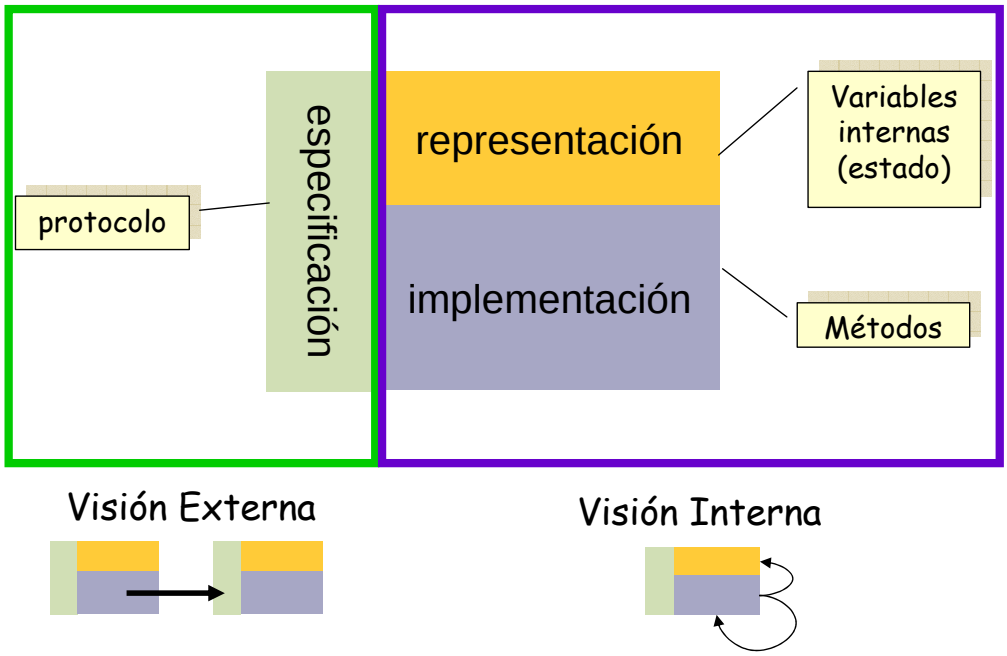
Tema 3



Lección 2

Clases en Smalltalk

Visiones de una Clase



Visión Externa Smalltalk

- Nombre de la clase
  - Variable global
- Protocolo
  - Patrones de mensaje
    - Selector
    - Argumentos
      - Sin tipo
    - Especificación
      - Lenguaje natural

at: pos put: val.

Selector → at: put:

Argumentos → pos, val

Especificación → “coloca el valor **val** en la posición **pos**”

Especificación

- Cláusula de requisitos (requires)
- Efecto (effect)
- Elementos modificados (modifies)
  - Efectos secundarios
- Valor devuelto (answer)
- Autor (author)
- Versión (version)
- Fecha (date)

Especificación

```
Object subclass: #Stack
        instanceVariableNames: 'elems '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'HEDES-aux'!
```

- Cláusula de requisitos
- Efecto
- Elementos modificados
  - Efectos secundarios
- Valor devuelto
- Autor
- Fecha
- Versión

comentario

```
isEmpty
"       Abstract: Yes/No
        Requires:
        Effect: determines if the receiver is empty
        Modifies:
        Answer: true or false
        Author: Maria Jose
        Version: 1
        Date:May 23, 1998"

^(elems isEmpty).
```


Especificación

```
Object subclass: #Stack
        instanceVariableNames: 'elems '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'HEDES-aux'!
```

- Cláusula de requisitos
- Efecto
- Elementos modificados
  - Efectos secundarios
- Valor devuelto
- Autor
- Fecha
- Versión

```
pop
"       Abstract: Yes/No
        Requires:
        Effect: removes the element of the top of the stack
        Modifies: elems
        Answer: the element of the top
        Author: Maria Jose
        Version: 1
        Date:May 23, 1998"

(elems isEmpty)  ifFalse: [^elems removeFirst.]
```

Especificación

```
Object subclass: #Stack
        instanceVariableNames: 'elems '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'HEDES-aux'!
```

- Cláusula de requisitos
- Efecto
- Elementos modificados
  - Efectos secundarios
- Valor devuelto
- Autor
- Fecha
- Versión

```
push: anElement
"       Abstract: Yes/No
        Requires:
        Effect: add a new element on the top of the stack
        Modifies: elems
        Answer:
        Author: Maria Jose
        Version: 1
        Date:May 23, 1998"

elems addFirst: anElement.
```


Especificación

```
Object subclass: #Stack
        instanceVariableNames: 'elems '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'HEDES-aux'!
```

- Cláusula de requisitos
- Efecto
- Elementos modificados
  - Efectos secundarios
- Valor devuelto
- Autor
- Fecha
- Versión

```
initialize
"       Abstract: Yes/No
        Requires:
        Effect: initialize instance variables
        Modifies: elems
        Answer:
        Author: Maria Jose
        Version: 1"

elems:= OrderedCollection new.
```

Especificación

```
Object subclass: #Stack
        instanceVariableNames: 'elems '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'HEDES-aux'!
```

- Cláusula de requisitos
- Efecto
- Elementos modificados
  - Efectos secundarios
- Valor devuelto
- Autor
- Fecha
- Versión

```
new
"       Abstract: Yes/No
        Requires:
        Effect:
        Modifies: creates a new Stack
        Answer: a new Stack
        Author: Maria Jose
        Version: 1"

^(super new) initialize.
```


Visión Interna Smalltalk

- Representación
  - Nombre de la clase
  - Nombre de la superclase
  - Variables de instancia
  - Variables de clase
  - Variables semi-globales
  - Categoría
- Implementación
  - Métodos de instancia
  - Métodos de clase

Representación

- Nombre de la clase
- Nombre de la superclase
- **Variables de instancia**
- Variables de clase
- Variables semi-globales
- Categoría

```
Object subclass: #Stack
        instanceVariableNames: 'elems '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'HEDES-aux'
```


Variables de Instancia

- Nombradas
- Indexadas

Variables de Instancia

- Nombradas
  - Definición de la clase

```
Object subclass: #Stack
    instanceVariableNames: 'elems'
    classVariableNames: ''
    poolDictinaries: ''
    category: 'HEDES-aux'
```

  - Creación del objeto

```
unaPila := Stack new.
```

  - Acceso a la variable
    - Por su nombre

Variables de Instancia

- Indexadas
  - Definición de la clase

```
ArrayedCollection variableSubclass: #Array
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictinaries: ''
    category: 'Collections-Arrayed'
```

  - Creación del objeto

```
nombres := Array new: 10
```

  - Acceso a la variable
    - Con los métodos at: y at: put:
      - nombres at:1 put:'juan'

Implementación

- Métodos de instancia
- Métodos de clase

selector y argumentos — *Patrón del mensaje*

"especificación"

| declaración de var. temporales |

cuerpo — *Secuencia de expresiones de mensaje*

^(valor devuelto) — *Opcional*

Selector Unario

```
Object subclass: #Punto
instanceVariableNames: 'coordX coordY'
classVariableNames: ''
poolDictinaries: ''
category: 'EjemploPDO'
```

```
invertirXeY
"intercambia las coordenadas del punto "
| t |
t := coordX.
coordX := coordY.
coordY := t.
```

Selector Binario

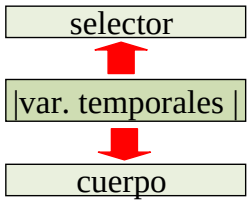
```
Object subclass: #Complejo
        instanceVariableNames: 'real imaginaria'
        classVariableNames: ''
        poolDictinaries: ''
        category: 'Magnitude-Numbers'
```

```
+ unComplejo
"Devuelve un Complejo igual a la suma del argumento y el receptor"
| sumParteReal sumParteImaginaria |
sumParteReal := real + unComplejo real.
sumParteImaginaria := imaginaria + unComplejo imaginaria.
^Complejo real: sumParteReal imaginaria: sumParteImaginaria.
```


Selector Palabras Clave

Argumentos y Var. Temporales

- Argumentos
  - Pseudovariables
  - Efectos secundarios
- Variables temporales
  - Estado transitorio
  - Inicialización a nil
  - Existencia ligada a la activación del método



Métodos

- Métodos de Instancia
- Métodos de Clase
- Métodos Primitivos

Métodos de Instancia

```
isEmpty
"       Abstract: Yes/No
        Requires:
        Effect: determines if the receiver is empty
        Modifies:
        Answer: true or false
        Author: Maria Jose
        Version: 1
        Date:May 23, 1998"

^(elems isEmpty).
```

```
pop
"       Abstract: Yes/No
        Requires:
        Effect: removes the element of the top of the stack
        Modifies: elems
        Answer: the element of the top
        Author: Maria Jose
        Version: 1
        Date:May 23, 1998"

(elems isEmpty)  ifFalse: [^elems removeFirst.]
```


Métodos de Instancia

```
push: anElement
"       Abstract: Yes/No
        Requires:
        Effect: add a new element on the top of the stack
        Modifies: elems
        Answer:
        Author: Maria Jose
        Version: 1
        Date:May 23, 1998"

elems addFirst: anElement.
```

```
initialize
"       Abstract: Yes/No
        Requires:
        Effect: initialize instance variables
        Modifies: elems
        Answer:
        Author: Maria Jose
        Version: 1"

elems:= OrderedCollection new.
```

Métodos de Clase

```
new
"       Abstract: Yes/No
        Requires:
        Effect:
        Modifies: creates a new Stack
        Answer: a new Stack
        Author: Maria Jose
        Version: 1"

^(super new) initialize.
```

Métodos Primitivos

- Métodos implementados en la máquina física
  - Forman parte de la máquina virtual
- Métodos que no llaman a otros

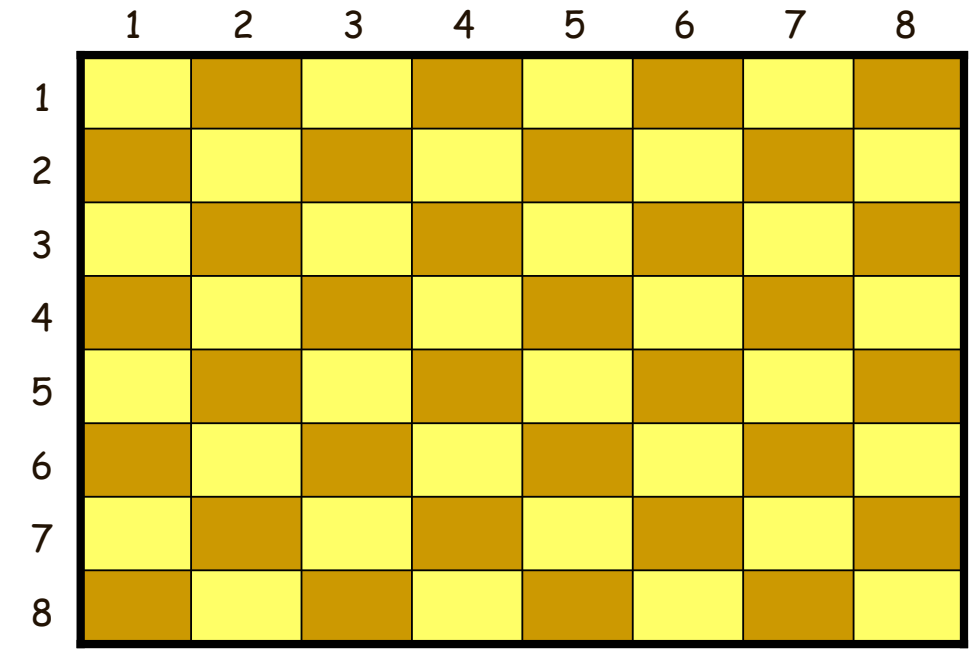
Ejemplos

- Ocho Reinas
- Contabilidad Macana

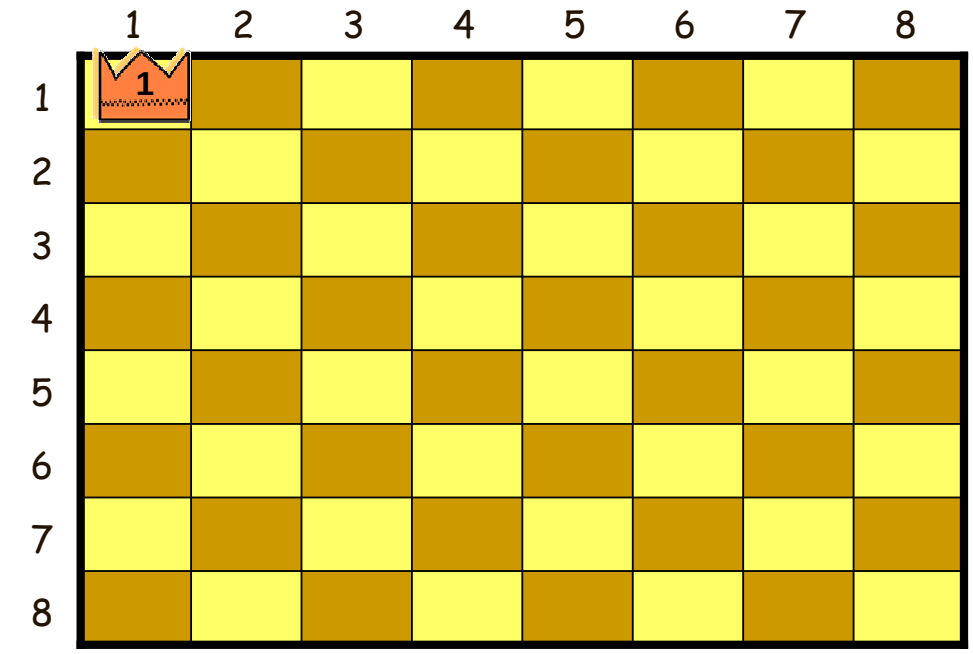
Ocho Reinas

```
|reina ultimaReina juego|
ultimaReina:= ReinaNula new.
juego:=Array new: 8.
1 to: 8 do: [ :i | reina:= Reina new.
reina asignarColumna: i vecina: ultimaReina.
juego at:i put: reina.
ultimaReina:= reina.].
reina primera.
juego inspect.
```

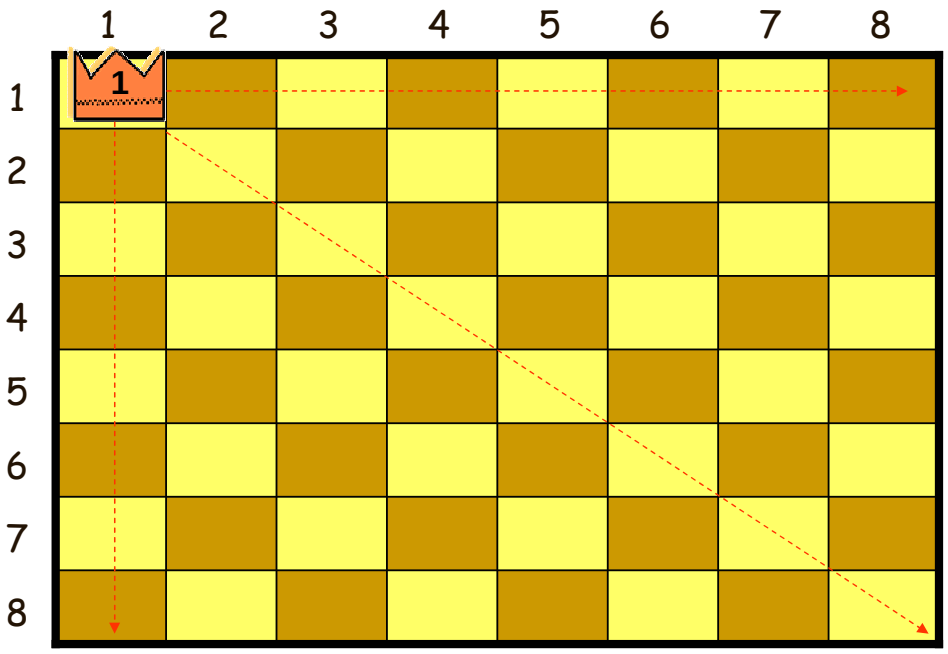
Ocho Reinas



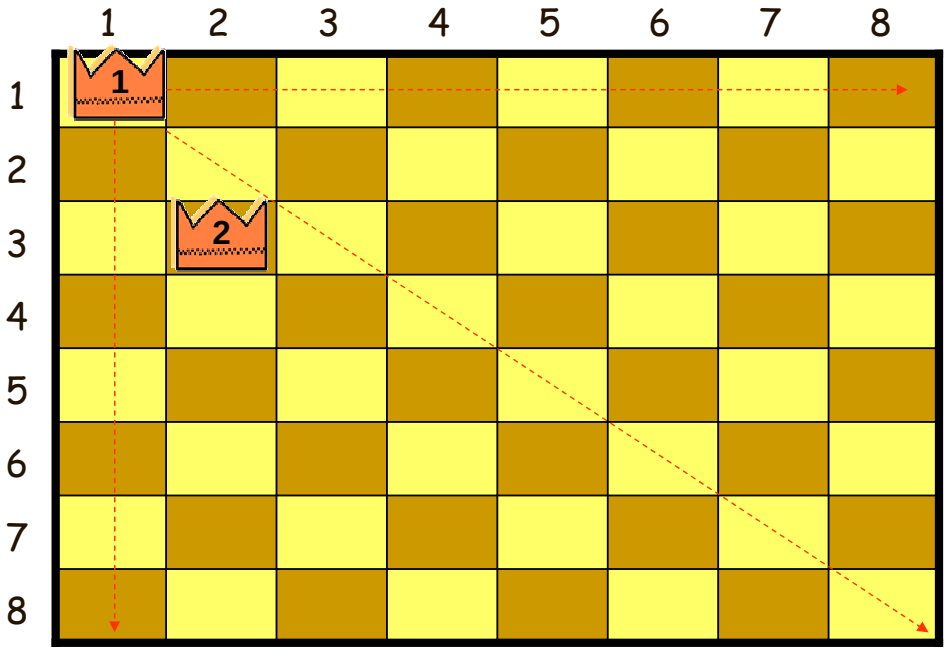
Ocho Reinas



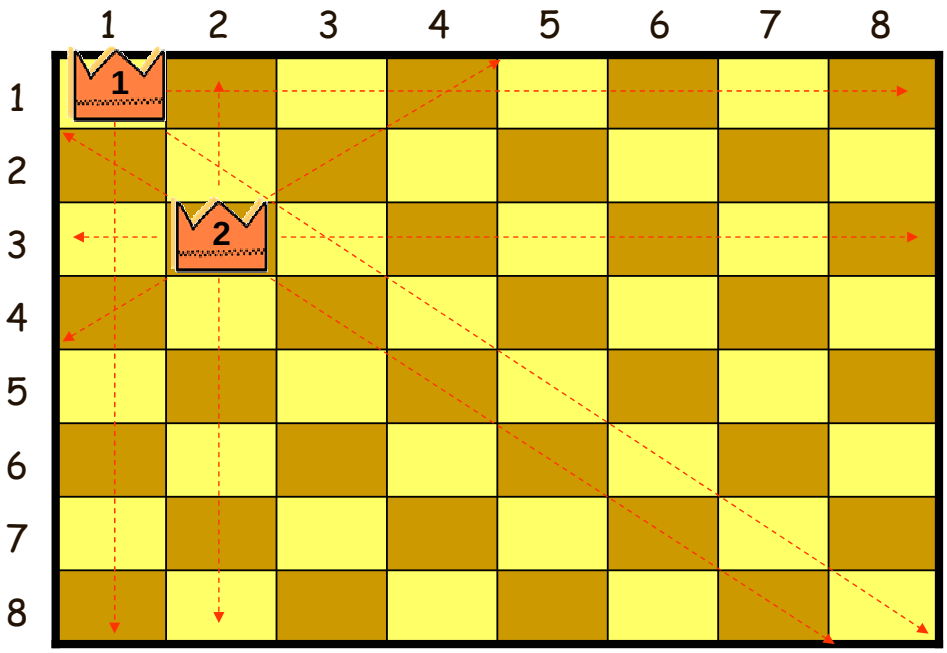
Ocho Reinas



Ocho Reinas



Ocho Reinas



Contabilidad Macana

```
Object subclass: #ContabilidadMacana
        instanceVariableNames: 'debe haber saldo '
        classVariableNames: 'TotalSaldos '
        poolDictionaries: ''
        category: 'PdoEjemplos'!
```

Contabilidad Macana

```
| conta1 conta2|
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 2000.
conta2 ingresaEnCapitulo: 'Trabajo' laCantidad: 2000.
conta1 consultaAdeudo: 'Juegos'.
conta1 consultaIngreso: 'Juegos'.
conta1 adeudaEnCapitulo: 'Cubatas' laCantidad: 5000.
conta1 saldo.
ContabilidadMacana totalSaldos.! !
```


Contabilidad Macana

```
| conta1 conta2|
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200
```

```
iniciaTotalSaldos
"       Abstract: No
        Requires:
        Effect: inicializa el total de saldos de todas las contabilidades.
        Modifies:
        Answer:
        Author: JosÃ© Parets
        Version: 1.0
        Date:January 14, 2002"

        TotalSaldos := 0.!
```

```
l: 1000.
l: 2000.
d: 2000.

d: 5000.
ContabilidadMacana totalSaldos.! !
```

Contabilidad Macana

#ContabilidadMacana

TotalSaldos = 0

Contabilidad Macana

```
| conta1 conta2|
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 2000.
```

```
new:anInteger
"       Abstract: No
        Requires:
        Effect: crea un objeto inicializado con saldo inicial anInteger
        Modifies:
        Answer:
        Author: JosÃ© Parets
        Version: 1.0
        Date:January 14, 2002"

        ^super new initialize: anInteger.! !
```

Contabilidad Macana

```
| conta1 conta2|
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
```

antidad: 2000.

Cantidad: 2000.

Cantidad: 5000.

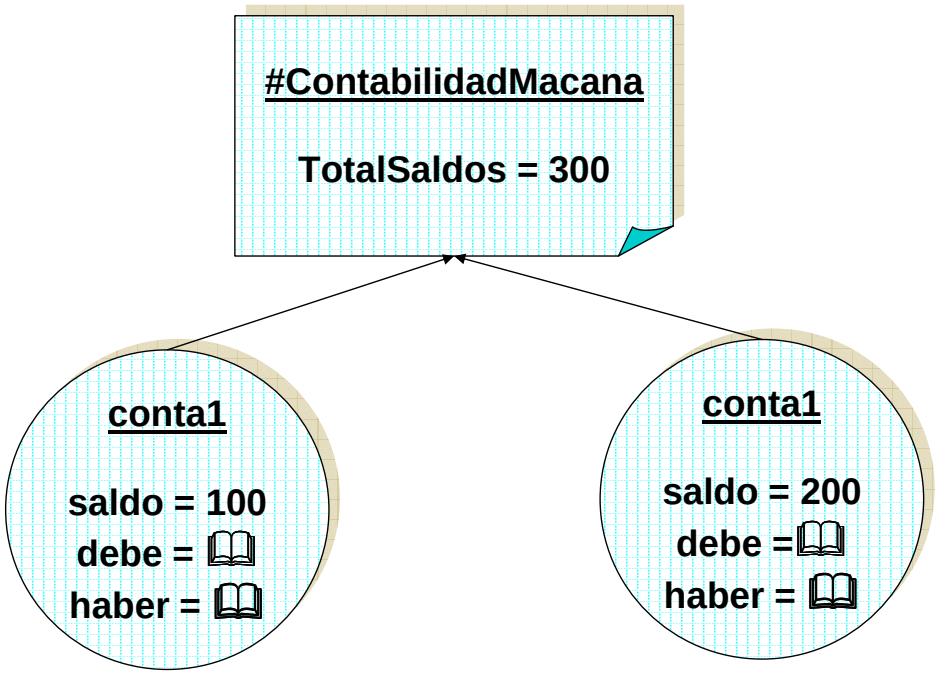
```
initialize: anInteger
"       Abstract: Yes/No
        Requires:
        Effect: inicializa el receptor con saldo a anInteger
        Modifies:
        Answer:
        Author:
        Version:
        Date:January 14, 2002"
        saldo := anInteger.
        haber := Dictionary new.
        debe := Dictionary new.
        TotalSaldos := TotalSaldos + anInteger.! !
```


Contabilidad Macana

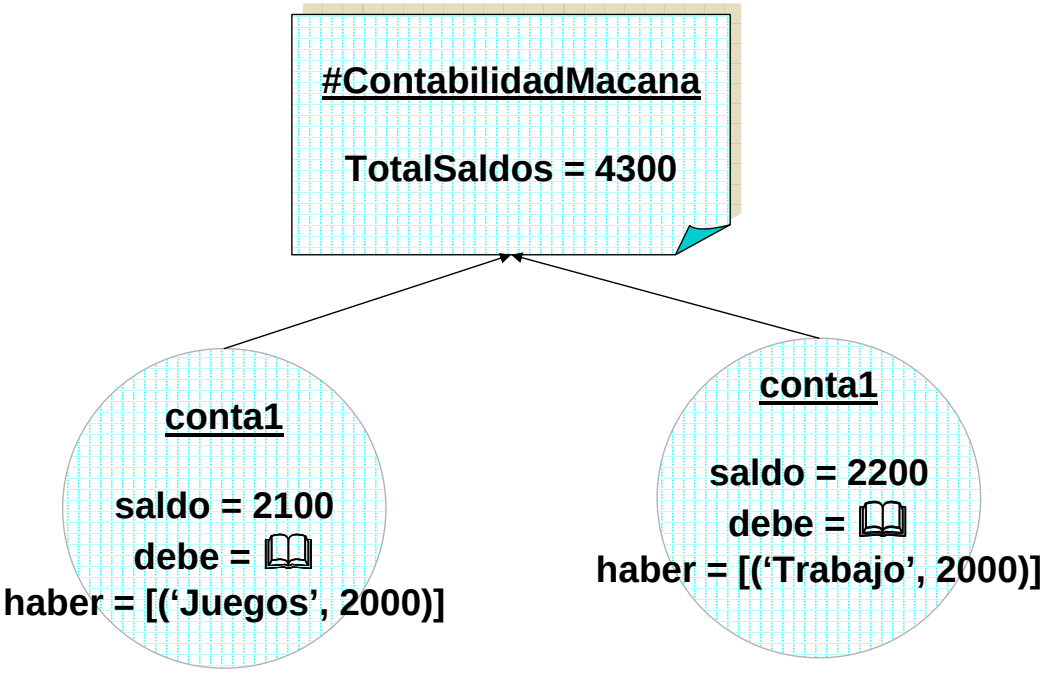
```
| conta1 conta2|
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 2000.
conta2 ingresaEnCapitulo: 'Trabajo' laCantidad: 2000.
conta1 consult
conta1 consult
conta1 adeuda
conta1 saldo.
ContabilidadM
```

```
ingresaEnCapitulo: unString laCantidad: unInteger
"       Abstract: Yes/No
        Requires:
        Effect: Ingresa en el capÃ-tulo la cantidad unInteger
        Modifies:
        Answer:
        Author: JosÃ© Parets
        Version: 1.0
        Date:January 14, 2002"

        haber at: unString ifAbsent: [haber at: unString put: 0].
        haber at: unString put: (haber at: unString) + unInteger.
        saldo := saldo + unInteger.
        TotalSaldos := TotalSaldos + unInteger.! !
```

Contabilidad Macana



Contabilidad Macana

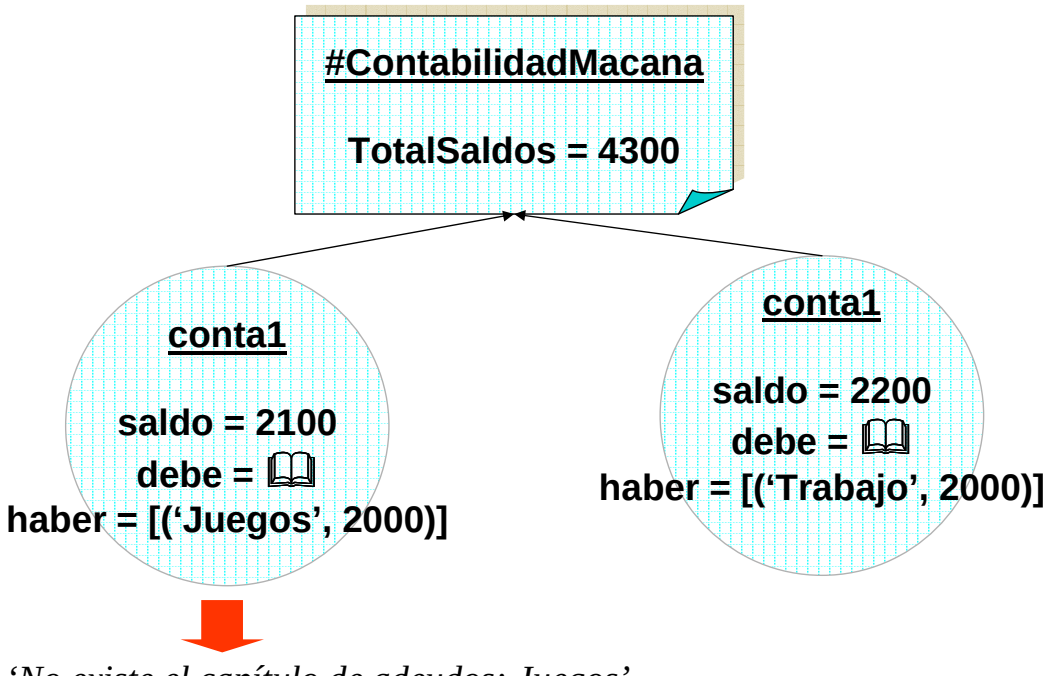


Contabilidad Macana

```
| conta1 conta2|
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 2000.
conta2 ingresaEnCapitulo: 'Trabajo' laCantidad: 2000.
conta1 consultaAdeudo: 'Juegos'.
conta1 consultaIngreso: 'Juegos'
```

```
consultaAdeudo: unString
"       Abstract: Yes/No
        Requires:
        Effect: Consulta el total de adeudos en el capÃ-tulo  unString
        Modifies:
        Answer:
        Author: JosÃ© Parets
        Version: 1.0
        Date:January 14, 2002"

        ^ debe at: unString ifAbsent: ['No existe el capÃ-tulo de adeudos: ', unString].
```


Contabilidad Macana



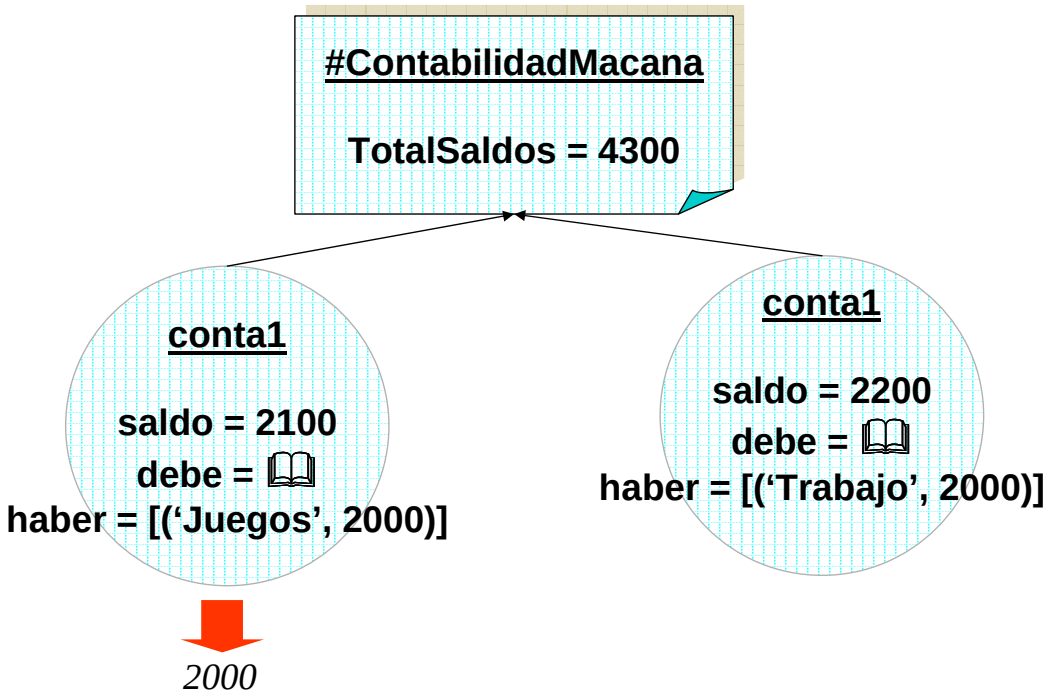
'No existe el capítulo de adeudos: Juegos'

Contabilidad Macana

```
| conta1 conta2 |
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 2000.
conta2 ingresaEnCapitulo: 'Trabajo' laCantidad: 2000.
conta1 consultaAdeudo: 'Juegos'.
conta1 consultaIngreso: 'Juegos'.
conta1 adeudaEnCapitulo: 'Cubatas' laCantidad: 5000.
```

```
consultaIngreso: unString
"       Abstract: Yes/No
        Requires:
        Effect: Consulta el total de adeudos en el capÃ-tulo  unString
        Modifies:
        Answer:
        Author: JosÃ© Parets
        Version: 1.0
        Date:January 14, 2002"

        ^ haber at: unString ifAbsent: ['No existe el capÃ-tulo de haberes: ', unString].
```


Contabilidad Macana

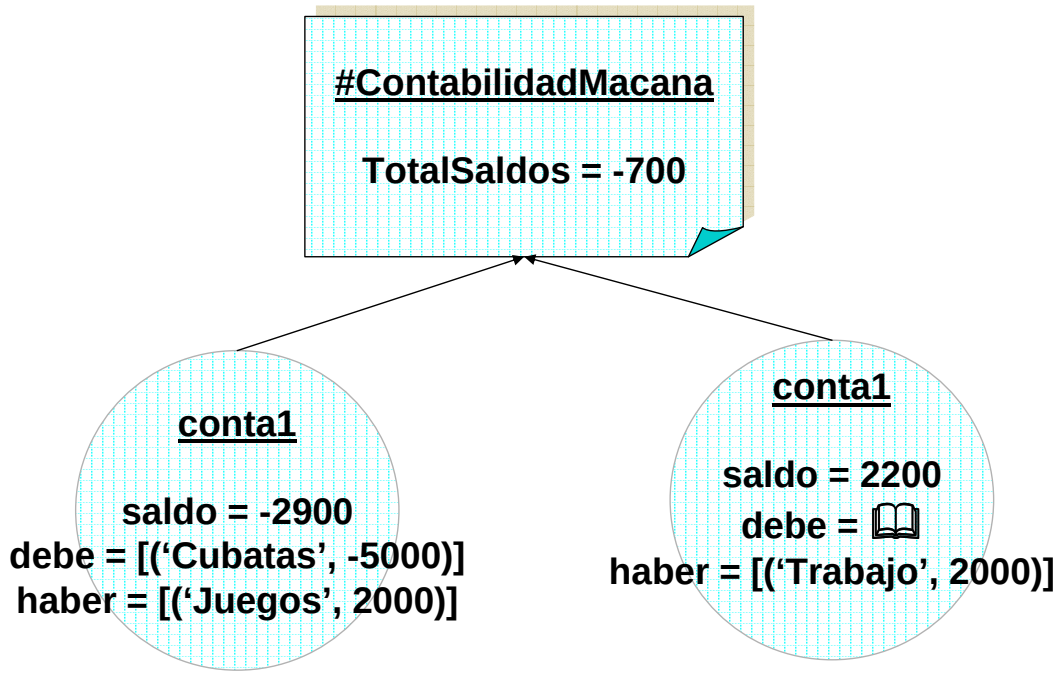


Contabilidad Macana

```
| conta1 conta2|
ContabilidadMacana inicializaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 2000.
conta2 ingresaEnCapitulo: 'Trabajo' laCantidad: 2000.
conta1 consultaAdeudo: 'Juegos'.
conta1 consultaIngreso: 'Juegos'.
conta1 adeudaEnCapitulo: 'Cubatas' laCantidad: 5000.
conta1 saldo.
ContabilidadM
```

```
adeudaEnCapitulo: unString laCantidad: unInteger
"       Abstract: Yes/No
        Requires:
        Effect: Adeuda en el capÃ-tulo la cantidad unInteger
        Modifies:
        Answer:
        Author: JosÃ© Parets
        Version: 1.0
        Date:January 14, 2002"

        debe at: unString ifAbsent: [debe at: unString put: 0].
        debe at: unString put: (debe at: unString) - unInteger.
        saldo := saldo - unInteger.
        TotalSaldos := TotalSaldos - unInteger.! !
```


Contabilidad Macana

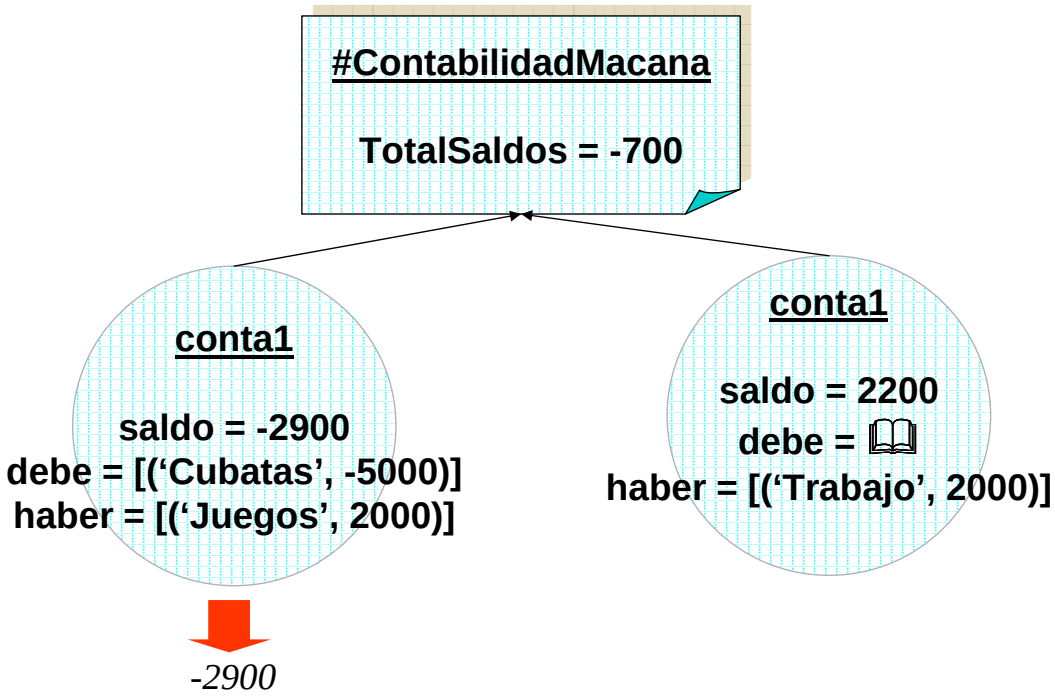


Contabilidad Macana

```
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 2000.
conta2 ingresaEnCapitulo: 'Trabajo' laCantidad: 2000.
conta1 consultaAdeudo: 'Juegos'.
conta1 consultaIngreso: 'Juegos'.
conta1 adeudaEnCapitulo: 'Cubatas' laCantidad: 5000.
conta1 saldo.
ContabilidadMacana totalSaldos.
```

```
saldo
"       Abstract: Yes/No
        Requires:
        Effect: Consulta el saldo.
        Modifies:
        Answer:
        Author: JosÃ© Parets
        Version: 1.0
        Date:January 14, 2002"

        ^saldo
```


Contabilidad Macana

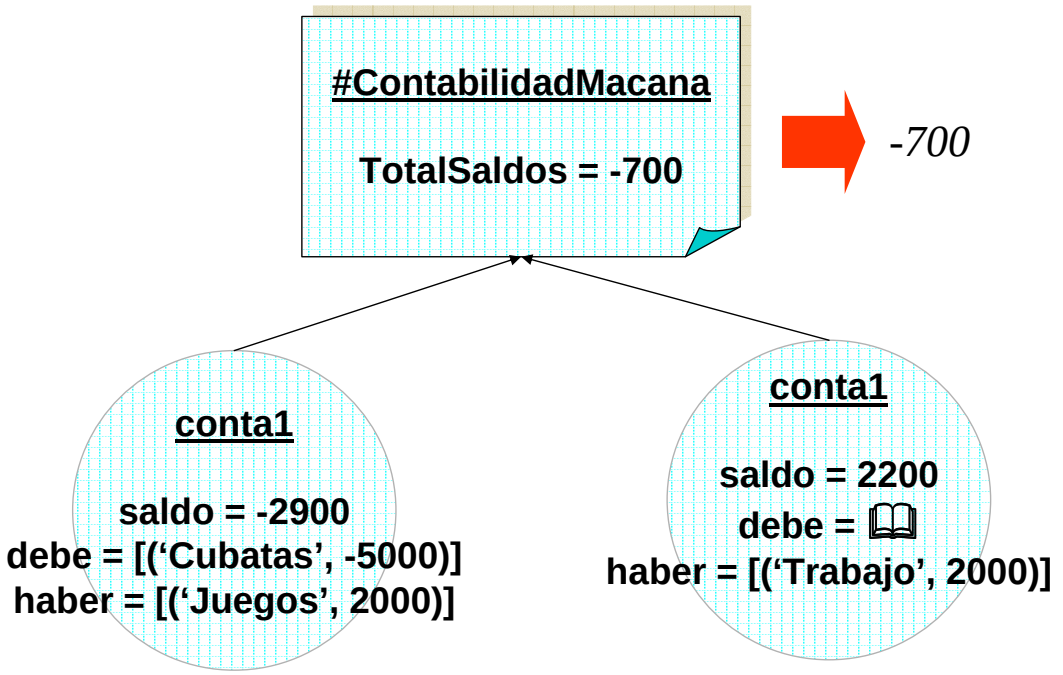


Contabilidad Macana

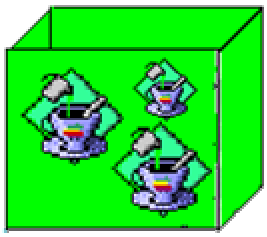
```
ContabilidadMacana iniciaTotalSaldos.
conta1 := ContabilidadMacana new:100.
conta2 := ContabilidadMacana new:200.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 1000.
conta1 ingresaEnCapitulo: 'Juegos' laCantidad: 2000.
conta2 ingresaEnCapitulo: 'Trabajo' laCantidad: 2000.
conta1 consultaAdeudo: 'Juegos'.
conta1 consultaIngreso: 'Juegos'.
conta1 adeudaEnCapitulo: 'Cubatas' laCantidad: 5000.
conta1 saldo.
ContabilidadMacana totalSaldos.! !
```

```
totalSaldos
"       Abstract: No
        Requires:
        Effect:
        Modifies:
        Answer: devuelve el total de saldos de todas las contabilidades
        Author: JosÃ© Parets
        Version: 1.0
        Date:January 14, 2002"

        ^ TotalSaldos.! !
```


Contabilidad Macana



Tema 3



Lección 3

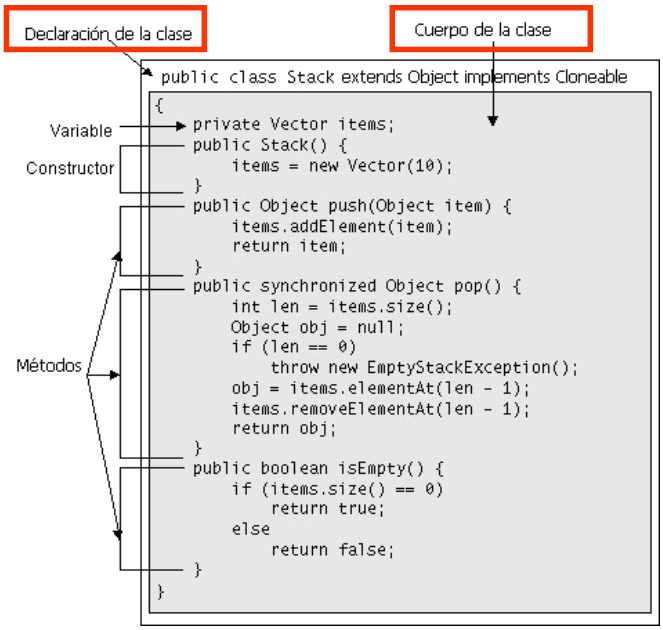
Clases en Java

Puntos a tratar

- Introducción
- Clases: Declaración de una clase
- Interfaces
- Clases: Cuerpo de una clase
  - Representación de clases: variables
  - Implementación de clases: métodos
- Clasificación de las variables: miembros, static, locales
- Creación e inicialización de objetos
- Creación en inicialización de clases
- Algunas clases en Java:
  - Clase Throwable y sus clases: manejo de errores y excepciones
  - Clases envoltorio
  - Clase Applet: los Applets Java
- Concepto de tipo
- Verificación de tipos y reglas de compatibilidad
- Polimorfismo estático (sobrecarga) y dinámico en Java
- Cambio de tipo estático: casting

Introducción

Ejemplo de declaración en implementación de una clase



Declaración de una clase

[public] [abstract | final] class NomClase
[extends NomSuperClase**]**
[implements NomInterface, ...**]**

```
public class Stack extends Object implements Cloneable
{
```

Declaración de una clase

[public] [**abstract** | **final**] **class** NomClase
[**extends** NomSuperClase]
[**implements** NomInterface, ...]

- *public* indica si la clase puede ser usada fuera del paquete al que pertenece

Declaración de una clase

[**public**] [**abstract** | **final**] **class** NomClase
[**extends** NomSuperClase]
[**implements** NomInterface, ...]

- *extends* define la relación de herencia

Declaración de una clase

[public] [**abstract** | **final**] **class** NomClase
  [**extends** NomSuperClase]
  [**implements** NomInterface, ...]

- *abstract* indica que la clase no puede ser instanciada
- *final* establece que la clase no puede tener subclases

Declaración de una clase
Agrupación de clases en paquetes

- Un conjunto de clases puede agruparse en una unidad cohesiva denominada **package**

```
package miPaquete;
public class MiClase {...}
```

- Todas las clases de un paquete deben guardarse en el mismo directorio (nombrado como el paquete)
- Los paquetes pueden organizarse unos dentro de otros

```
package otroPaquete.unSubpaquete
public class TuClase {...}
```

Declaración de una clase
Agrupación de clases en paquetes

- Para usar fuera del paquete una de sus clases públicas podemos:
  a) Especificar la ruta de la clase cada vez
     public class TuClase { ... miPaquete.MiClase ...}
  b) Importar la clase concreta
     import miPaquete.MiClase;
     public class TuClase { ...MiClase ...}
  c) Importar el paquete completo (import miPaquete.*;)

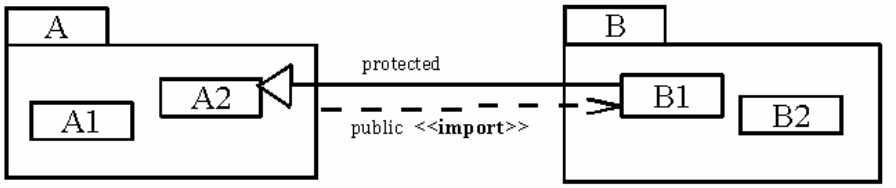
Declaración de una clase
Agrupación de clases en paquetes

- Para que JAVA encuentre el paquete
  a) Se incluye en el CLASSPATH el directorio raíz de la estructura de paquetes
     - Linux: CLASSPATH=/pdo/pract/MI_PROYECTO; export CLASSPATH
     - Windows: set CLASSPATH=C:\pdo\pract\MI_PROYECTO
  b) Se indica el directorio al ejecutar java con la opción –cp o –classpath
     - java -cp /pdo/pract/MI_PROYECTO ClasePrincipal
     - java -cp C:\pdo\pract\MI_PROYECTO ClasePrincipal
- Se puede crear un archivo .jar con uno o varios paquetes comprimidos

Declaración de una clase

Ámbito de una clase

- Una clase puede acceder a todos los elementos declarados package, protected o public en las clases que se encuentran en su mismo paquete
- Una clase sólo puede acceder a los elementos declarados public en las clases incluidas en el paquete que importa (y a las declaradas protected en su superclase)

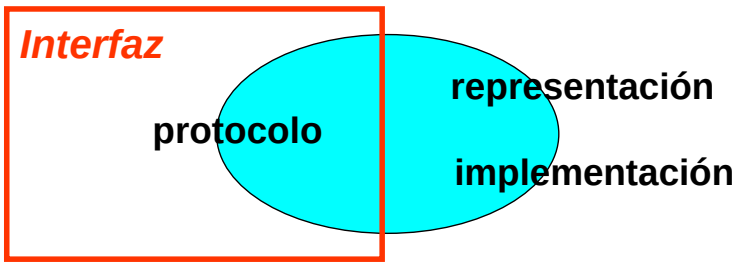


Interfaces

[public] [abstract | final] class NomClase
[extends NomSuperClase]
[implements NomInterface, ...]

- *implements* indica las interfaces que implementa la clase

Interfaces



- Una interfaz no es una clase
  - Una interfaz no puede ser instanciada
- Una interfaz describe un protocolo
  - No declara variables
  - No implementa métodos

Interfaces

- Definición de protocolos
  - Las clases que las usen
    - deben añadir la implementación de todos los métodos
    - Pueden usar varias interfaces
  - Una interfaz puede ampliar el protocolo de otra interfaz
- Diferencia con las clases:
  - Clases: unen protocolo (QUÉ) con implementación (CÓMO)
  - Interfaces: sólo protocolo (QUÉ)
- Ejemplos de interfaces en java.lang:
  - Comparable: define método **compareTo**
  - Observer: define método **update**

Interfaces

[**public**] **interface** NombInterface
[**extends** otraInterface, ...]
{ declaración de constantes y métodos **}**

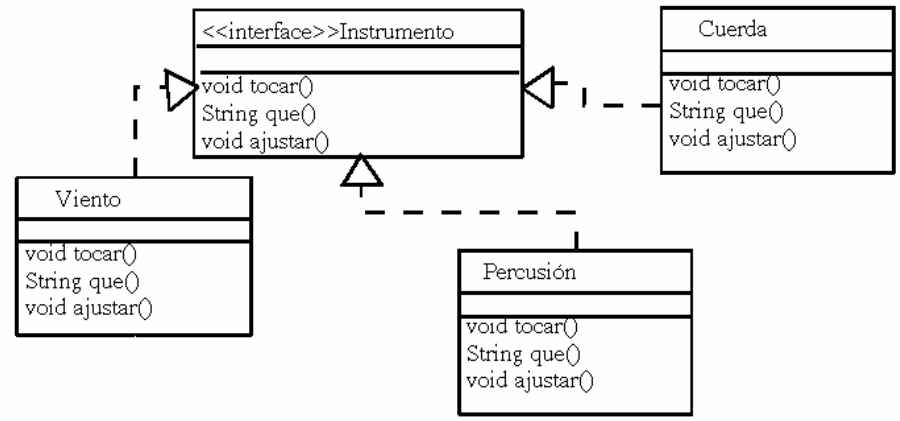
- *public* indica que la interfaz puede ser usada fuera de su paquete
- *extends* permite heredar de otra u otras interfaces

Interfaces

- Ejercicio: escribir una interfaz Animal que defina los métodos **reproducirse**, **desplazarse**, **comer**, **protegerse** y clases **Koala** y **Pingüino** que las implementen.

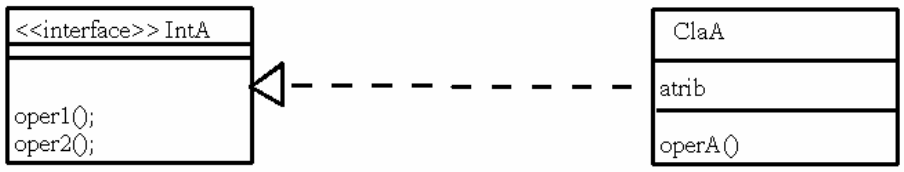
Interfaces

- Utilidad de las interfaces
  - Garantiza un determinado comportamiento en todas las clases que la implementan
    - Permite anticipar comportamiento de clases que no existen



Interfaces

- Una interfaz para ser usada debe ser implementada en una clase
- La clase debe implementar todos los métodos de la interfaz




```
interface IntA{
  public void oper1();
  public void oper2();
}
```

```
class ClaA implements IntA{
    private int atrib;
    public void operA() {...}
    public void oper1() {...}
    public void oper2() {...}
}
```

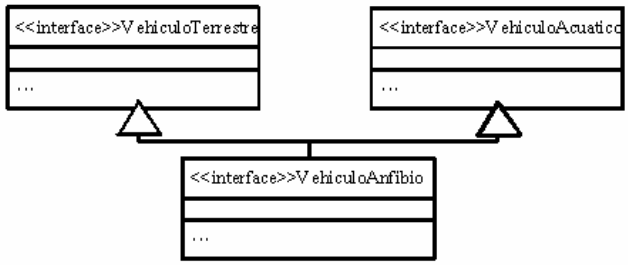
Interfaces

- ¿Por qué añadir interfaces si existen las clases abstractas?
  - Las interfaces permiten simular herencia múltiple???
    - Heredar de varias interfaces
    - Implementar varias interfaces

Interfaces

- ¿Por qué añadir interfaces si existen las clases abstractas?
  - Las interfaces permiten simular herencia múltiple
    - **Heredar de varias interfaces**
    - Implementar varias interfaces

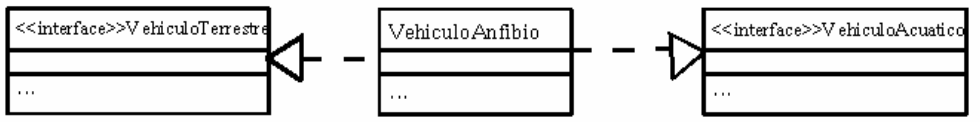
interface VehiculoAnfibio extends VehiculoTerrestre, VehiculoAcuatico { ... }



Interfaces

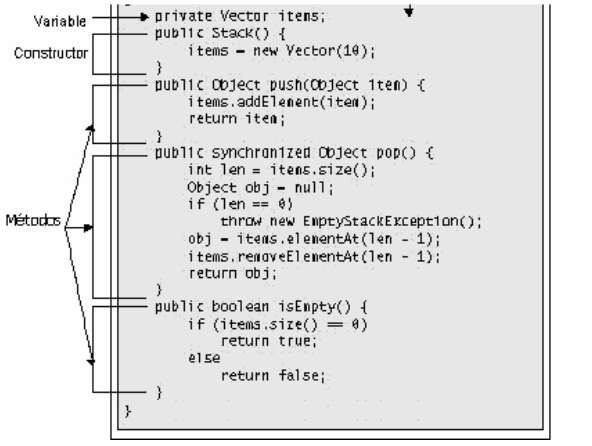
- ¿Por qué añadir interfaces si existen las clases abstractas?
  - Las interfaces permiten simular herencia múltiple
    - Heredar de varias interfaces
    - **Implementar varias interfaces**

class VehiculoAnfibio implements VehiculoTerrestre, VehiculoAcuatico { ... }



Cuerpo de una Clase

- Declaración de variables miembro
- Implementación de métodos miembro



Representación de clases: variables miembro

Declaración de Variables Miembro

[**private** | **package** | **protected** | **public**]
[**static**] [**final**] tipo nomVar [=inicialización];

Representación de clases: variables miembro

Declaración de Variables Miembro

[**private** | **package** |
protected | **public**]
[**static**] [**final**] tipo nomVar
[=inicialización];

- Especificadores de acceso a la variable

		Mismo paquete		Otro paquete	
		Subclase	Otra	Subclase	Otra
-	private				
~	package				
#	protected				
+	public				

Representación de clases: variables miembro

Declaración de Variables Miembro

[private | package |
protected | public]
[static] [final] tipo nomVar
[=inicialización];

- Especificadores de acceso a la variable

		Mismo paquete		Otro paquete	
		Subclase	Otra	Subclase	Otra
-	private				
~	package	x	x		
#	protected	x	x	x	
+	public	x	x	x	x

Representación de clases: variables miembro

Declaración de Variables Miembro

[private | package | protected | public]
[static] [final] tipo nomVar [=inicialización];

- *static* denota una variable de clase

```
class Empleado {
        private int id;
        public static double PorcentajeRetencion = 0,16;
}
```

Representación de clases: variables miembro

Declaración de Variables Miembro

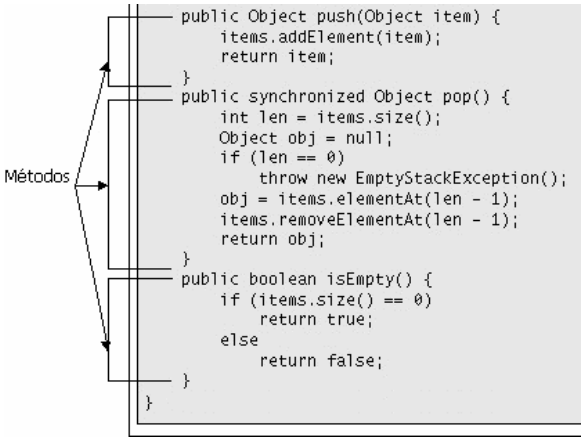
[private | package | protected | public]
[static] [**final**] tipo nomVar [=inicialización];

- *final* indica que es una variable inmutable

Implementación de clases: métodos miembro

Declaración de Métodos Miembro

[private | package | protected | public] [static]
[abstract | final] returnType nomMetodo
([listaParametros]**) [throws** listaExcepciones];



Implementación de clases: métodos miembro

Declaración de Métodos Miembro

[private | package |
protected | public]
[static] [abstract | final]
returnType nomMetodo
([listaParametros])
[throws
listaExcepciones];

- Accesibilidad: Especificadores de ejecución del método

		Mismo paquete		Otro paquete	
		Subclase	Otra	Subclase	Otra
-	private				
~	package	×	×		
#	protected	×	×	×	
+	public	×	×	×	×

Implementación de clases: métodos miembro

Declaración de Métodos Miembro

[private | package | protected | public] [static]
[abstract | final] returnType nomMetodo
([listaParametros]) [throws listaExcepciones];

- *static* denota un método de clase

```
public static void main(String args[]) {
  String p= new String("oros");
  Integer n= new Integer (7);
  Carta cartilla = new Carta(p,n);
  System.out.println(cartilla.palo()); //Escribe “oros”
  p = new String("bastos");
  cartilla.modificaPalo(p);
  System.out.println(cartilla.palo()); //Escribe “bastos”
}
```

Implementación de clases: métodos miembro

Declaración de Métodos Miembro

[private | package | protected | public] [**static**]
[**abstract** | **final**] returnType nomMetodo
([listaParametros]**)** [**throws** listaExcepciones];

- *static* denota un método de clase

```
class Empleado {
        private int id;
        public static double PorcentajeRetencion = 0,16;
        public static double PorcentajeRetencion(){
            return PorcentajeRetencion;
        }
}
double p = Empleado.PorcentajeRetencion();
```


Implementación de clases: métodos miembro

Declaración de Métodos Miembro

[private | package | protected | public] [**static**]
[**abstract** | **final**] returnType nomMetodo
([listaParametros]**)** [**throws** listaExcepciones];

- *abstract* indica que el método no está implementado

```
abstract void update( ) { }; // error compilación
abstract void update( );
```

- *final* indica que el método no puede ser redefinido

Implementación de clases: métodos miembro

Declaración de Métodos Miembro

[**private** | **package** | **protected** | **public**] [**static**] [**abstract** | **final**] returnType nomMetodo ([listaParametros]) [**throws** listaExcepciones];

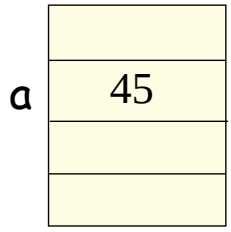
- *throws* especifica el tipo de excepciones que puede lanzar el método

Implementación de clases: métodos miembro

Declaración de Métodos Miembro: Paso por Valor

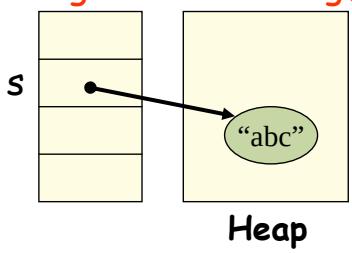
- Java realiza paso por valor para los argumentos y valores devueltos en los métodos

int a = 45;



- Paso por valor: 45
- Paso por referencia: La dirección de memoria donde se almacena la variable a

String s = new String("abc")



- Paso por valor: La dirección de memoria almacenada en la variable s (dirección del objeto)
- Paso por referencia: La dirección de memoria donde se almacena s

Implementación de clases: métodos miembro

Declaración de Métodos Miembro: Paso por Valor

```
class Empleado {
private String dni;
private double sueldo;
private static double PorcentajeRetenciones = 0.5;
public Empleado (String d, double s){
   dni = d;
   sueldo = s;
}
public void dni (String d) { dni = d;}
public String dni() {    return dni; }

public void sueldo (double s) { sueldoi = s;}
public double sueldo() {    return sueldo; }
}
```


Implementación de clases: métodos miembro

Declaración de Métodos Miembro: Paso por Valor

```
class Empleado {
private String dni;
private double sueldo;
private static double PorcentajeRetenciones = 0.5;
public Empleado (String d, double s){
   dni = new String(d);
   sueldo = s;
}
public void dni (String d) { dni = new String(d);}
public String dni() { return new String(dni); }

public void sueldo (double s) { sueldoi = s;}
public double sueldo() {    return sueldo; }
}
```

Creación e Inicialización de Objetos

- Constructores
  - Igual nombre que la clase
  - Operador new
    - Date d = new Date();
    - String s = new Date().toString();
  - Sobrecarga
    - Firma de un método
      - indexOf(int)
      - indexOf(int, int)
      - indexOf(String)
      - indexOf(String, int)
    - Resolución

Creación e Inicialización de Objetos

  - Nombre de los parámetros
    - public Empleado(String **n**) { nombre = n }
    - public Empleado(String **unNombre**) { nombre = unNombre }
    - public Empleado(String **nombre**) { this.nombre = nombre }
  - Constructor predeterminado
    - Inicialización por defecto
  - Llamar a otro constructor
    - this
    - super

Creación e Inicialización de Objetos

- Inicializaciones de campo

```
class Empleado {
   private String nombre = “Juan”
}
```

- Bloques de inicialización

```
class Empleado{
   ...
   private String nombre;
   private String dni;
   { dni=“64543211”; nombre=”Juan”;}
   …
}
```


Creación e Inicialización de Objetos

- Orden en la creación de un objeto
  - Reserva de espacio en la memoria dinámica para almacenar el objeto
  - Inicialización por defecto de las variables de instancia
  - Incialización de campo y bloques de inicialización
  - Ejecución del constructor

Inicialización de variables de clase

```
class Empleado {
private String dni;
private double sueldo;
private static double PorcentajeRetenciones = 0.5;
public Empleado (String d, double s){
   dni = new String (d);
   sueldo = s;
}
public static void modificaPorcentajeRetenciones(double pR){
   PorcentajeRetenciones = pR;
}
…
}
```


Inicialización de variables de clase

```
class Empleado {
.
.
private static double[] PorcentajesRetenciones;

static {
    PorcentajesRetenciones = new PorcentajesRetenciones[10];
    double p=1;
    for (int i=0; i<10; i++) {
          PorcentajesRetenciones[i] = p;
          p = p + 3;
    }
}
```

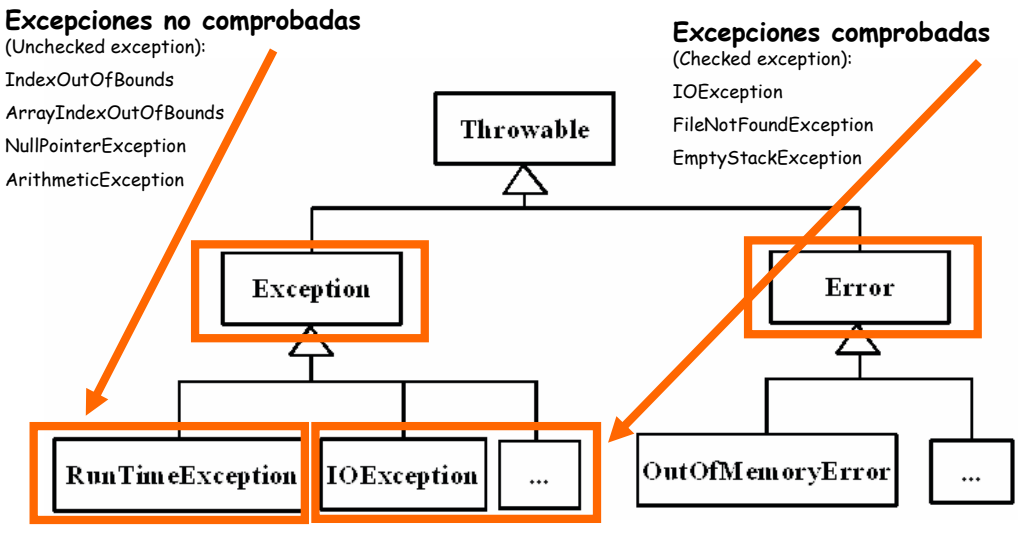
Inicialización de variables de clase

- La primera vez que se referencia una clase, el interprete de Java la busca y carga en memoria
- Se ejecutan una única vez todos sus inicializadores de variables de clase
- Se sigue el orden en la construcción de objetos

```
Empleado e = new Empleado ("Juan", "76873640");
Empleado.modificaPorcentajeRetenciones(0,8);
Empleado e = new Empleado("Ana", "49532349");
```

Se inicializa sólo una vez la variable PorcentajeRetenciones

Clase Throwable y subclases: Manejo de Excepciones



Estructura de clases para el manejo de errores y excepciones

Clase Throwable y subclases: Manejo de Excepciones

- En Java:
  - Excepción: Señal que indica la ocurrencia de una condición excepcional. Clase Exception
  - Error: Fallo de la JVM. Clase Error
  - Manejo de errores y de excepciones: Mecanismo proporcionado para tratar de forma sistemática las excepciones y los errores
  - El programador sólo puede cambiar el tratamiento de las excepciones, no de los errores.
  - Mecanismo para el manejo de excepciones:
    - Definir nuevas excepciones. Serán clases que hereden de Exception o sus subclases
    - Lanzamiento: cuándo producir la situación de excepción
    - Captura y tratamiento: cómo y dónde tratar la excepción
    - Propagación: transferir la responsabilidad de tratamiento de la excepción

Clase Throwable y subclases: Manejo de Excepciones

- **Throwable** es la clase a la que pertenecen los objetos lanzados por la MV de Java o con la sentencia throw
  - Métodos:
    - String getMessage()
    - void printStackTrace()
    - void printStackTrace(PrintStream s)
- Un **Error** es un problema serio que no debe tratarse en el programa (condiciones anormales que no deberían ocurrir nunca)
- Una **Exception** es un evento que ocurre durante la ejecución del programa e interrumpe el flujo normal de instrucciones
- Cuando se llama a un método que puede lanzar una excepción es obligatorio tratar la excepción o propagarla (pasarla hacia arriba), salvo cuando es una excepción no comprobada (las **RunTimeException**)

Clase Throwable y subclases: Manejo de Excepciones: Definición de nuevas clases

- Constructores de la clase Exception y sus subclases:
  - 2 constructores
    - Exception()
    - Exception (String mensaje)
- Se recomienda usar los mismo al declarar nuevas clases Exception

Clase Throwable y subclases: Manejo de Excepciones: Definición de nuevas clases

Ejemplos de definición de subclases de Excepción:

```
Class ExcepciónDeDivisiónPorCero extends Exception {
        ExcepciónDeDivisiónPorCero ( ) {
            super( );
}
ExcepciónDeDivisiónPorCero (String msg) {
            super("ExcepciónDeDivisiónPorCero "+msg);
}
}
```

Clase Throwable y subclases: Manejo de Excepciones:
Definición de nuevas clases

Ejemplos de definición de subclases de Exception:

```
Class ExcepciónDeDirectorNulo extends
      NullPointerException {
       ExcepciónDeDirectorNulo ( ) {
          super( );
}
ExcepciónDeDirectorNulo(String msg) {
      super(msg);
}
}
```

Clase Throwable y subclases: Manejo de Excepciones:

- Objetivo: Controlar de forma explícita las excepciones que se puedan producir en tiempo de ejecución
- Para usar el manejador de excepciones hay que (sólo es necesario hacerlo de forma explícita en excepciones comprobables):
  1. Lanzar la excepción
     1. Detectar la situación de error
     2. Construir un objeto excepción y lanzarlo
  2. Resolver la excepción: Dos formas de hacerlo:
     1. Propagarla (Transferir la responsabilidad del tratamiento): Declarar que el método puede lanzar una excepción
     2. Capturar la excepción y tratarla

Clase Throwable y subclases: Manejo de Excepciones

- Detectar la situación de error
  if(dir==null)
- Construir y lanzar la excepción
  if(dir==null) throw new EmpleadoException("dir es nulo");
- Declarar que el método puede lanzar una excepción
  - Si lanza varias excepciones se pueden declarar cada una de ellas separadas por comas o la superclase de todas

```
public  void nombreDirector() throws EmpleadoException {
    ...
    if(dir==null) throw new NullPointerException("dir es nulo");
    ...
 }
```

Clase Throwable y subclases: Manejo de Excepciones

- Cuando dentro de un método se llama a otro que puede producir una excepción, existen dos posibilidades:
  - Propagar la excepción
  - Capturar y manejar la excepción
- Cuando dentro de un método se lanza una excepción pueden ocurrir dos cosas:
  - Si la excepción se propaga (no se captura), la ejecución del método termina y se pasa el control y la excepción al método desde donde éste se llamó
  - Si la excepción se captura, el control pasa a la sección catch que trata la excepción lanzada (después se pasa a la sección finally si existe)

Clase Throwable y subclases: Manejo de Excepciones

- Manejar la excepción

```
try {
   // llamada al método que podría generar la excepción
}
catch (UnTipoDeExcepcion excepcion1){ gestión del error1 }
catch (OtroTipoDeExcepcion excepcion2){ gestión del error2 }
   ...
finally {// actividades que se ejecutan siempre}
```

- Propagar la excepción
  - En este caso hay que declarar en la cabecera que el método puede lanzar una excepción

```
throws Exception { ....
```


Clase Throwable y subclases: Manejo de Excepciones

- Ventajas de usar excepciones:
  - Separar explícitamente el código normal y el código de gestión de errores
  - Posibilidad de propagar hacia arriba la excepción hasta encontrar el método encargado de gestionarla
  - Agrupar y diferenciar distintos tipos de errores

Clase Throwable y subclases: Manejo de Excepciones:
Propagación de excepciones comprobadas

```
class DniException extends Exception {
    DniException() {
        super("El DNI no es correcto");
    }
}

class Empleado {
    …
    public  void cambiaDNI(String dni) throws DniException {
        ...
        if(dniErroneo)
            throw new DniException();
        ...
    }
}
```


Clase Throwable y subclases: Manejo de Excepciones:
Propagación de excepciones comprobadas

```
class GestionEmpresaModoTexto {
    …
    try {
        nuevoDNI = Entrada.leeString(); //throws IOException
        …
        e.cambiarDNI(nuevoDNI); //throws DniException
        …
    } catch (DniException e) {
        System.out.println("Se ha producide la excepción de DNI" + e);
    } catch (IOException e) {
        System.out.println("Excepción de entrada-salida" + e);
    } catch (Exception e) {
        System.out.println("Otra excepción" + e);
    }
    …
```

Clase Throwable y subclases: Manejo de Excepciones:
Propagación de excepciones comprobadas

```
class GestionEmpresaModoTexto {
    …
    try {
            nuevoDNI = Entrada.leeString(); //throws IOException
            …
            e.cambiarDNI(nuevoDNI); //throws DniException
            …
    } catch (Exception e) {
             System.out.println("Otra excepción" + e);
    } catch (DniException e) {
            System.out.println("Se ha producide la excepción de DNI" + e);
    } catch (IOException e) {
             System.out.println("Excepción de entrada-salida" + e);
    }
```


Clase Throwable y subclases: Manejo de Excepciones:
Propagación de excepciones comprobadas

```
class Empleado {
      …
      public  double calcularSueldoPorPeriodo(int totalPeriodos) throws
          Arithmetic Exception {
              ...
              return sueldo/totalPeriodos();
              ...
      }
}
```

- Las excepciones no comprobadas no necesitan propagarse de manera explícita.
- Si totalPeriodos fuera 0 se propagaría la excepción aun sin poner la cláusula throws ArithmeticException

Clases Envoltorio

- Tipos
  - Tipos primitivos
    - Java: boolean, char, double, int, long, short
      - No se pueden definir nuevos
    - C++: struct, array, record
      - Se pueden definir nuevos tipos estructurados
  - Clases

Clases Envoltorio

- Clase Envoltorio: dotar a los datos primitivos de un envoltorio que permita tratarlos como objeto

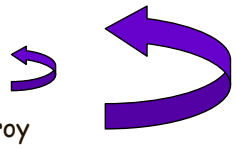
Clase	Tipo primitivo
Byte	byte
Short	short
Integer	int
Long	long
Boolean	boolean
Float	float
Double	double
Character	char

```
public class Integer {
    private int valor;

    Integer(int valor) {
        this.valor = valor;
    }

    int intValue() {
        return valor;
    }
}
```

```
Integer i = new Integer(5);
int x = i.intValue();
```

Clase Applet: los applets JAVA

- Un programa Java puede ser
  - Aplicación
  - Applet
    - Clase que hereda de **java.applet.Applet**
    - Ciclo de vida:
      - init
      - start
      - stop
      - Destroy



    - Otros métodos:
      - paint (Graphics)
      - update()
      - getParameter(String)
    - Inclusión en la página web
      - SANDBOX: Entorno de ejecución restringido

```
<applet code="Clase.class" width="34" height="10"> </applet>
```


Clase Applet: los applets JAVA

- Ejemplo:

```
import java.awt.*;
 import java.applet.Applet;
public class cajaDeColor extends Applet {

        Color boxColor;

        public void init() {
        String s = getParameter("color");
        boxColor = Color.gray;
        if (s != null) {
              if (s.equals("red")) boxColor = Color.red;
              if (s.equals("white")) boxColor = Color.white;
              if (s.equals("blue")) boxColor = Color.blue;
        }
        }

        public void paint(Graphics g) {
        g.setColor (boxColor);
        g.fillRect (0, 0, size().width, size().height);
    }
```

Clase Applet: los applets JAVA

Código fuente de la página:

```
<HTML>
<HEAD>
<Caja de color</TITLE>
</HEAD>
<BODY>
<H1>cajaDeTexto</H1>
<P>
<APPLET CODE="cajaDeColor.class" WIDTH=50 HEIGHT=50>
<PARAM NAME=color VALUE="blue">
</APPLET>
</BODY>
</HTML>
```


Concepto de tipo

- El tipo estático de una variable restringe los objetos que puede referenciar en tiempo de ejecución
  - Tipo de retorno en métodos
- Utilidad
  - Verificación en tiempo de compilación
  - Fiabilidad del programa
- Regla de Compatibilidad
  - Tipos primitivos: Regla de verificación fuerte de tipos

Verificación de tipos

Regla de la compatibilidad fuerte de tipos

"Dos variables **X** e **Y** son asignables si
el tipo declarado para ambas es el mismo"

"Un parámetro formal **X** es sustituible
por el argumento actual **Y** si
su tipo declarado es el mismo"

- Es una compatibilidad por nombre no por estructura
- El tipo estático y dinámico deben coincidir
- No permite polimorfismo

Verificación de tipos

Regla de la compatibilidad fuerte de tipos

```
TYPE   uno = ARRAY [1..10] of  INTEGER,
       dos  = ARRAY [1..10] of INTEGER;

VAR X: uno,
    Y: dos;

X = Y;
```

- Es una compatibilidad por nombre no por estructura
- El tipo estático y dinámico deben coincidir
- No permite polimorfismo

Verificación de tipos

Regla de la compatibilidad de tipos modificada para los L.O.O.

```
class Alumno {
   ...
   ... matricularse(){...}
   ... estudiar(){...}
}
```

```
class AlumnoPDO extends Alumno {
   ...
   ... programarSmalltalk(){...}
}
```

```
Alumno vX = new Alumno();
AlumnoPDO vY =
          new AlumnoPDO();
vX = vY;
vX.matricularse();
vX.estudiar();
```

```
Alumno vX = new Alumno();
AlumnoPDO vY =
          new AlumnoPDO();
vY = vX;
vY.programarSmalltalk();
```


Polimorfismo de mensajes en lenguajes con clases: verificación de tipos

Regla de la compatibilidad de tipos modificada para los L.O.O.

```
class A {
   ...
   ... metodo1(){...}
   ... metodo2(){...}
}
```

```
class B extends A {
   ...
   ... metodo3(){...}
}
```

```
A varX;
B varY = new B();
varX = varY;
varX.metodo1();
varX.metodo2();
```

```
A varX = new A();
B varY;
varY = varX;
varY.metodo3();
```

Verificación de tipos

- Regla modificada para los L.O.O

> "Dada una variable X declarada del tipo de la clase A
> y una variable Y del tipo de la clase B,
> Y puede ser asignada a X si ¿? es subclase de ¿?"
>
> "Lo mismo sucede si X es un parámetro formal
> e Y es el argumento actual

- El tipo estático y dinámico no deben coincidir
- Permite polimorfismo
- Permite X = Y, pero no Y = X

Verificación de tipos

Regla de la compatibilidad de tipos modificada para los L.O.O.

> "Dada una variable X declarada del tipo de la clase A
> y una variable Y del tipo de la clase B,
> Y puede ser asignada a X si B es subclase de A"
>
> "Lo mismo sucede si X es un parámetro formal
> e Y es el argumento actual

- El tipo estático y dinámico no deben coincidir
- Permite polimorfismo
- Permite X = Y, pero no Y = X

Polimorfismo de mensajes en lenguajes con clases: verificación de tipos

Regla de la compatibilidad de tipos modificada para los L.O.O.

- En Java las interfaces pueden usarse en las declaraciones de tipo

"Dada una variable X declarada del tipo de la interfaz A y una variable Y del tipo de la clase B, Y puede ser asignada a X si B implementa la interfaz A"

"Lo mismo sucede si X es un parámetro formal e Y es el argumento actual

Polimorfismo estático y dinámico en Java

Sobrecarga de métodos ≠ Redefinición de métodos o polimorfismo de mensajes

Polimorfismo Estático (Compilación):	Polimorfismo Dinámico (Ejecución):
Los métodos se distinguen por sus argumentos	Los métodos se distinguen por la clase a la que pertenece el objeto

Polimorfismo de mensajes: Ligadura de Métodos

- Polimorfismo de mensajes:
  - Ligadura estática
    - Depende del tipo declarado
  - Ligadura dinámica (Java)
    - Depende de la clase del objeto
    - El polimorfismo de mensajes es dinámico en Java pero puede ser estático: static/final
  - Ligadura mixta
    - Decide el programador
      - Métodos virtuales en C++

Polimorfismo dinámico versus estático: Ejemplo en c++

```
class Base {
public:
 virtual char* ver( ) {
return "base"};
};
class Derivada {
public:
 virtual char* ver( ) {
return "derivada"};
};
```

```
void f(Base b) {b.ver();}
void g (Base & b) {b.ver();}
Derivada d;
f(d); //  "base"
g(d); //  "derivada"
```

Polimorfismo de mensajes: Ligadura de Métodos

```
CLASS Uno
        A () {write ('Soy Uno -- A')}
        B () {write ('Soy Uno  --  B'}
        C () { self A(). self B() }

CLASS Dos: inherit from Uno
        A() { write ('Soy Dos - A') }
        B() { write ('Soy  Dos -B') }
```

- Si hacemos:

```
Uno X;
Dos Y;
X := Y.
X.C()
```

- El resultado es:

Ligadura Dinámica	Ligadura Estática

Polimorfismo de mensajes: Ligadura de Métodos

```
CLASS Uno
        A () {write ('Soy Uno -- A')}
        B () {write ('Soy Uno  --  B'}
        C () { self A(). self B() }

CLASS Dos: inherit from Uno
        A() { write ('Soy Dos - A') }
        B() { write ('Soy  Dos -B') }
```

- Si hacemos:

```
Uno X;
Dos Y;
X := Y.
X.C()
```

- El resultado es:

Ligadura Dinámica	Ligadura Estática
'Soy Dos - A' *'Soy Dos - B'*	*'Soy Uno - A'* *'Soy Uno - B'*

Polimorfismo de mensajes

```
class Persona{
        private String nombre;
        protected String cancion;
        public void canta(){System.out.println("la la la");}
} //End Persona
```

```
class Ladron extends Persona{
        private int ganancias;
        public void canta(){System.out.println("Yo no he sido");}
        public void roba(int n){ ganancias:= ganancias + n; }
} //End Ladron
```

```
class Juez extends Persona {
        public void canta() {System.out.println("Yo no robo -mucho-");}
} //End Juez
```


Polimorfismo de mensajes

```
public class Poli5 {

public static void main(String args[])
        {
        Persona fulanito = new Persona();
        Ladron ladroncito = new Ladron();
        Juez juececito = new Juez();
        fulanito.canta();       //
        fulanito = ladroncito;
        fulanito.canta();       //
        fulanito.roba();        //
        fulanito = juececito;
        fulanito.canta ();      //
        }
} // End Poli5
```

Polimorfismo de mensajes

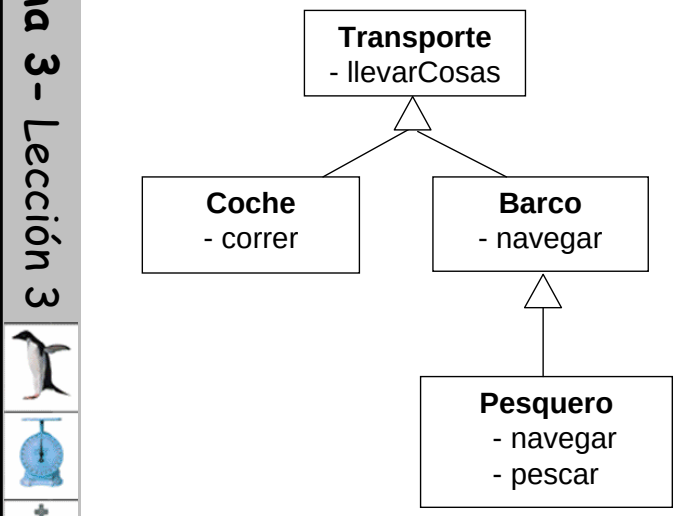
```
public class Poli5 {

public static void main(String args[])
        {
        Persona fulanito = new Persona();
        Ladron ladroncito = new Ladron();
        Juez juececito = new Juez();
        fulanito.canta();       // canta una persona
        fulanito = ladroncito;
        fulanito.canta();       // canta un ladrón
        fulanito.roba();        //error de compilación: fulanito es Persona.
        fulanito = juececito;
        fulanito.canta ();      // canta un juez
        }
} // End Poli5
```


Polimorfismo de mensajes

```
public class Poli4{

public static void main(String args[]) throws java.io.IOException {
    Persona uno;
    System.out.println("Introduzca 1 para el ladron y 2 en otro caso");
            int dato = System.in.read();
            if ((char) dato=='1')
                uno = new Ladron( );
            else
                uno = new Persona( );
    System.out.println("Clase del objeto: "+ (uno.getClass().getName()));
    uno.canta();

} // End Poli4
```

Polimorfismo de mensajes

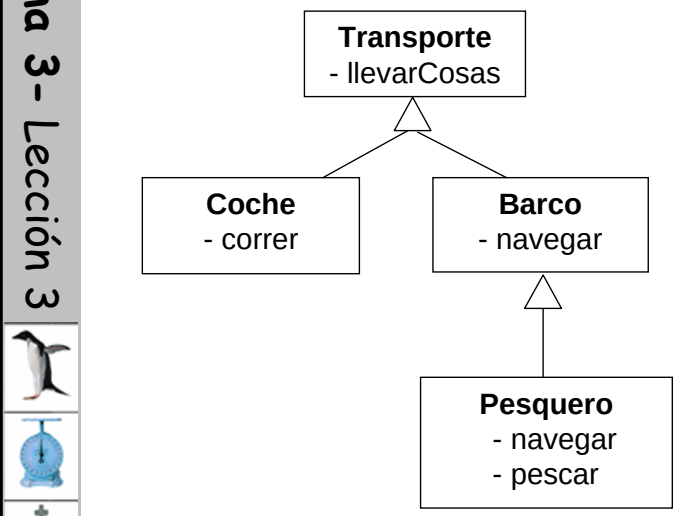
Tema 3- Lección 3




```
class PruebaLigadura {

  public static void main ( String args[]) {

        Coche c1 = new Coche();
        c1. llevarCosas();
        c1.correr();

        Barco b = new Barco();
        b.llevarCosas();
        b.navegar();
        b.correr();
        b=c1;

        Pesquero p = new Pesquero();
        p.navegar();
        p.pescar();
        p.llevarCosas();
        p=b;
        b=p;
        b.navegar();
        b.pescar();

        Vector v= new Vector();
        v.addElement(c1);
        v.addElement(p);
        (v.elementAt(1)).navegar();
        (v.elementAt(1)).llevarCosas();
  }
}
```


Polimorfismo de mensajes

Tema 3- Lección 3




```
class PruebaLigadura {

  public static void main ( String args[]) {

        Coche c1 = new Coche();
        c1. llevarCosas();
        c1.correr();

        Barco b = new Barco();
        b.llevarCosas();
        b.navegar();
        b.correr();
        b=c1;

        Pesquero p = new Pesquero();
        p.navegar();
        p.pescar();
        p.llevarCosas();
        p=b;
        b=p;
        b.navegar();
        b.pescar();

        Vector v= new Vector();
        v.addElement(c1);
        v.addElement(p);
        (v.elementAt(1)).navegar();
        (v.elementAt(1)).llevarCosas();
  }
}
```

Casting y cambio de tipos

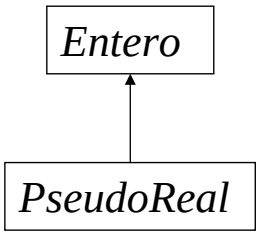
Si no se cumple la regla de compatibilidad modificada de los LOO, podemos forzar su cumplimiento mediante:

cambio explícito del tipo estático:
(down)casting

Casting y cambio de tipo estático

```
class Entero extends Numero {
        int valorEntero;
        …
        public String toString() {
            return Integer.toString(valorEntero);
        }
}

class PseudoReal extends Entero {
        int valorDecimales;
        public String toString() {
            return Integer.toString(valorEntero) + "," +
                    Integer.toString(valorDecimales);
        }
}
```

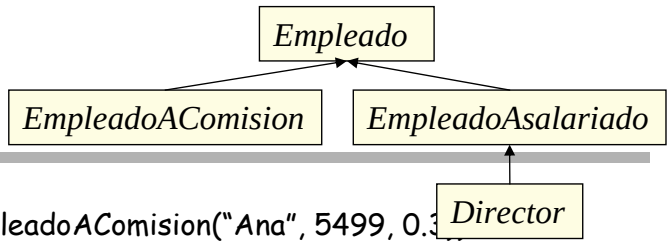
Casting y cambio de tipo estático




```
1. float f = 1.6;
2. int i = (int) f;
3. imprime(i);
4. float f2 = (float) i;
5. imprime(f2);
6. PseudoReal f = new PseudoReal(1.6);
7. Entero i = (Entero) f;
8. imprime(i.toString());
9. PseudoReal f2 = (PseudoReal) i;
10.imprime(f2.toString());
```

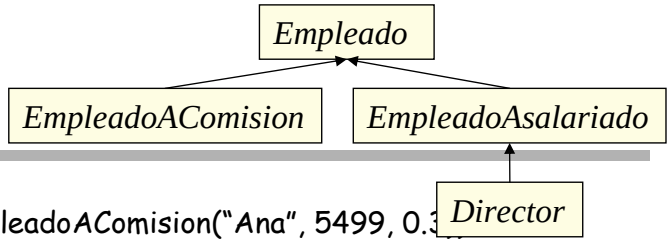
8 y 10: ¿con ligadura dinámica? ¿y con ligadura estática?

Casting y cambio de tipo estático




```
EmpleadoAComision ec = new EmpleadoAComision("Ana", 5499, 0.3
EmpleadoAsalarado ea = new EmpleadoAsalariado("Juan",65426758, 1200));
Director d = new Director("Paqui",76546476,3000));
ArrayList<Empleado> empresa = new ArrayList<Empleado> ();
empresa.add(ec);
empresa.add(ea);
empresa.add(d);
Iterator<Empleado> itEmpr = empresa.Iterator();
while(itEmpr.hasNext()) {
    Empleado e =  itEmpr.next();
    String n = e.dimeNombre();
    float s = e.calculaSueldo();
    if (e instanceof EmpleadoAComision)
            float c =  ((EmpleadoAComision) e).dimeComision();
```

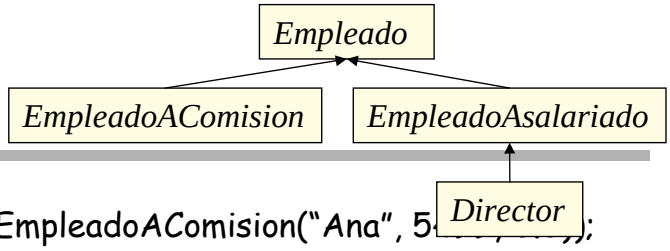
Casting y cambio de tipo estático




```
EmpleadoAComision ec = new EmpleadoAComision("Ana", 5499, 0.3...
EmpleadoAsalarado ea = new EmpleadoAsalariado("Juan",65426758, 1200));
Director d = new Director("Paqui",76546476,3000));

ArrayList<EmpleadoAsalariado> asalariados =
                        new ArrayList<EmpleadoAsalariado> ();
```

- asalariados.add(**ec**);
- asalariados.add((EmpleadoAsalariado) **ec**);
- asalariados.add(**d**);
- asalariados.add((EmpleadoAsalariado) **d**);
- Empleado e = (Empleado) d;
- asalariados.add(**e**);
- asalariados.add((EmpleadoAsalariado) **e**);

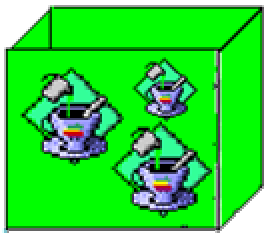
Casting y cambio de tipo estático




```
EmpleadoAComision ec = new EmpleadoAComision("Ana", 5...
EmpleadoAsalarado ea = new EmpleadoAsalariado("Juan",758, 1200));
Director d = new Director("Paqui",7656,3000));
```

1. EmpleadoAsalariado ea2 = **ec**; *Error de compilación*
2. EmpleadoAsalariado ea2 = (EmpleadoAsalariado) **ec**; *Error en ejecución*
3. EmpleadoAsalariado ea3 = **d**; *Correcto*
4. EmpleadoAsalariado ea3 = (EmpleadoAsalariado) **d**; *Correcto*
5. Empleado **e** = (Empleado) **d**; *Correcto*
6. EmpleadoAsalariado ea4 = **e**; *Error de compilación*
7. EmpleadoAsalariado ea4 = (EmpleadoAsalariado) **e**; *Correcto*

Tema 4



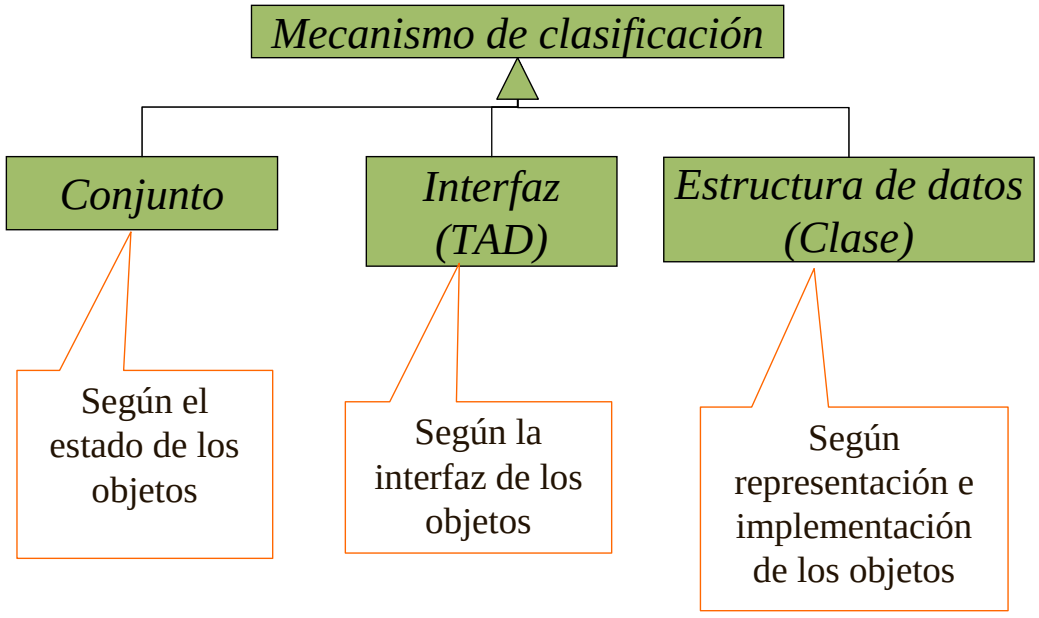
Lección 1

Polimorfismo

Puntos a tratar

- La clasificación en la orientación a objetos
- Concepto de polimorfismo
- El polimorfismo en los LOO
  - Polimorfismo de objetos
  - Polimorfismo sintáctico
- Heurísticas de diseño para aprovechar el polimorfismo

La clasificación de los objetos

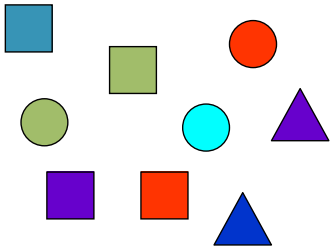


La clasificación de los objetos

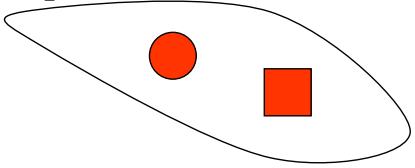
- **Conjuntos**: Clasificación a partir de una propiedad INTENSIONAL denotada por un predicado (consulta). Relativamente frecuente en la BDOO.

Definición INTENSIONAL (consulta):

Objetos rojos



Definición por extensión:



La clasificación de los objetos

- **TAD**: Típica clasificación funcional en base a la interfaz externa de los objetos
- Ejemplos: Lista, NumeroRacional, Pila

Pila: (ejemplo en c)

```
long stack_create(); /* crea una pila */
void stack_push(long stack, void *item); /* introduce elemento en la pila */
void *stack_pop(long stack); /* extrae elemento de la pila */
void stack_delete(long stack); /* borra la pila */
```

La clasificación de los objetos

- **Interfaz**: Tipo abstracto de datos en LOO con herencia (múltiple).
- Ejemplos: Collection, List, Cloneable

List: (ejemplo en Java)

```
Interface List extends Collection
{
boolean add(Object o)     // añadir si no está o si se permiten duplicados
boolean addAll(Collection c) // añadir c a la colección
void clear() // eliminar todos los elementos
Boolean contains(Object o) // está o?
Object get(int index) // devolver objeto en posición index     …
}
```

La clasificación de los objetos

- **Estructura de datos**: Define la representación y la implementación concreta.

Ejemplo en c++:

```
template <typename T>
typedef struct ListaDeEnlazadoSimple {
T element;
ListaDeEnlazadoSimple* next;
~ ListaDeEnlazadoSimple ( ) {if (next!=NULL) delete next; next=NULL;}
ListaDeEnlazadoSimple( ) {next=NULL;}
ListaDeEnlazadoSimple* getNext( ) {return next;}

void addElement (T & element) { next=new ListaDeEnlazadoSimple ();
next.element=element;  }
```


La clasificación de los objetos

- **Clase**:
  - Define la representación e implementación concreta en LOO.
  - Estructura de datos con herencia y polimorfismo

Ejemplo en Java:

```
Class ArrayList implements List
{
Object[ ] elementData;
public Object get(int index)
{return elementData[i];};
…
}
```

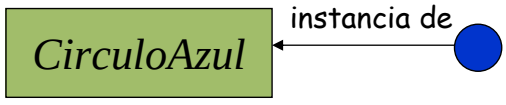
```
Class LinkedList implements List {
ListNode first;
public Object get (int index)
{
ListIterator
listIterator=listIterator(index);
return listIterator.next( );
};
…
}
```

Clasificación teórica de los objetos

- **Deductiva**: La categoría se deduce de la clasificación realizada por el programador.
  - Lenguajes con clases
- **Inductiva**: La categoría se obtiene de las propiedades de los objetos
  - Lenguajes basados en prototipos

Clasificación teórica de los objetos

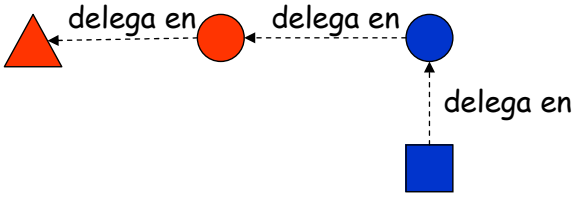
- **Deductiva**:



- **Inductiva**:

Objetos azules:

color.equalsTo(Color.getColor("azul"))

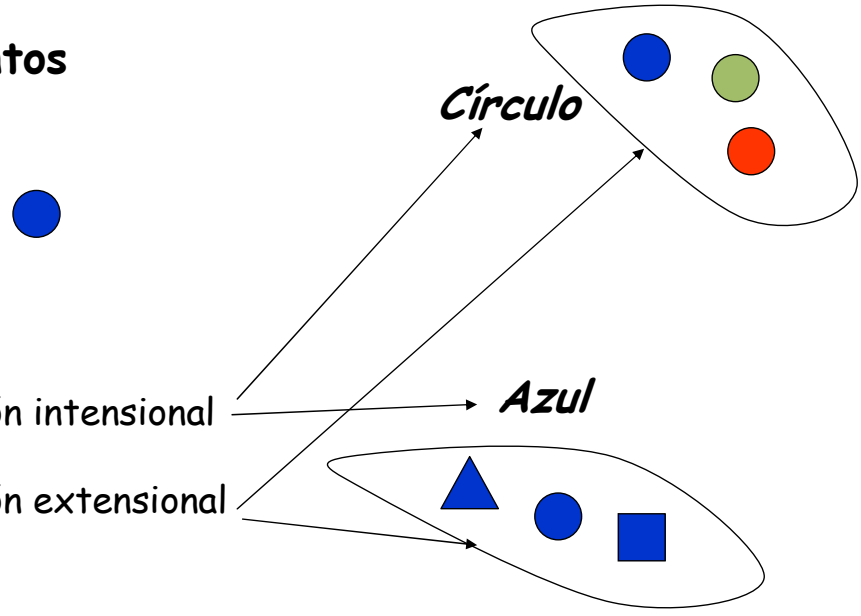


Concepto de polimorfismo

- Concepto polimórfico a su vez (polisemia)
  - Polimorfismo de objetos: Un mismo **objeto** puede pertenecer a **distintas categorías**
  - Polimorfismo sintáctico: Un mismo componente (**identificador**) sintáctico (como un operador, una función, un método, una variable) tiene **diferentes significados** funcionales dependiendo del tipo de los operandos, argumentos u objetos con los que se usan

Polimorfismo de objetos: Ejemplos

Conjuntos



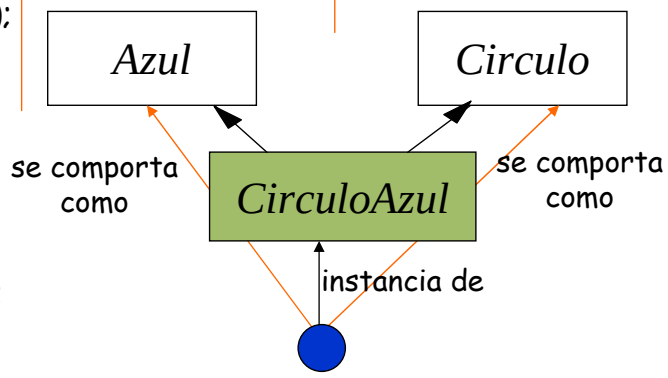
Polimorfismo de objetos: Ejemplos

TAD: Interfaces

```
Interfaz Circulo
{
Punto getCenter( );
float getRadio( );
void setCenter(Punto p);
void setRadio(float r);
};
```

```
Interfaz Azul
{
aclaraAzul(int a);
}
```

```
Class CirculoAzul
implements Circulo,
Azul
{… }
```

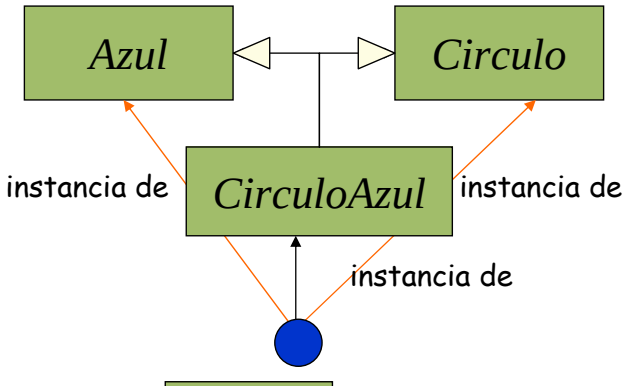



```
CirculoAzul circuloAzul;
```

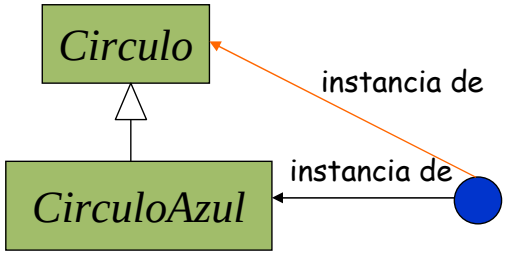

Polimorfismo de objetos: Ejemplos

Clases

Herencia múltiple



Herencia simple



CirculoAzul circuloAzul

Polimorfismo sintáctico

A su vez, varios significados:

1. **Sobrecarga de operadores** (polimorfismo ad hoc de operadores): cambia la implementación según los operandos
   `3 / 5` (división entera) `3.5 / 5` (división real)
2. **Sobrecarga de funciones y métodos** (polimorfismo ad hoc): cambia la implementación según los argumentos
   `print(int)` `print(char)`
   `figura.rotar(Punto p)` `figura.rotar(int x, int y)`

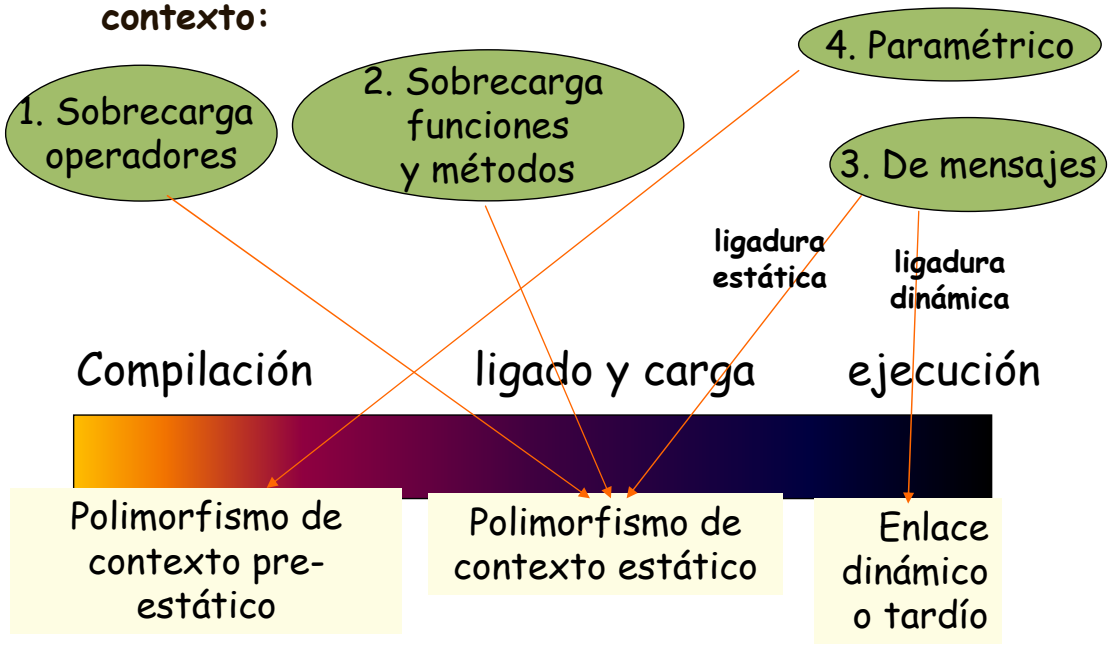
3. **Polimorfismo de mensajes**: cambia la implementación según el receptor del mensaje

```
Racional* racional=new Racional(); ...; racional->print();
Fecha* fecha=      new Fecha(); ...; fecha->print();
```

4. **Polimorfismo paramétrico**: cambia la implementación según el parámetro
   `ArrayList<int>` `ArrayList<Date>`

Polimorfismo sintáctico en los LOO:

- Métodos de determinación del valor semántico según el **contexto**:



1. Sobrecarga de operadores

Ventajas:

- Facilitar la escritura de programas, asociando la misma sintaxis a operaciones que se comportan de forma similar en distintos tipos de datos

Ejemplo:

```
String unEjemplo="un"+"Ejemplo"

int a=10, b=20; a=a+b;
```


1. Sobrecarga de operadores

Ventajas:

- La sobrecarga de operadores **no** ahorra escritura de código de forma independiente, pero
- **sí** ahorra combinada con
  - herencia (polimorfismo de objetos)
  - polimorfismo paramétrico

1. Sobrecarga de operadores

```
Class Empleado {
...
bool equalsTo(Empleado & a)
{if (*this==a) return true; else return false;}
}
virtual bool operator==(Empleado & a)=0;
}
Class EmpleadoAsalariado : Empleado{
...
 virtual bool operator==(Empleado & a){
if ((nombre== a.nombre) && (sueldo==a.sueldo)
return true; else return false;}
}
Class EmpleadoAComision : Empleado{
...
virtual bool operator==(Empleado & a){
if ((nombre== a.nombre) && (comision==a.comision)
    return true; else return false;}            }
```

Ejemplo en c++ con herencia:

```
Empleado * ea, *ea2, *ec, *ec2;

ea=new EmpleadoAsalariado(
"Juan", 1000);

ea2=new EmpleadoAsalariado(
"Juan", 2000);

ea.equalsTo(ea2); // false

ec=new
EmpleadoAComision("Ana", 5 ),
ec2=new EmpleadoAComision(
"Ana", 5);
ec.equalsTo(ec2); // false
```


1. Sobrecarga de operadores

- Ejemplo en C++ con polimorfismo paramétrico:

```
template <class T>
 class Vector
 {
        T* v;
        int sz;
 public:
        Vector ( );
        T& operator [] (int);
        T& elem (int i) {return v[i]; };
        T& sumar (int i, int j) { return v[i] + v[j] }

 };
```

Uso del template:

```
Vector<Integer> v1;
Vector<String> v2;
...
v1.sumar(7,8);
v2.sumar(7,8);
```

2. Sobrecarga de funciones y métodos

- Ventajas
  - Facilitar la escritura de programas, asociando la misma sintaxis a funciones que se comportan de forma similar salvo en la forma
    de recibir los valores de entrada (argumentos de distintos tipos)
  - Ahorra escritura de código convirtiendo los argumentos para reusar después el código que corresponde a la funcionalidad

```
print(int)
figura.desplazar(int grados)
Color.getColor(String nm, int v)
string.substring(int beginIndex)
    endIndex)
```

```
print(char)
figura.desplazar(int x, int y)
Color.getColor(String nm, Color c)
string.substring(int beginIndex, int
```


2. Sobrecarga de funciones y métodos

- Ejemplo en Java de sobrecarga de funciones donde se ahorra código:

```
public String substring(int beginIndex , int endIndex) {
  String s=new String();
  for (int i=beginIndex;i<endIndex)
    s=s+self.charAt(i);
  return s;
}
```

```
public String substring(int beginIndex) {
   return substring(beginIndex, self.length);
}
```

3. Polimorfismo de mensajes

- Ligadura:
  - Estática: El método se distingue por la variable a la que pertenece el objeto (t. compilación)
  - Dinámica: El mismo mensaje se atiende de una forma o de otra según la clase del objeto que lo reciba (t. ejecución)

3. Polimorfismo de mensajes

Ventajas:

- Facilitar la escritura de programas, asociando la misma sintaxis a tareas que se comportan de forma similar en distintos tipos de datos

Ejemplos: size, insert (Java, Smalltalk, C++)
toString (Java)
storeOn (Smalltalk)

3. Polimorfismo de mensajes

Ventajas:

- El polimorfismo de mensajes **no** ahorra escritura de código de forma independiente, pero
- **sí** ahorra combinada con
  - herencia (polimorfismo de objetos)
    - pues permite especificar una tarea implementando un método con la parte común en la clase y delegar la parte específica (polimorfismo de mensajes) en métodos de las subclases
  - polimorfismo paramétrico

3. Polimorfismo de mensajes

```
Object:
storeString
    "Answer a String representation of the receiver
    from which the receiver can be reconstructed."

    | aStream |
    aStream := WriteStream on: (String new: 16).
    self storeOn: aStream.
    ^aStream contents
----------------------------------------------------
Implementors of storeOn: aStream:
    Boolean
    ByteArray
    Association
    Character
    Collection ...
```

Ejemplo en ST con herencia:

```
$a storeString // 'a'
true storeString // 'true'
```

3. Polimorfismo de mensajes

```
Class PrintString {
...
public void print (Object o) {
return String.valueOf(o);
}

Class String {
...
String valueOf (Object o) {
return o.toString();
}
```

Implementors of toString:

- Integer
- Float
- Boolean
- ArrayList
- Punto
- ...

Ejemplo equivalente en Java:

```
String s=new String("hola");
Integer i=new Integer(45);
System.out.print(s);
System.out.print(i);
```

Si no hay herencia (como en tipos básicos):

```
int i=4;
char c='a';
System.out.print(i);
System.out.print(c);
```

¿Qué ocurre?

3. Polimorfismo de mensajes

- Ejemplo en C++ con polimorfismo paramétrico:

```
template <class T>
class Vector
{
        T* v;
        int sz;
public:
        Vector ( );
        T& operator [] (int);
        T& elem (int i) {return v[i]; };
        T& sumar (int i, int j) { return v[i].sumar(v[j]) }

};
```

Uso del template:

```
Vector<Integer> v1;
Vector<String> v2;
...
v1.sumar(7,8);
v2.sumar(7,8);
```

3. Polimorfismo de mensajes

- Ventajas:
  - Homogeneizar métodos entre clases no relacionadas entre sí, para poder hacer referencia a ellos antes de que exista la propia clase.

    Ejemplo en SmallTalk:

    quieroSumar: unObjeto

    "Precondiciones: unObjeto debe responder a asInteger"

    ^self + unObjeto asInteger

3. Polimorfismo de mensajes. Ejemplo:

Ejemplo en Java:

```
class Empleado {
private String nombre;

public Empleado (String n){
     nombre = n;
}

public boolean equals(Object o){
 Empleado e = (Empleado) o;
   return ((e.nombre()). equals(nombre);
}
...
}
```

¿por qué Object?

3. Polimorfismo de mensajes. Ejemplo:

Ejemplo en Java:

```
class EmpleadoAsalariado extends Empleado{
private double sueldo;
public EmpleadoAsalariado (String n, double s){    nombre = n;    sueldo = s; }

public Object clone(){
  return new EmpleadoAsalariado(new String(nombre), sueldo);
}
public boolean equals(Object o){
 EmpleadoAsalariado e = (EmpleadoAsalariado) o;
    return (  ( (e.nombre()). equals(nombre) ) && (e.sueldo()==sueldo) );
}
}
public String toString ( ) {
    return ("Empleado con nombre "+nombre+" y sueldo "+sueldo);
}
```


3. Polimorfismo de mensajes. Ejemplo:

Ejemplo en Java:

```
class EmpleadoAComision extends Empleado{
private double comision;
public EmpleadoAComision (String d, double c){    nombre = n;    comision = c; }

public Object clone(){
  return new EmpleadoAComision(new String(dni), comision);
}
public boolean equals(Object o){
 EmpleadoAComision e = (EmpleadoAComision) o;
    return (  ( (e.nombre()). equals(nombre) ) && (e.comision()==comision) );
}
public String toString ( ) {
    return ("Empleado con nombre "+nombre+" y comisión "+comision);
}
```

3. Polimorfismo de mensajes. Ejemplo:

```
class Prueba {
public static void main(String args[]){
    HashSet s=new HashSet();
    Empleado e = new EmpleadoAsalariado("Juan", 1000);
    Empleado e2= (Empleado) e.clone();
    Empleado e3 = new EmpleadoAComision("Juan", 5);
    Empleado e4 = new EmpleadoAComision("Juan", 8);

    s.add(e);
    s.add(e2);
    s.add(e3);
    s.add(e4);
    System.out.print(s);
}
}
```

Salida:

Empleado con nombre Juan y sueldo 1000

Empleado con nombre Juan y comisión 5

Empleado con nombre Juan y comisión 8

4. Polimorfismo paramétrico

- Necesario el polimorfismo de mensajes cuando se envían mensajes a los parámetros.

Ejemplo en Java:

```
class ArrayList
    {
    public:
      ...
      String toString() {
        String s=String("");
        for (int i=0;i<sz;i++)
          if i<(sz-1)  s=s+v[i].toString()+","
          else s=s+v[i].toString();
        return s;
      };
    };
```

Polimorfismo de mensajes

Uso del template:

```
ArrayList <int> v1=new ArrayList<int>;
...
ArrayList<String> v2=new ArrayList<String>;
...
System.out.print(v1);
System.out.print(v2);
```

- Se realiza sobrecarga de funciones cada vez que cambia el tipo con el que se instancia el genérico

4. Polimorfismo paramétrico

- Necesaria la sobrecarga de operadores cuando se utilizan operaciones del tipo parámetro.

Ejemplo en C++:

```
template <class T>
class Vector {
  T* v;
  int sz;
public:
  vector (int);
  T& operator [] (int);
  T& elem (int i) {return v[i]; };
  T& sumar (int i, int j) { return v[i] + v[j] }

};
```

Uso del template:

```
Vector<String>v2;
v2.sumar(7,8);
Vector<Persona>v3;
v3.sumar(7,8);
```


Polimorfismo sintáctico

Ventajas:

- Evitar la repetición de código
  - Ej. Contenedores genéricos
- En general, el ahorro de código en todos los polimorfismos permite
  - Mayor facilidad de modificación de código
  - Menos oportunidades de introducir errores

Heurística de diseño para aprovechar el polimorfismo

1) Tratar de mantener la semántica abstracta de un identificador sea del tipo o clase que sea.

2) No recurrir a la herencia cuando se puede utilizar con comodidad un tipo genérico.

3) Es difícil evaluar el polimorfismo sin tener en cuenta la herencia.

Polimorfismo dinámico versus estático

4) Factores para decidir el tipo de ligadura en lenguajes que permiten ambos:

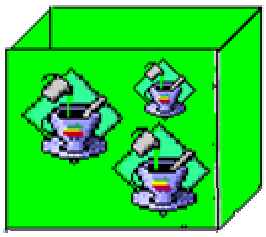
Estático:

↑ Eficiencia, ↓ Heterogeneidad, ↓ Flexibilidad, ↓ Especialización

Dinámico:

↑ Flexibilidad, ↑ Heterogeneidad, ↓ Eficiencia , ↑ Especialización

Tema 4



Lección 2

TAD y clases

Puntos a tratar

- Concepto de TAD
- Definición de un TAD
- Abstracción y conjunto especificado
- Propiedades que debe cumplir la especificación de una abstracción
  - Claridad
  - Adecuación de las operaciones
- Interfaces en Java
  - List
  - Set
  - Cloneable
  - Sequentiable
  - Listener
  - Observer

Concepto de TAD

- Conjunto de abstracciones procedimentales
  - permiten interacciones entre ellas
  - representan todo el comportamiento asociado a una clase de objetos reconocidos en el mundo real (dominio del problema)
- NO es una estructura de datos

Definición de un TAD

- SIGNATURA O CABECERA SINTACTICA: Nombre de la abstracción y cabecera de sus operaciones

Sintaxis

- DESCRIPCION: Descripción de las características de la abstracción y sus operaciones sin realizar ninguna referencia a la implementación
- OPERACIONES: Descritas en forma de abstracciones procedimentales
  - Precondiciones
  - Efecto (postcondición)
  - Parámetros modificados

Semántica

Definición de un TAD

- Ejemplo de signatura y descripción en c:

Pila:

```
long stack_create();
void stack_push(long stack, void *item);
void *stack_pop(long stack);
void stack_delete(long stack);
```

```
// crea una pila
// introduce elemento en  pila
// extrae elemento de la pila
// borra la pila
```


Definición de un TAD

- Ejemplo de signatura y descripción en Java:

```
Interface List extends Collection
{
boolean add(Object o)                  // añadir si no está o si se permiten duplicados
boolean addAll(Collection c)           // añadir c a la colección
void clear()                           // eliminar todos los elementos
Boolean contains(Object o)             // está o?
Object get(int index)                  // devolver objeto en posición index   …
}
```

Abstracción y conjunto especificado

- Una abstracción A puede ser implementada de muchas formas
- Conjunto especificado C: Todas las implementaciones del universo U de implementaciones que satisfacen una abstracción A

java.util

Interface List

All Superinterfaces:

Collection

All Known Implementing Classes:

AbstractList, ArrayList, LinkedList, Vector

Ejemplo en Java:

¿Podemos saber qué clases del universo java.util implementan List?

Propiedades que debe cumplir la especificación de una abstracción

- Claridad

- Adecuación de las operaciones

Propiedades que debe cumplir la especificación de una abstracción

- Claridad: Facilita que las implementaciones satisfagan la especificación.

  Para conseguir claridad, la especificación debe ser:
  - Concisa
  - Estructurada
  - Con ciertas redundancias (identificadas) que permitan al usuario y al implementador comprobar que la ha comprendido.
- Adecuación de las operaciones

Propiedades que debe cumplir la especificación de una abstracción

claridad

Ejemplo en Smalltalk:

contiene: unString

Precondiciones: self size > 0

Efecto: Devuelve el lugar de self en que contiene, por primera vez, unString como substring. Devuelve 0 si no lo contiene, -1 si self size < unString size. Si unString size = 0 devuelve -2.

Por ejemplo, 'abc' contiene: 'ab' -> 1
'abc' contiene: 'abcd' -> -1
'abc' contiene: '' -> -2

Propiedades que debe cumplir la especificación de una abstracción

claridad

- Si la especificación de la abstracción es clara es más fácil que se cumpla la propiedad de Exactitud en el conjunto especificado:
- Es posible determinar C con exactitud si se cumplen dos propiedades:
  - Restrictividad: C no debe incluir implementaciones que no satisfagan A. $Satisf(A) \subseteq C$
  - Generalidad: C no debe omitir ninguna implementación que satisfaga A. $C \subseteq Satisf(A)$

Propiedades que debe cumplir la especificación de una abstracción

Restrictividad

- Consideremos el siguiente ejemplo: Iterador de la clase Bag:
  do: unBloqueConUnArgumento
  Precondiciones:
  Efecto: ejecuta unBloqueConUnArgumento para cada elemento de self
- Esta especificación admite implementaciones que no parecen muy adecuadas:
  1) Se puede cambiar el cotenido de self
  2) El bloque se ejecuta una sola vez para los elementos repetidos
  3) El bloque se ejecuta tras ordenar los elementos de self
- Una especificación más restrictiva sería la siguiente:

do: unBloqueConUnArgumento
Precondiciones: unBloqueConUnArgumento no debe modificar self
Efecto: ejecuta unBloqueConUnArgumento para cada elemento de self, en orden arbitrario. Por cada elemento de self el bloque se ejecuta tantas veces como esté en self.

Propiedades que debe cumplir la especificación de una abstracción

Generalidad

- Consideremos un método de **búsqueda en una Array**:

- La siguiente especificación es muy restrictiva porque supone una implementación concreta:

buscar: unObject
Precondiciones:
Efecto: examina secuencialmente self hasta encontrar unObject devolviendo el índice en que lo encuentra. 0 si no lo encuentra.

CÓMO

- Esta especificación es más general:

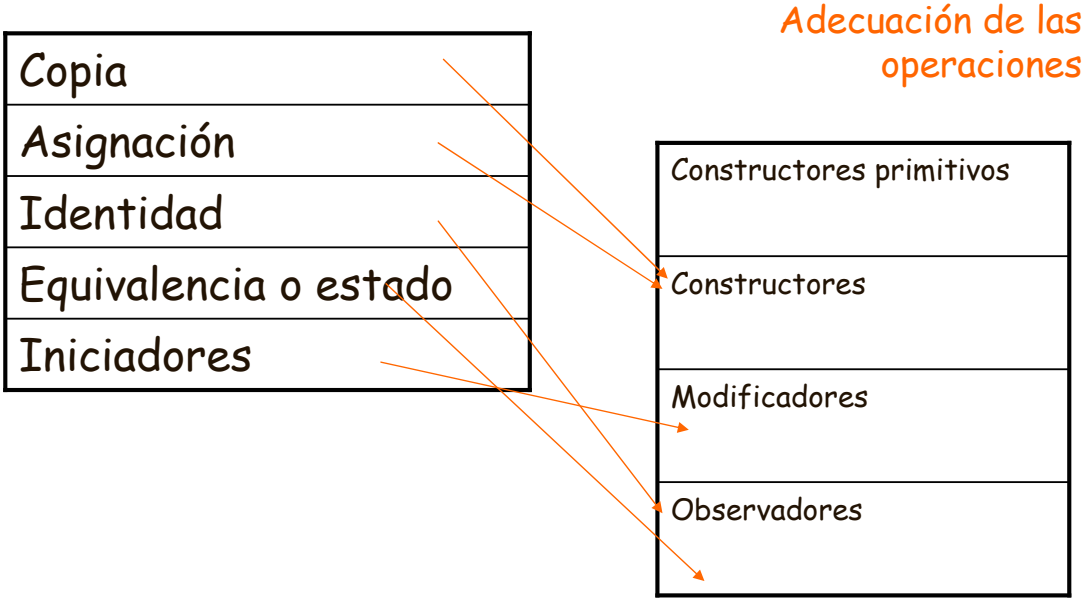
buscar: unObject
Precondiciones:
Efecto: busca unObject en self devolviendo el primer índice en que lo encuentra. 0 si no lo encuentra.

Propiedades que debe cumplir la especificación de una abstracción

- Claridad
- Adecuación: Heurística para la adecuación de las operaciones
  1. Debe poseer operaciones de cada un de los tipos siguientes:
     - CONSTRUCTORES PRIMITIVOS: crean objetos del tipo sin necesitar objetos del tipo.
     - CONSTRUCTORES: toman objetos del tipo como entrada y crean otros objetos del tipo.
     - MODIFICADORES: modifican objetos del tipo. Los tipos inmutables carecen de modificadores.
     - OBSERVADORES: a partir de objetos del tipo devuelven objetos de otros tipos.

Propiedades que debe cumplir la especificación de una abstracción

Adecuación de las operaciones



Propiedades que debe cumplir la especificación de una abstracción

Adecuación de las operaciones

¿Es siempre la asignación una operación de construcción?

	Pila	Montículo
Pascal		-
Smalltalk, Eiffel	-	
C++	tipos primitivos y clases	clases
Java	tipos primitivos	clases

Propiedades que debe cumplir la especificación de una abstracción

Adecuación de las operaciones

¿Es siempre la asignación una operación de construcción?

Ejemplo en c++:

```
Pair* a=newPair(1,2), *b;
b=a;
```

```
Pair a=Pair(1,2),b;
b=a;
```

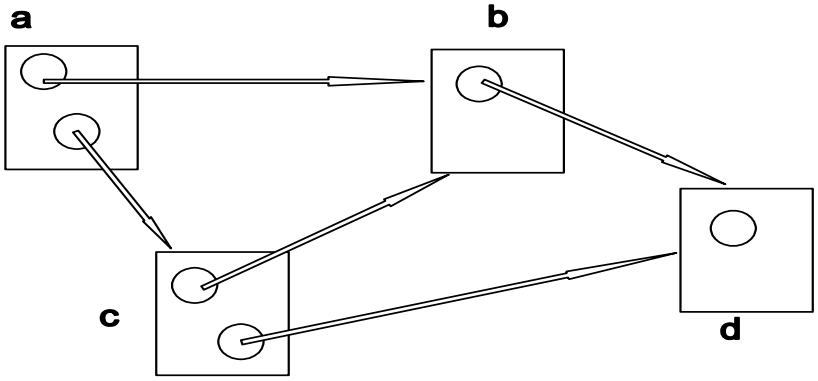
Montículo

Pila

Propiedades que debe cumplir la especificación de una abstracción

Adecuación de las operaciones

El constructor de copia y la asignación como operador de construcción en las clases requiere una implementación específica para cada clase que a veces puede ser muy compleja.



Propiedades que debe cumplir la especificación de una abstracción

Adecuación de las operaciones

2. El acceso a elementos de la representación puede reducir considerablemente el número de operaciones necesarias, pero aumentará el acoplamiento entre clases.

Ley de Demeter: "Habla sólo con tus amigos"
En un método **m** de una clase **o** sólo deberían invocarse métodos de:
-el propio objeto **o**
-los parámetros que recibe el propio método **m**
-cualquier objeto que *instancie el propio método* **m**
-cualquier atributo de **o**

Por definición, la abstracción no accede a la representación.

Propiedades que debe cumplir la especificación de una abstracción

Adecuación de las operaciones

3. El exceso de operaciones puede hacer menos comprensible la abstracción y más difícil la implementación y el mantenimiento.
4. Algunas operaciones adicionales podrán implementarse a partir de operaciones más básicas.
5. Usar una semántica consistente. Ejem: constructor de copia: clone en Java

Propiedades que debe cumplir la especificación de una abstracción

Adecuación de las operaciones

6. La operación de EQUIVALENCIA o comparación del ESTADO debe existir en todos los TADs.

C++	Java	Smalltalk
==	equals	=

Propiedades que debe cumplir la especificación de una abstracción

- Claridad
- Adecuación: Criterios para la adecuación de las precondiciones
  - **Precondición RESTRICTIVA**: más eficiente y limita su efecto, haciendo la implementación más sencilla. En general también son menos seguros.
  - La **LAXITUD en las restricciones** implica procedimientos más complejos, con efecto más amplio, mayor complejidad de implementación y una mayor seguridad.

?

Propiedades que debe cumplir la especificación de una abstracción

- Adecuación de las precondiciones: Ejemplo

Métodos de la clase Collection:

mezcla: unaCollectionDeInteger
Precondición: self y unaCollection deben ser colecciones de números ordenadas mediante la operación >=
Efecto: mezcla ordenadamente self y unaCollectionDeInteger
Modifica:
Devuelve: la mezcla de ambas colecciones

mezcla: unaCollectionDeInteger
Precondición: self y unaCollection deben ser colecciones de números.
Efecto: mezcla ordenadamente self y unaCollectionDeInteger
Modifica:
Devuelve: la mezcla de ambas colecciones

Interfaces en Java

- Ejemplos:
  - Comparable: java.lang
  - Collection: java.lang
    - List: java.util
  - Serializable: java.io (java Beans)
  - Cloneable: java.lang
  - Observer: java.util
  - EventListener: java.util
    - ActionListener: java.awt.event

Interfaces en Java

comparable

java.lang
Interface Comparable

All Known Implementing Classes:

BigDecimal, BigInteger, Byte, ByteBuffer, Character, CharBuffer, Charset, CollationKey, Date, Double, DoubleBuffer, File, Float, FloatBuffer, IntBuffer, Integer, Long, LongBuffer, ObjectStreamField, Short, ShortBuffer, String, URI

public interface **Comparable**

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

The natural ordering for a class C is said to be *consistent with equals* if and only if (e1.compareTo((Object)e2) == 0) has the same boolean value as e1.equals((Object)e2) for every e1 and e2 of class C. Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.

For example, if one adds two keys a and b such that (!a.equals((Object)b) && a.compareTo((Object)b) == 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement comparable have natural orderings that are consistent with equals. One exception is java.math.BigDecimal, whose natural ordering equates BigDecimal objects with equal values and different precisions (such as 4.0 and 4.00).

For the mathematically inclined, the *relation* that defines the natural ordering on a given class C is:

{(x, y) such that x.compareTo((Object)y) <= 0}. The *quotient* for this total order is: {(x, y) such that x.compareTo((Object)y) == 0}. It follows immediately from the contract for compareTo that the quotient is an *equivalence relation* on C, and that the natural ordering is a *total order* on C. When we say that a class's natural ordering is *consistent with equals*, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's equals(Object) method: {(x, y) such that x.equals((Object)y)}. This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

Comparator, Collections.sort(java.util.List), Arrays.sort(Object[]), SortedSet, SortedMap, TreeSet, TreeMap

Method Summary int **compareTo**(Object o)
Compares this object with the specified object for order.

Interfaces en Java

comparable

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to athe class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

- The natural ordering for a class C is said to be *consistent with equals* if and only if (e1.compareTo((Object)e2) == 0) has the same boolean value as e1.equals((Object)e2) for every e1 and e2 of class C. Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.
- It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.
- For example, if one adds two keys a and b such that (!a.equals((Object)b) && a.compareTo((Object)b) == 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.
- Virtually all Java core classes that implement comparable have natural orderings that are consistent with equals. One exception is java.math.BigDecimal, whose natural ordering equates BigDecimal objects with equal values and different precisions (such as 4.0 and 4.00).
- For the mathematically inclined, the *relation* that defines the natural ordering on a given class C is:
- {(x, y) such that x.compareTo((Object)y) <= 0}. The *quotient* for this total order is: {(x, y) such that x.compareTo((Object)y) == 0}. It follows immediately from the contract for compareTo that the quotient is an *equivalence relation* on C, and that the natural ordering is a *total order* on C. When we say that a class's natural ordering is *consistent with equals*, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's equals(Object) method: {(x, y) such that x.equals((Object)y)}. This interface is a member of the Java Collections Framework.

Interfaces en Java

comparable

Method Detail
compareTo

public int **compareTo**(Object o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. In the foregoing description, the notation sgn(*expression*) designates the mathematical *signum* function, which is defined to return one of -1, 0, or 1 according to whether the value of *expression* is negative, zero or positive. The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)

The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

Finally, the implementer must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

It is strongly recommended, but *not* strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Parameters:

o - the Object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

ClassCastException - if the specified object's type prevents it from being compared to this Object.

Interfaces en Java

comparable

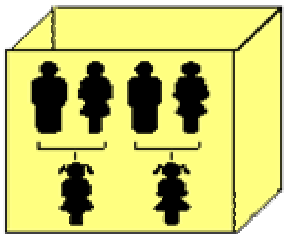
```
Class Empleado implements Comparable {
int salario;
String dni;
public Empleado (String dni, int salario) {
this.salario=salario;
dni=new String(dni);
}
public String toString() {return "dni:"+dni+", salario"+salario;}
public int getSalario() {return salario;}
public int compareTo(Object o) {
 return signum(salario-(Empleado)o.getSalario());}
}
ArrayList<Empleado> listaEmpleados=new ArrayList<Empleado>();
listaEmpleado.add(new Empleado("23344445", 1000);
listaEmpleado.add(new Empleado("23354445", 1500);
Collections.sort(listaEmpleado);
System.out.print(listaEmpleado);
```

Output:

dni: 23354445, salario: 1500

dni: 23344445, salario: 1000

Tema 5

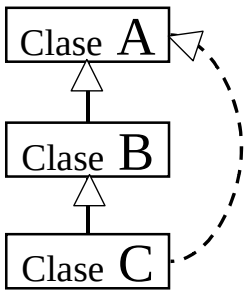


Lección 1
Concepto de herencia



¿Qué es la Herencia?

- Propiedad de que las instancias de una clase hija tengan acceso a la estructura de datos* y al comportamiento definidos en la clase padre
  - * En Java las variables *private* no pueden ser accedidas directamente en la subclase
- La herencia es transitiva



Nomenclatura

- Superclase, ancestro o clase padre
- Subclase, descendiente o clase hija
- La herencia de variables (atributos o datos miembro) es siempre ampliativa y no se pueden redefinir.
- La herencia de métodos (funciones miembro) es ampliativa y los métodos se pueden redefinir (sobreescritura)

Composición vs. Construcción

- Problema: Dada la clase *Numero,* definir la clase *Numero2* con la misma funcionalidad más una función de conversión a romano

```
class Numero {
  private int num;
  public Numero (int n) {num = n;}
  public valorInt () {return num;}
  public suma(int num2) { return num+num2;}
  public resta(int num2) { return num-num2;}
  ...
}
```

Composición vs. Construcción

- Solución por Construcción (Heredando de la clase):

```
class Numero2 extends Numero{
  public String romano(){ … uso this.valorInt() …}
}
```

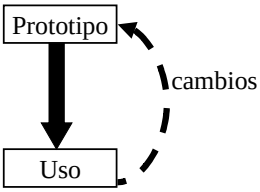
Composición vs. Construcción

- Solución por Composición (Usando la clase):

```
class Numero2 {
  private Numero num2;
  public Numero2 (int n) {num2 = new Numero(n);}
  public valorInt () {return num2.valorInt();}
  public suma(int n) { return num2.suma(n);}
  public resta(int n) { return num2.resta(n);}
  …
  public String romano () { … uso num2.valorInt() …}
}
```

Beneficios de la Herencia

- Reusabilidad
- Fiabilidad
- Compartición
- Consistencia
- Biblioteca de componentes software
- Ocultación de información
- Modelado de prototipos



Costes de la Herencia

- Velocidad de ejecución
  - Código general vs. código especializado
- Tamaño del programa ?

Heurísticas para la Herencia

- ¿Cómo crear subclases?
  - Especialización
  - Generalización
  - Especificación
  - Extensión
  - Limitación
  - Variación
  - Combinación

Heurísticas para la Herencia

- ¿Cómo crear subclases?
  - Especialización

Si dados dos conceptos A y B,
tiene sentido el enunciado *B **es un** A*,
entonces es probable que B sea una subclase de A

Si dados dos conceptos A y B,
tiene sentido el enunciado *B **tiene un** A*,
entonces es probable que en la clase B exista un atributo de tipo A

Heurísticas para la Herencia

- ¿Cómo crear subclases?
  - Especialización

Perro **es un** Mamífero ➔
class Perro extends Mamifero { ...}
Mamifero subclass: #Perro ...

Automóvil **tiene un** Motor ➔
class Automovil { private Motor unMotor; ...}
Object subclass: #Automovil variableInstanceNames:'unMotor ...' ...

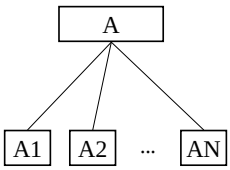
Heurísticas para la Herencia

- ¿Cómo crear subclases?
  - Generalización
    - Es lo opuesto a la Especialización
    - La subclase es más general
      - Se redefine al menos un método
    - Sólo si la superclase no puede ser modificada

VentanaNegra **es una** VentanaAColor (donde color es negro) ➔
class VentanaAColor extends VentanaNegra {
... public void dibuja(Color unColor) {...} ...}

Heurísticas para la Herencia

- ¿Cómo crear subclases?
  - Especificación
    - Interfaces de Java



Usar la herencia para garantizar que
un conjunto de clases tiene un **protocolo común**

Heurísticas para la Herencia

- ¿Cómo crear subclases?
  - Extensión
    - No es obligatorio redefinir ningún método

Se **amplía la subclase** añadiendo métodos y variables

Heurísticas para la Herencia

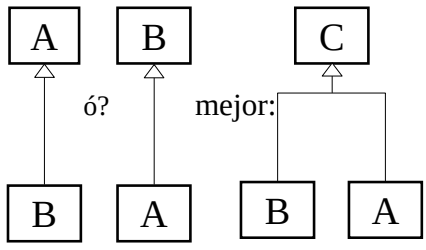
- ¿Cómo crear subclases?
  - Limitación
    - Es lo opuesto a la Extensión
    - Es obligatorio redefinir al menos un método dando un mensaje de error
    - No debe utilizarse

Se **reduce el comportamiento** de la subclase

Heurísticas para la Herencia

- ¿Cómo crear subclases?
  - Variación

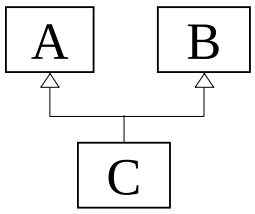
Agrupar el **código común**
de dos clases sin relación de herencia



Heurísticas para la Herencia

- ¿Cómo crear subclases?
  - Combinación
    - Herencia múltiple

Combinar en una clase características de **dos o más superclases**



Modos de la herencia

- Según el número de jerarquías
  - Monojerárquica: una sóla jerarquía de herencia (JAVA, ST, Eiffel)
  - Plurijerárquica: cuando se define una nueva clase puede formar una nueva jerarquía (C++)
- Según el número de ancestros (superclases)
  - Simple: cada clase tiene una superclase (JAVA, ST)
  - Múltiple: una clase puede tener varias superclases (Eiffel, C++)

Herencia simple

- Una subclase hereda TODAS las variables y métodos de su superclase.
- Existe un ancestro común que se suele denominar Object.

Herencia múltiple

- La subclase hereda todas las variables y métodos de sus superclases
- Permite modelar problemas en los que un objeto tiene propiedades según criterios diferentes:
  - Vehículos clasificados según el permiso de conducir
  - Vehículos clasificados según el medio (anfibios, terrestres, aéreos)
- Las estructuras de herencia múltiple son más flexibles, pero más difíciles de manejar.
- Presenta problemas de implementación

Herencia Múltiple en los Lenguajes

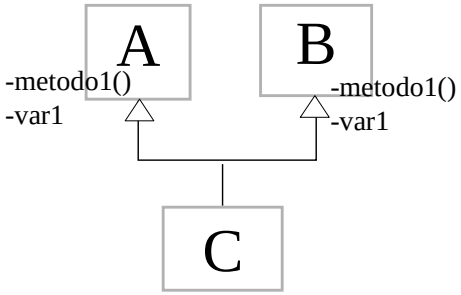
- Smalltalk : Herencia simple
- Java: Herencia simple
  - Las interfaces permiten simular la herencia múltiple
- C++: Herencia múltiple

Problemas de la Herencia Múltiple

- **Ambigüedad en los nombres**

¿Elijo uno o me quedo con los dos?

Renombrar y Redefinir



Problemas de la Herencia Múltiple

```
class A {
    public:

    void virtual mostrar (int i)
            {.....}
    void virtual ver (int p)
            {....}
}
```

Renombrar:

```
class B {
    public:

    void virtual mostrar (float f)
            {.....}
    void virtual ver (int n)
            {....}
}
```

```
class C: public B, public A {
    public:
    void virtual verA (int p)
            {A::ver(p);}

    void virtual verB (int p)
            {B::ver(p);}
}
```

Problemas de la Herencia Múltiple

```
class Mamifero {

public:
        int tiempo_incubacion;
        char* habitat;
        void comer(char*);
}
```

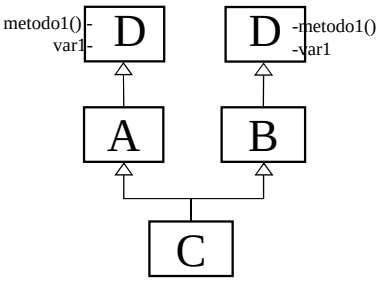
Redefinir:

```
class Oviparo {

public:

        int tamaño_huevos;
        int habitat;
        void comer(char*);
}
```

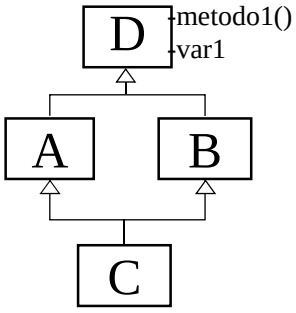
```
class Ornitorrinco: public Mamifero, public Oviparo {

void comer (char* c) {

        // this.comer(c) es ambigüo y no puede escribirse aquí

        this.Mamifero::comer(c)      //deshace la ambigüedad
}
```

Problemas de la Herencia Múltiple

- **Herencia de ancestros comunes: número de copias de variables (C++)**



¿Dos copias o una?



Class A: public D
Class B: public D
Class C: public A, public B

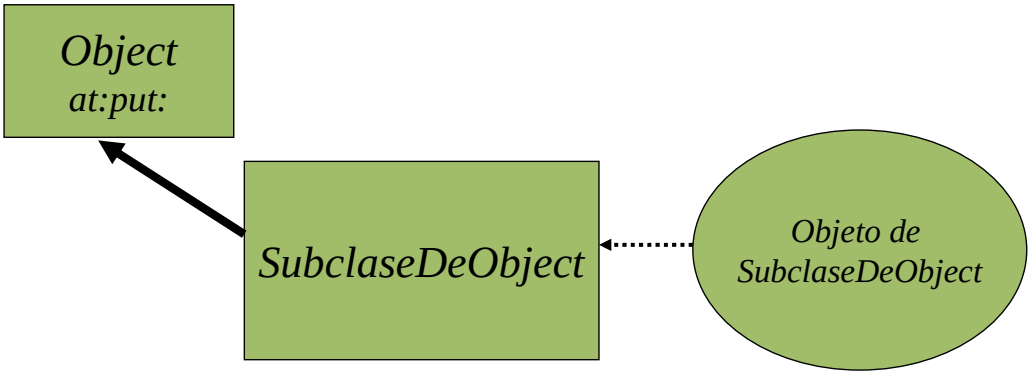
Class A: virtual public D
Class B: virtual public D
Class C: public A, public B

Problemas de la Herencia Múltiple

- **Herencia de ancestros comunes: ejecución de métodos**
  - ¿Qué camino se sigue para ejecutar un método que se ha redefinido en varias subclases?
  - Soluciones:
    - No se permiten ancestros comunes (Eiffel)
    - Se da prioridad a uno de los caminos (Lisp OO)
    - Se deben redefinir los métodos de cada camino como para la colisión de nombres (C++)

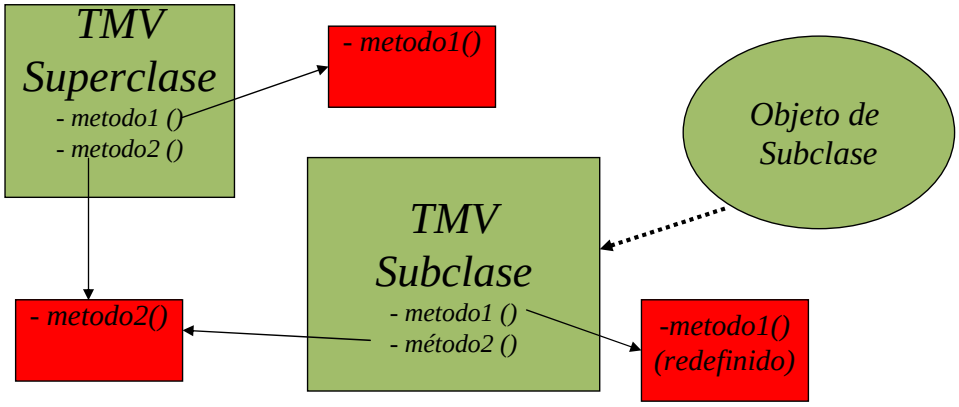
Implementación de la herencia: lenguajes interpretados

- Cada objeto conoce su clase
- Cada clase conoce su superclase
- Se necesita la estructura de clases en memoria
- Se pueden introducir nuevos métodos sin recompilar



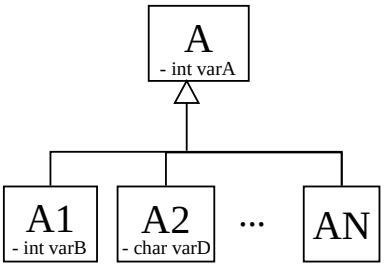
Implementación de la herencia: lenguajes compilados

- Cada objeto tiene una referencia a la Tabla de Métodos Virtuales de su clase
- En la TMV se repiten las direcciones de los métodos no redefinidos. No es necesario buscar en la superclase.
- Para añadir nuevos métodos hay que recompilar



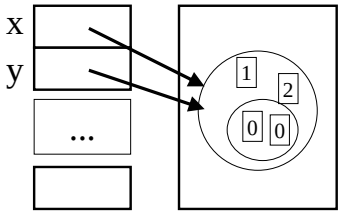
Asignación de Memoria

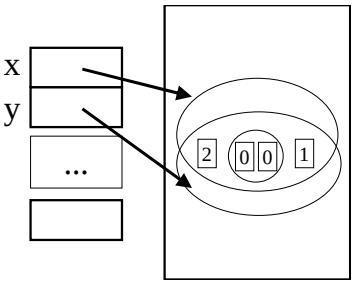
- Se reserva en pila memoria para la clase declarada
  - Se pierde información
    - A var; var = new A2(34, 'c')
- Se reserva en pila memoria para la clase y cualquiera de sus subclases
  - No se conocen todas las subclases hasta tiempo de ejecución
- Se reserva en pila memoria para un puntero y en tiempo de ejecución se reserva en el montículo memoria para la clase referenciada



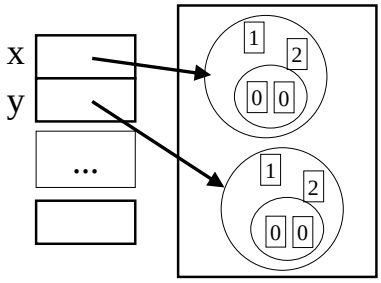
Asignación de Variables

- Objetos en el Heap
  - En Smalltalk (y :=x.)
  - En Java (y = x;)
- Copia en Smalltalk





COPY: Las variables de instancia se comparten con el original



DEEPCOPY: Las variables de instancia son copias del original

Asignación de Variables

- Ejemplo Smalltalk:

```
Object subclass:#Rectangulo instancevariablleNames: 'alto ancho
    centro' ...
unPunto := Punto cordx:0 cordy:0.
x := Rectangulo ancho:1 alto:2 centro:unPunto.
```

- Ejemplo Java:

```
class Rectangulo {
        private int alto;
        private int ancho;
        private Punto centro;
        ...
}
Punto unPunto := new Punto(0,0);
Rectangulo x:= new Rectangulo(1, 2, unPunto);
```


Asignación de Variables

- Copia en Java
  - Método clone
    - Heredado de Object
    - Hay que redefinirlo
      - También el método equals
  - La asignación copia la referencia en los objetos (heap), pero el valor en los valores primitivos (stack)

```
class Rectangulo {
        ...
        public Object clone() {
                ancho2 = ancho;
                alto2 = alto;
                centro2 = centro.clone();
                return new Rectangulo(ancho2, alto2, centro2);
        }
}
```

Comparación de Objetos

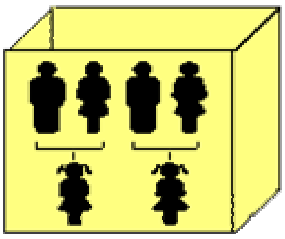
- En Smalltalk
  - Comparar identidad (==)
  - Comparar estado (=)
- En Java
  - Comparar identidad de objetos y valor de tipos primitivos (==)
  - Comparar estado de objetos (equals)

```
class Rectangulo {
        ...
        public boolean equals(Object o){
            Rectangulo r = (Rectangulo) o;
            return (  ( (r.centro()). equals(centro) ) && (r.alto()==alto) &&
            (r.ancho()==ancho));
        }
}
```

Comparación de Objetos

- Responde V o F:
  - Samlltalk:
    - y:=x. y==x.
    - y := x copy. y==x.
    - y := x copy. y centro == x centro.
    - y := x deepCopy. y centro ==x centro.
  - Java:
    - y = x; y==x;
    - y=x; y.equals(x);
    - y=x.clone(); y==x;
    - y=x.clone(); y.ancho() == x.ancho()
    - y=x.clone(); y.centro() == x.centro()
    - y=x.clone(); y.equals(x);

Tema 5



Lección 2

La herencia en Smalltalk

Puntos a tratar

Tema 5- Lección 2

- Herencia de métodos y variables de instancia
  - Jerarquía de clases
- Herencia de métodos de clase
  - Concepto de metaclase
  - Herencia de metaclases
  - Instanciación de metaclass

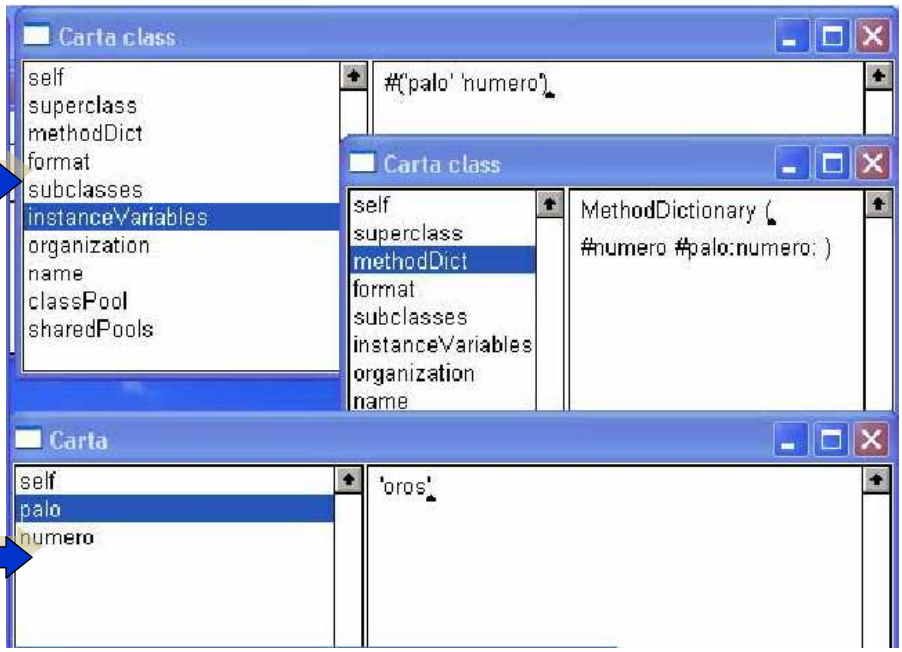
Herencia de variables y métodos de instancia

- Los nombres de las variables de instancia y los métodos de instancia se almacenan en la clase
  - Podemos verlos inspeccionando la clase (inspect)
    - methodDict
    - instanceVariables
  - Podemos añadir nuevas variables (addInstVarName:)
  - Podemos borrar métodos (removeSelector:)
  - etc.

Herencia de variables y métodos de instancia

Carta inspect

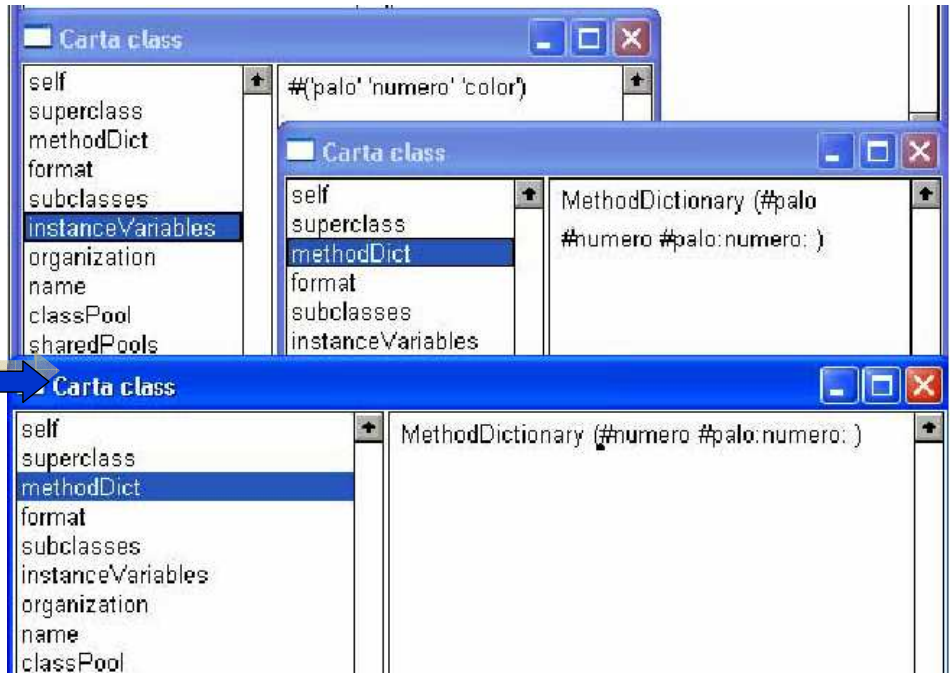
unaCarta inspect



Herencia de variables y métodos de instancia

Carta addInstVarName:'color'

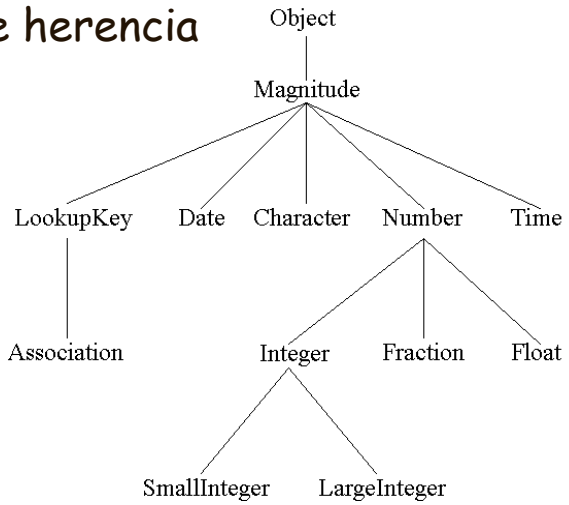
Carta removeSelector:'#palo:'



Jerarquía de clases

- La subclase hereda las variables y métodos de instancia de su superclase
- Jerarquía de herencia

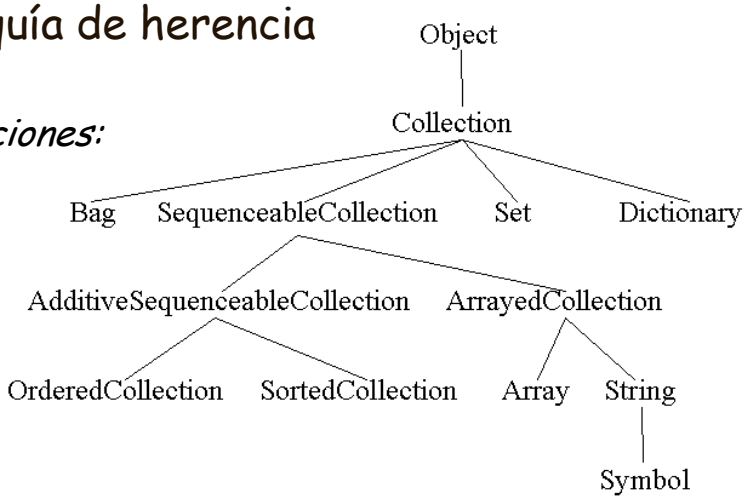
Magnitudes:



Jerarquía de clases

- La subclase hereda las variables y métodos de instancia de su superclase
- Jerarquía de herencia

Colecciones:




```
Object subclass:#Empleado instanceVariableNames:''
classVariableNames: 'SueldoBase' ...

SueldoBase
    SueldoBase :=  1000.
    ^ SueldoBase

sueldo
   ^((self class) SueldoBase) "o ^self class SueldoBase pero no
                                self SueldoBase ¿por qué?"

sueldoInc: num
    ^((self sueldo)*num)

Empleado subclass:#EmpleadoAsalariado
instanceVariableNames:''
classVariableNames:'' ...

SueldoBase
    SueldoBase := 2000.
    ^ SueldoBase

EmpleadoAsalariado subclass:#Director
instanceVariableNames:''
classVariableNames: '' ...

SueldoBase
    SueldoBase := 3000.
    ^ SueldoBase

sueldo
   ^((super class) SueldoBase)

sueldoInc: num
    ^((super sueldo)*num)
```

Self y Super

Superclases abstractas

- Collection
  - do: aBlock
    ^self subclassResponsibility

- Una superclase abstracta tiene métodos abstractos.
- Nunca se instancia

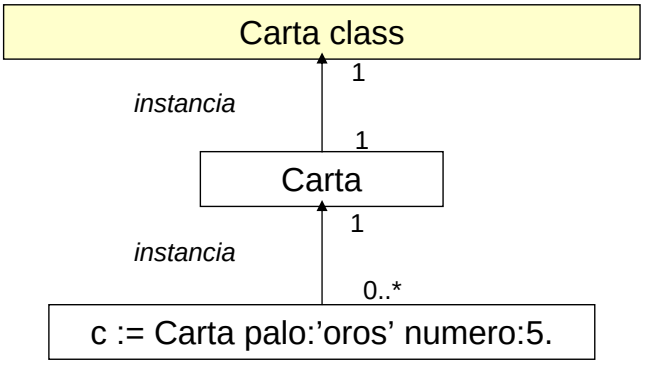
Herencia de variables y métodos de clase

- Los nombres de las variables de clase y los métodos de clase se almacenan en la metaclase
  - La metaclase de la que es instancia una clase se referencia con el nombre de la clase seguido del mensaje **class**

Clase	Metaclase
Object	Object class
Collection	Collection class
String	String class

Concepto de metaclase

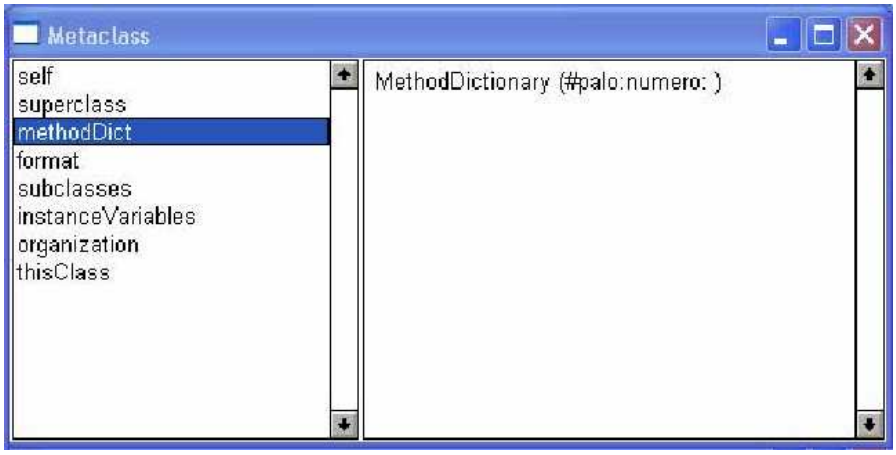
- La metaclase es una clase
  - Se puede instanciar (tiene sólo una instancia)
  - Se puede inspeccionar, añadir y borrar variables y métodos de clase



Concepto de metaclase

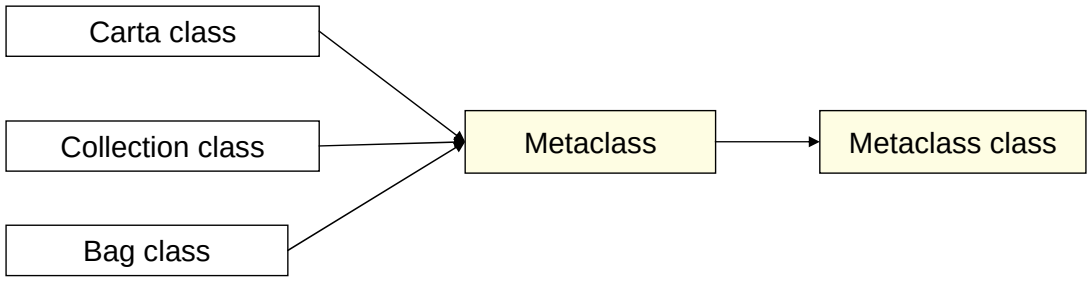
- La metaclase es una clase
  - Se puede instanciar (tiene sólo una instancia)
  - Se puede inspeccionar, añadir y borrar variables y métodos de clase

(Carta class) inspect



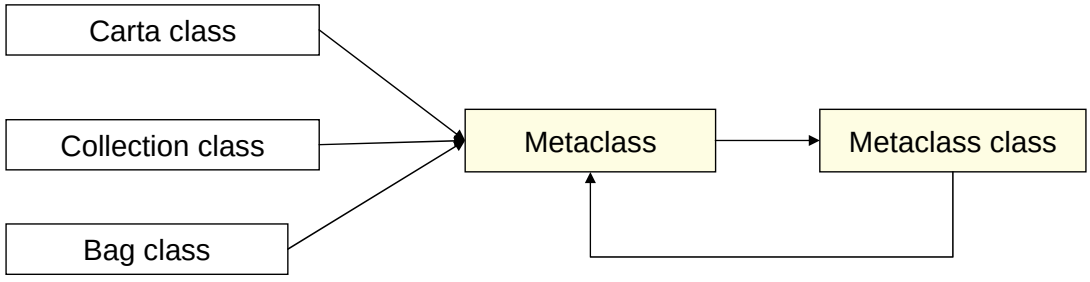
Concepto de metaclase

- La metaclase es un objeto
  - Su clase es Metaclass
  - La metaclase de Metaclass es Metaclass class



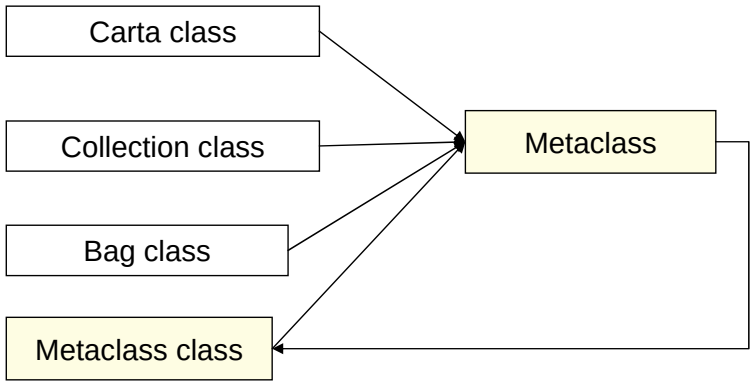
Herencia de la metaclase

- La metaclase es un objeto
  - Su clase es MetaClass
  - La metaclase de MetaClass es MetaClass class



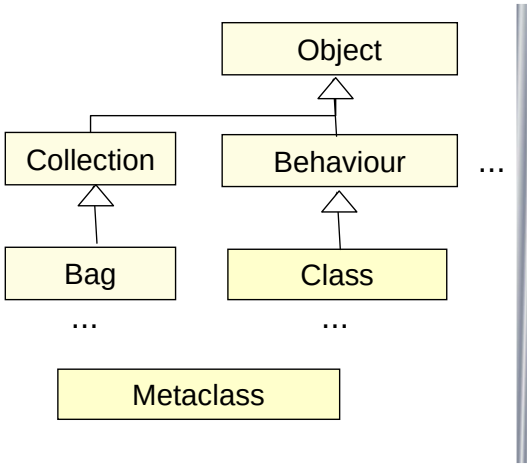
Herencia de la metaclase

- La metaclase es un objeto
  - Su clase es MetaClass
  - La metaclase de MetaClass es MetaClass class



Herencia de la metaclase

- Jerarquía de herencia con metaclases



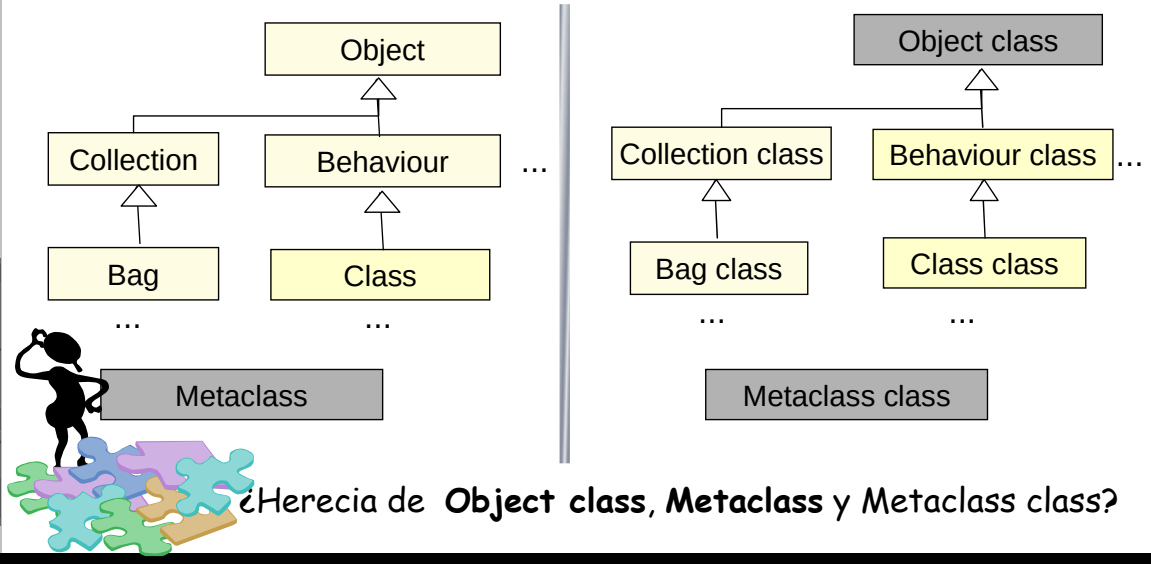
Reflejo de la jerarquía de clases

...

...

Herencia de la metaclase

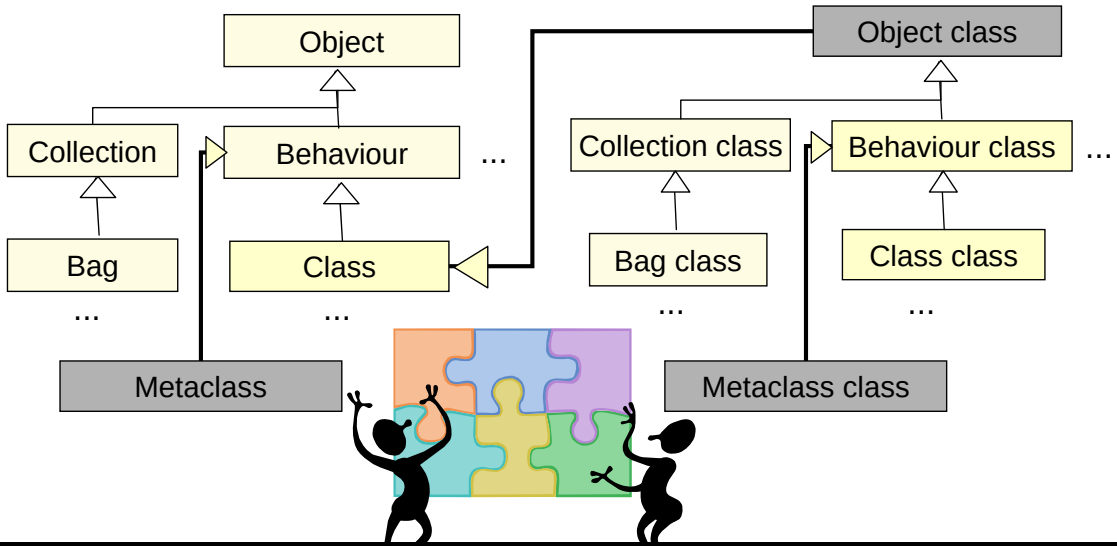
- Jerarquía de herencia con metaclases



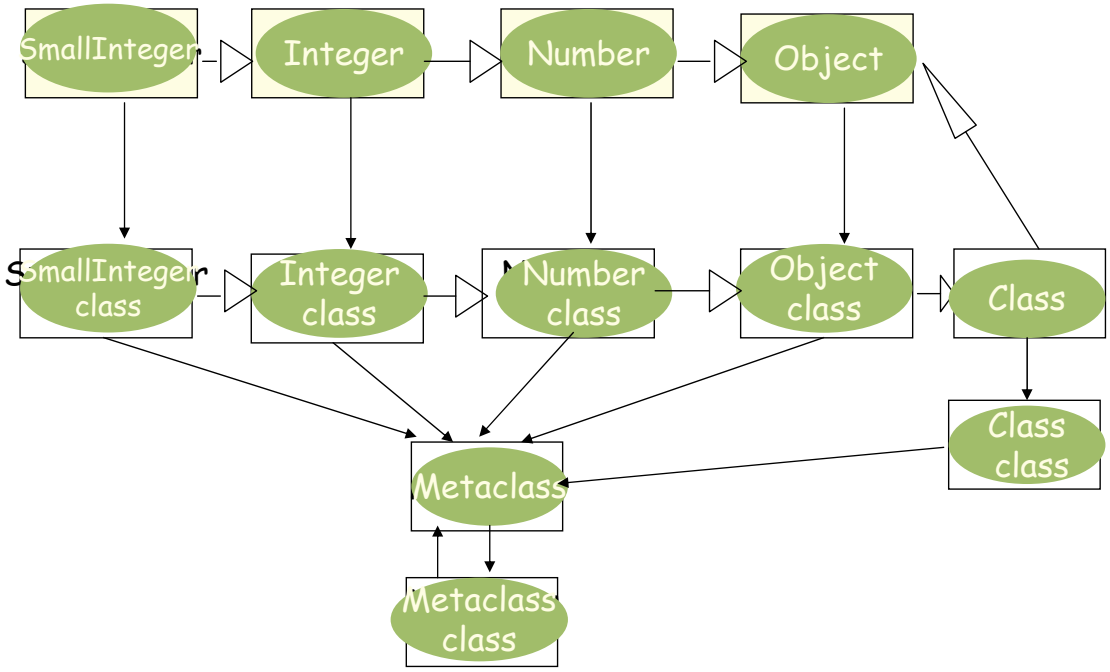
¿Herecia de **Object class**, **Metaclass** y Metaclass class?

Herencia de la metaclase

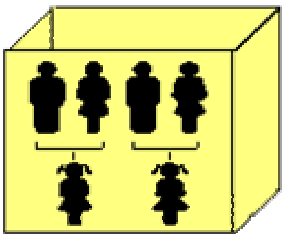
- Jerarquía de herencia con metaclases



Herencia e instanciación de las metaclases



Tema 5



Lección 3
Herencia en Java

Puntos a tratar

- Herencia de clases
- Herencia de interfaces
- Polimorfismo y herencia
- Estructura de clases en JAVA: colecciones
- La clase Class
- Simulación de herencia múltiple con interfaces

Herencia de clases

[public] [abstract | final] class NomClase
extends NomSuperClase

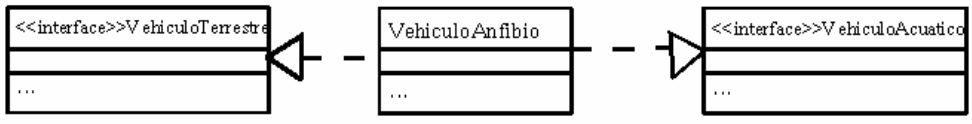
Sólo se permite herencia simple

- Una clase abstracta no puede instanciarse, pero permite garantizar que un conjunto de clases tenga un **protocolo común**
- ¿Cómo se declaran métodos de clase abstractos en Java?

Herencia de interfaces

- ¿Por qué añadir interfaces si existen las clases abstractas?
  1. Las interfaces permiten simular herencia múltiple mediante la posibilidad de que una clase
     - implemente varias interfaces
     - herede de otra clase e implementa una interfaz

```
class VehiculoAnfibio implements VehiculoTerrestre, VehiculoAcuatico { ... }
```

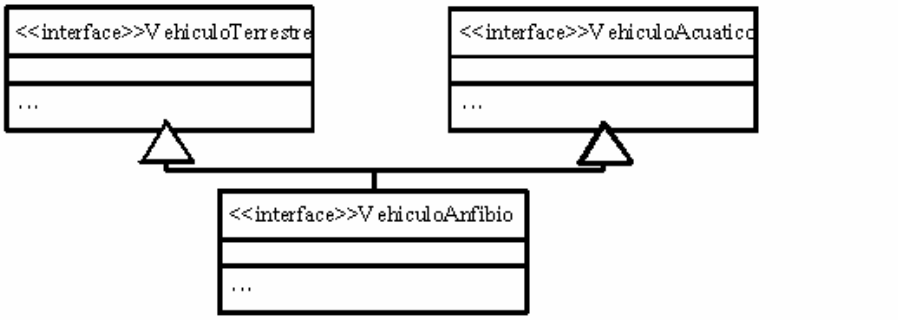


Herencia de interfaces

2. Se permite herencia múltiple entre interfaces:

[**public**] **interface** NombInterface
extends Interface1, Interface2, …

interface VehiculoAnfibio extends VehiculoTerrestre, VehiculoAcuatico { … }



Herencia en interfaces

- Diferencias entre interfaz y clase abstracta
  - Una clase puede tener variables declaradas
    - Una interfaz no declara ninguna variable
  - Una clase abstracta puede tener métodos implementados
    - La interfaz no tiene ningún método implementado
  - Una clase abstracta sólo puede heredar de una clase
    - La interfaz puede heredar de varias interfaces

Simplifica la herencia múltiple

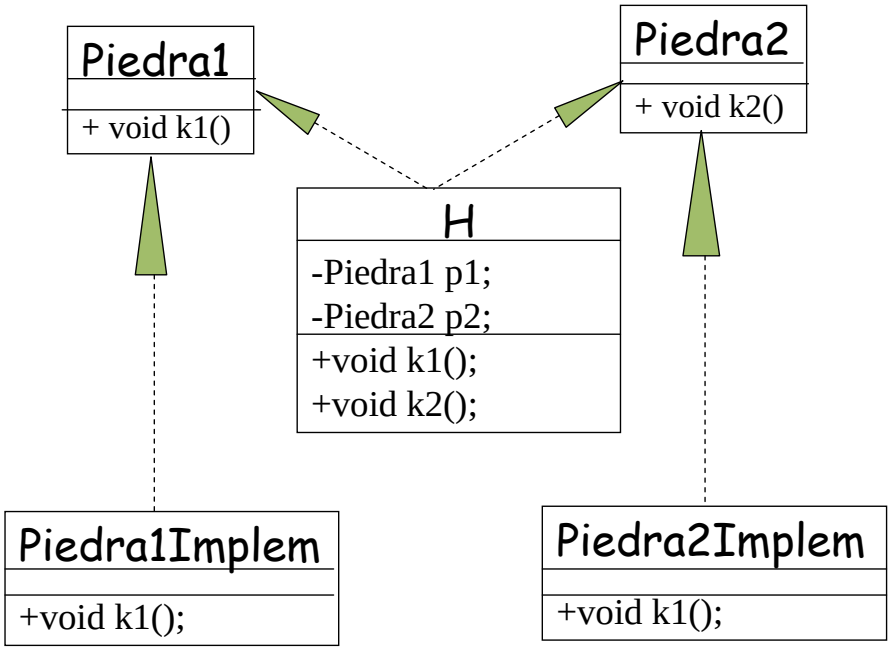
Polimorfismo y herencia

- Las interfaces permiten polimorfismo sin relación de herencia
  - NombreInterfaz var;
  - var puede referenciar distintas clases sin ninguna relación de herencia entre ellas

Simulación de herencia múltiple con interfaces

- Las interfaces permiten simular la herencia múltiple
  - Una clase hereda de otra clase e implementa una interfaz
  - Una clase implementa varias interfaces

Simulación de herencia múltiple con interfaces



Simulación de herencia múltiple con interfaces

```
Interface Piedra1{public void k1();};
Interface Piedra2(){public void k2();};

Class Piedra1Implem() implements Piedra1{public void k1(){};};

Class Piedra2Implem() implements Piedra2{public void k2(){};};

Class H implements Piedra1, Piedra2 {
 private Piedra1 p1;
 private Piedra2 p2;
 public H(){
 p1 = new Piedra1Impl();
 p2 = new Piedra2Impl();
 public void k1(){p1.k1();}
 public void k2(){p2.k2();} }
```

De esta manera
H "ES UNA" Piedra1, y
H "ES UNA" Piedra2.

La clase class

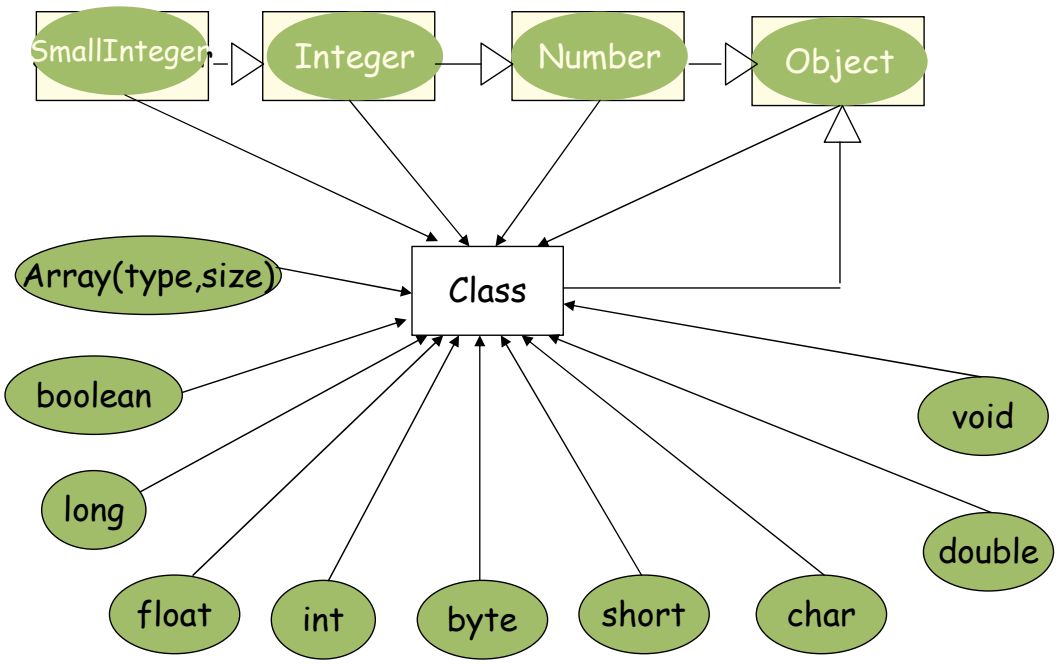
- Ejemplo:

```
void printClassName(Object obj) {
System.out.println("La clase de " + obj + " es " +
    obj.getClass().getName()); }
```

Se puede acceder también al objeto Class usando el literal "class":

```
System.out.println("El nombre de la clase Foo es:
    "+Foo.class.getName());
```


La clase Class



La clase Class

Algunos métodos de la clase Class:

Method[] **getMethods**()
Incluye los métodos heredados

String **getName**()

boolean **isInterface**()

Constructor[]**getConstructors**()