



Programación Orientada Objeto Aplicación al lenguaje JAVA



Institut de recherche
pour le développement

Jérémie HABASQUE – 2007
mailto:jeremie_habasque@yahoo.fr

La formación

➤ **Es una formación de programación Java**

➤ **Objetivo general**

Introducción de los conceptos de la programación orientada objeto a través del aprendizaje y la experimentación del lenguaje JAVA.

➤ **Objetivos operacionales**

Este curso proporciona un enfoque teórico y práctico de la programación orientada objetos (P.O.O.) con Java que tiene por objetivos el aprendizaje de:

- Tomada en mano de las herramientas de desarrollo Java
- Escritura de programas simples en Java
- Realización y escritura de clases simples
- Utilización de tablas, familiarizarse con los conceptos de herencia, polimorfismo
- Utilización de las excepciones, de package y de colecciones
- Utilización de un EDI (Entorno de Desarrollo Integrado): NetBeans

La formación

➤ Desarrollo

- Cursos y ejercicios en 10 sesiones de 2h : 20h

➤ Para quién ?

- Destino a informáticos, investigadores, ingenieros, estudiantes que quieren desarrollar aplicaciones

➤ Requeridos

- La experiencia práctica o teórica de un lenguaje de programación estructurada (C, FORTRAN, Pascal, Cobol, etc.), sería útil.

➤ Porque Java ?

- Pedagógico porque es un ejemplo de la programación orientada objeto
- Características:
 - verdadero lenguaje orientado objeto;
 - disponibilidad, gratuidad;
 - plataforma independiente y de desarrollo completa (lenguaje + herramientas);
 - lenguaje muy utilizado en el mundo profesional;
 - facilidad de transposición a otros lenguajes.

Programación estructurada vs. POO

➤ Objetivos de la POO

- ❑ Facilidad de reutilización del código, encapsulación y abstracción
- ❑ Facilidad de evolución del código
- ❑ Mejorar la concepción y el mantenimiento de grandes sistemas
- ❑ Programación por « componentes ». Concepción de un software a la manera de la fabricación de un carro

➤ Programación estructurada

- ❑ Unidad lógica : el modulo
- ❑ Una zona para las variables
- ❑ Una zona para las funciones
- ❑ Estructuración « descendente » del programa
- ❑ Cada función soluciona una parte del problema

```
program Hola;  
procedure DecirHola(n: string);  
begin  
  writeln('hola');  
end;  
  
var a:string;  
begin  
  write('ingresar su nombre: ');  
  readln(a);  
  DecirHola(a);  
end.
```

Príncipes POO

➤ Programación por objetos

- Unidad lógica : el objeto
- Un objeto esta definido por
 - un estado
 - un comportamiento
 - una identidad
- Estado : representado por atributos (variables) que almacén valores
- Comportamiento : definido por métodos (procedimientos) que modifican estados
- Identidad : permite de distinguir un objeto de otro objeto

miCarro
- Color = azul - Velocidad = 100

Príncipes POO

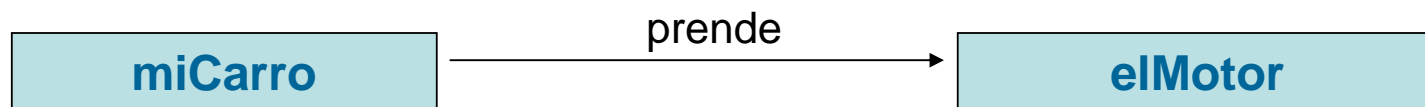
➤ Los objetos comunican entre ellos por mensajes

□ Un objeto puede recibir un mensaje que activa :

□ un método que modifica su estado

y / o

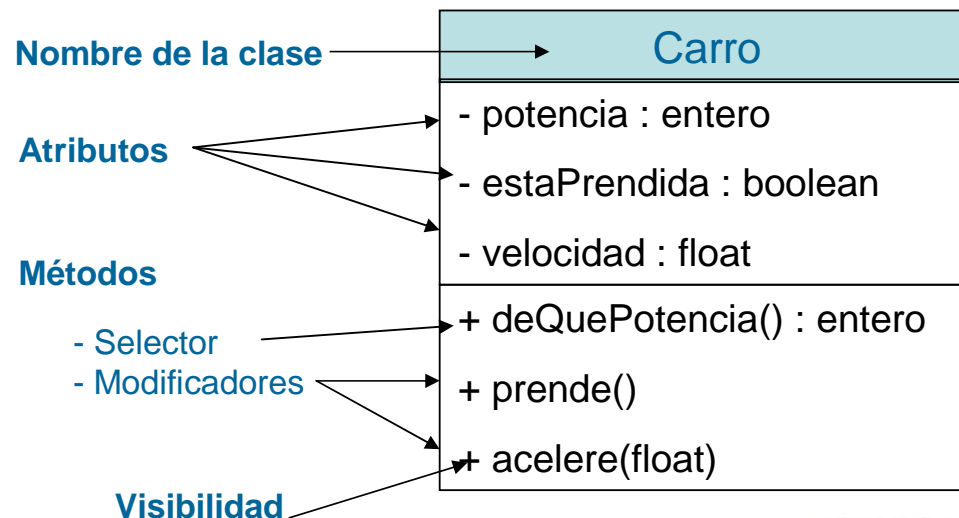
□ un método que envía un mensaje a otro objeto



Príncipes POO

➤ Noción de clase

- Se agrupan los objetos que tienen los mismos estados y los mismos comportamientos: es una clase
- Las clases sirven de « moldes » para la creación de objetos : un objeto es una « instancia » de una clase
- Un programa OO esta constituido de clases que permiten de crear objetos que se envían mensajes



Príncipes POO

- El conjunto de interacciones entre los objetos define un algoritmo
- Las relaciones entre las clases reflejan la descomposición del programa



De que tendrán necesidad ?

➤ **Todas las herramientas y documentaciones son en el CD de formación :**

- El Java SE Development Kit JDK 6.0
(Contiene un compilador, un interpretador, clases básicas y otras herramientas)
- Entornos de desarrollo
 - NetBeans 5.5 + JDK 6.0 (para Windows y Linux)
 - Eclipse 3.2.1
- Un manual de utilización de NetBeans
- La documentación sobre el API de Java
- El tutorial de SUN
- Este curso al formato PDF más los códigos Java de los ejemplos
- Enunciados de los ejercicios al formato HTML

Desarrollo del curso ...

➤ Estructuración del curso

- Presentación de los conceptos
- Ilustración con muchos ejemplos
- Burbujas de ayuda a lo largo del curso:



Eso es una alerta



Eso es una astucia

➤ Instauración del curso

- Curso de Mickael Baron <http://mbaron.developpez.com/>
- Curso de Patrick Itey
<http://www-sop.inria.fr/semir/personnel/Patrick.Itey/cours/index.html>
- Curso de Philippe Genoud
<http://www.inrialpes.fr/helix/people/genoud/ENSJAVA/M2CCI/cours.html>

Organización ...

- Parte 1 : Introducción al lenguaje JAVA
- Parte 2 : Bases del lenguaje
- Parte 3 : Clases y objetos
- Parte 4 : Herencia
- Parte 5 : Herencia y polimorfismo
- Parte 6 : Los indispensables : package, jar, javadoc, flujos I/O, collection y exception





Programación Orientada Objeto Aplicación al lenguaje JAVA

Introducción al lenguaje Java



Institut de recherche
pour le développement

Jérémie HABASQUE – 2007

mailto:jeremie_habasque@yahoo.fr

Java ?

➤ **Una tecnología desarrollada por SUN Microsystems lanzado en 1995**

➤ **Referencias**

- ❑ Wikipedia : http://es.wikipedia.org/wiki/Java_%28Sun%29
- ❑ White papers : <http://java.sun.com/docs/white/langenv/index.html>

➤ **Sun define el lenguaje Java como**

- ❑ Sencillo
- ❑ Distribuido
- ❑ Robusto
- ❑ Portable
- ❑ Multithreaded
- ❑ Orientado objeto
- ❑ Arquitectura neutra
- ❑ Seguro
- ❑ Rendimiento medio
- ❑ Lenguaje dinámica

...

Príncipe de funcionamiento de Java

➤ Fuente Java : archivo .java

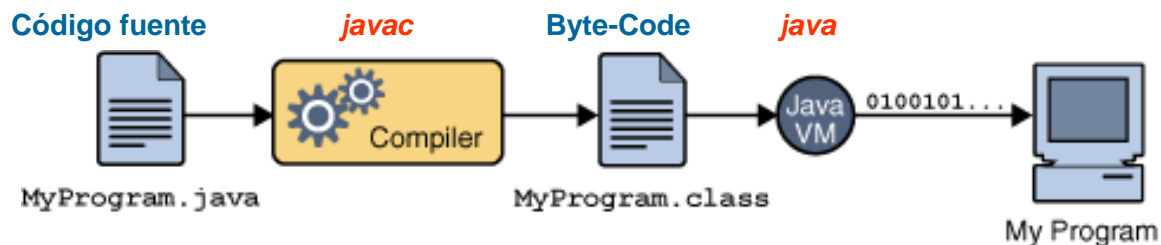
- ❑ Archivo utilizado durante la fase de programación
- ❑ El único archivo realmente inteligible por el programador !

➤ Byte-Code Java : archivo .class

- ❑ Código objeto destinado a ser ejecutado en toda « Máquina Virtual » Java
- ❑ Procede de la compilación del código fuente

➤ Máquina Virtual Java

- ❑ Programa interpretando y ejecutando el Byte-Code Java. Este máquina virtual es específica a sistema de explotación.
- ❑ Conclusión : es suficiente disponer de una « Máquina Virtual » Java para poder ejecutar todo programa Java incluso si se compiló con otro sistema de explotación (Windows, MacOS X, Linux...).



“Compile once, run everywhere”

Maquinas virtuales Java

➤ Navegadores WEB, estaciones de trabajo, Network Computers

➤ WebPhones

➤ Celulares

➤ Tarjeta inteligente

➤ ...



Principales etapas de un desarrollo

➤ Creación del código fuente : archivo .java

- A partir de especificaciones (por ejemplo en UML)
- Herramienta : editor de texto, IDE

➤ Compilación en Byte-Code : archivo .class

- A partir del código fuente
- Herramienta : compilador Java

➤ Difusión en la arquitectura objetiva

- Transferencia del Byte-Code solo
- Herramientas : network, disco, etc.

➤ Ejecución en la maquina objetiva

- Ejecución del Byte-Code
- Herramienta : maquina virtual Java

Java y sus versiones

➤ Diferentes versiones de la maquina virtual



**Standard Edition
JSE**

Proporciona los compiladores, herramientas, runtimes, y APIs para escribir, desplegar, y realizar applets y aplicaciones en el lenguaje de programación Java



**Enterprise Edition
JEE**

Destinada al desarrollo de aplicaciones “de empresa” (“business applications”) robustas y ínter operables. Simplificar el desarrollo y el despliegue de aplicaciones distribuidas y articuladas alrededor de la red.



**Mobile Edition
JME**

Medio ambiente de ejecución optimizado para los dispositivos “ligeros”:

- Tarjeta inteligente (smart cards)
- Teléfonos móviles
- Ayudantes personales (PDA)



En este curso, vamos a estudiar principalmente los API proporcionado por Java SE

Java y sus versiones

➤ Diferentes finalidades

- SDK (Software Development Kit) proporciona un compilador y una maquina virtual
- JRE (Java Runtime Environment) proporciona únicamente una maquina virtual. Ideal para el despliegue de sus aplicaciones.

➤ Versión actual de Java

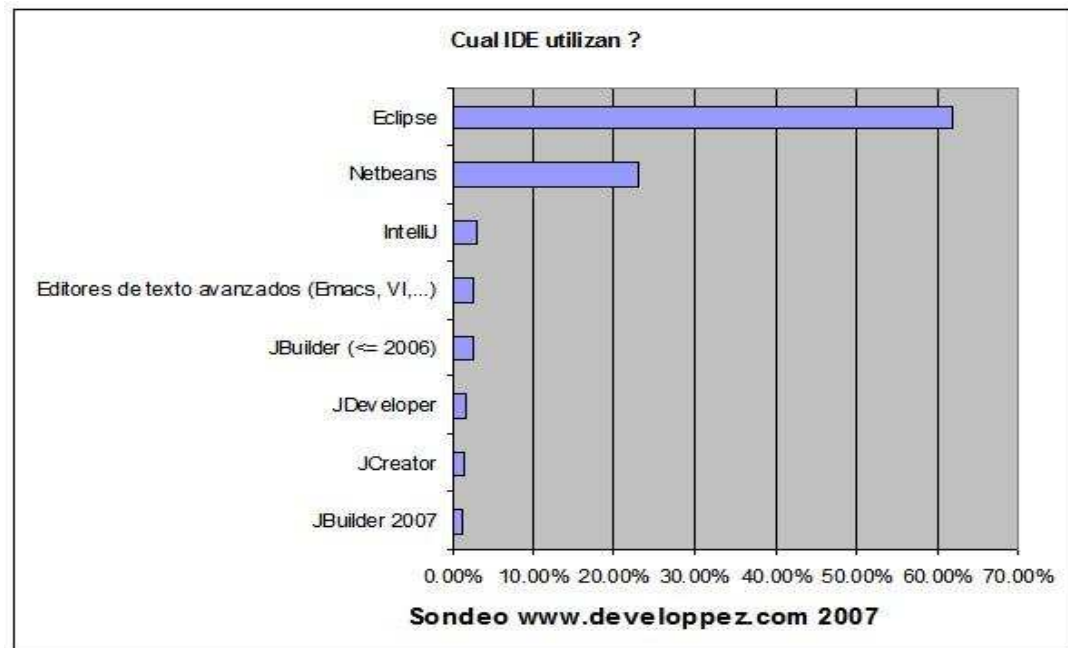
- Actualmente « Java SE 6.0 » o también llamada « JDK 6.0 »
- En 2008, Java SE 7.0 (Dolphin) : paso de Java en la comunidad open source

Versiones Windows, Linux, Solaris se descargan sobre <http://java.sun.com/>

Herramientas...

➤ Simples editores o entornos de desarrollo comerciales y open source

- Eclipse
- NetBeans
- JBuilder
- ...



➤ Los recursos sobre Java

- Sitio WEB oficial Java de SUN : <http://java.sun.com>
- API : <http://java.sun.com/javase/6/docs/api/index.html>
- Tutorial de Sun : <http://java.sun.com/docs/books/tutorial/index.html>
- Cursos y foros : <http://www.javahispano.com/>
- Cursos y foros : <http://www.programacion.net/java/>

El API de Java

- Extensa colección de componentes informáticos (clases e interfaces)
- Organizada en bibliotecas (*packages*)
- Ofrezca numerosos servicios de manera estándar (independientemente de la plataforma material)

User Interface Toolkits	AWT		Swing		Java 2D™		
	Accessibility	Drag 'n Drop	Input Methods	Image I/O	Print Service	Sound	
Integration Libraries	IDL	JDBC™	JNDI™	RMI	RMI-IIOP		
Other Base Libraries	Beans	Int'l Support	I/O	New I/O	JMX	JNI	Math
	Networking	Std. Override Mechanism	Security	Serialization	Extension Mechanism	XML	JAXP
lang & util Base Libraries	Lang & Util	Collections	Concurrency Utilities	JAR	Logging	Management	
	Preferences	Ref Objects	Reflection	Regular Expressions	Versioning	Zip	



**Programar en Java requiere un buen conocimiento del API.
El aprendizaje puede ser largo**

El API de Java

Packages

The screenshot displays the Java API documentation for Java Platform, Standard Edition 6. The left sidebar contains a list of packages and classes. The main content area shows the 'Packages' section with a table listing various packages and their descriptions.

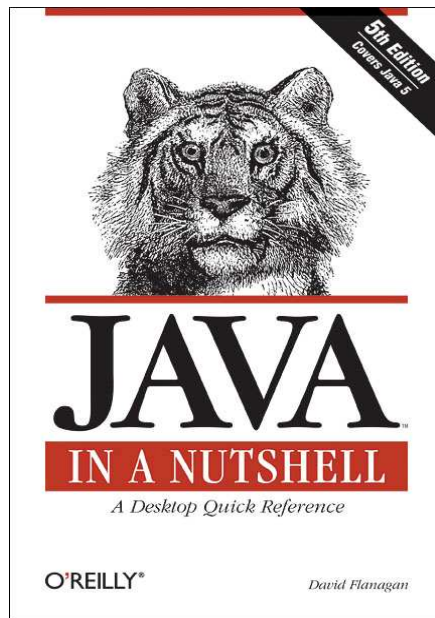
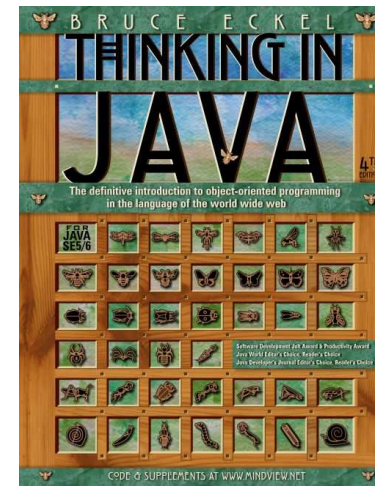
Package	Description
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.
java.awt.image	Provides classes for creating and modifying images.
java.awt.image.renderable	Provides classes and interfaces for producing rendering-independent images.
java.awt.print	Provides classes and interfaces for a general printing API.
java.beans	Contains classes related to developing <i>beans</i> -- components based on the JavaBeans™ architecture.
java.beans.beancontext	Provides classes and interfaces relating to bean context.
java.io	Provides for system input and output through data streams, serialization and the file system.

Clases

Descripción, atributos, métodos

Libros de Java

« Thinking in Java », Bruce Eckel - Prentice-Hall
(www.BruceEckel.com)



« JAVA in a nutshell, 5th Edition »,
David Flanagan - O'Reilly 2005



Programación Orientada Objeto Aplicación al lenguaje JAVA

Bases del lenguaje

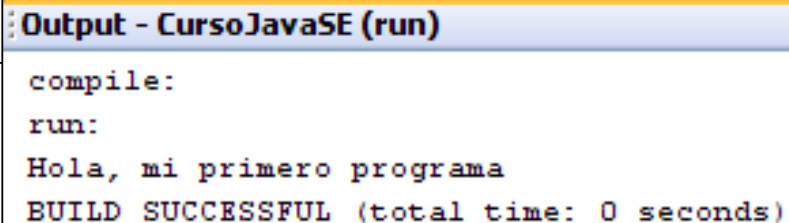


Institut de recherche
pour le développement

Jérémie HABASQUE – 2007
mailto:jeremie_habasque@yahoo.fr

Primer ejemplo de programa Java

```
public class PrimerPrograma {  
  
    public static void main(String[ ] args) {  
        System.out.println("Hola, mi primer programa");  
    }  
}
```



Output - CursoJavaSE (run)

```
compile:  
run:  
Hola, mi primero programa  
BUILD SUCCESSFUL (total time: 0 seconds)
```

➤ *public class PrimerPrograma*

- Nombre de la clase

➤ *public static void main*

- La función principal equivalente a la función *main* de C/C++

➤ *String[] argv*

- Permite de recuperar argumentos transmitidos al programa al momento de su lanzamiento

➤ *System.out.println("Hola ... ")*

- Método de visualización en la ventana consola

Aplicación

➤ No separación entre definición y implementación de operaciones

- Un solo archivo « NombreDeClase.java »
- No archivo de encabezado como C/C++



Nombre de la clase = nombre del archivo Java

➤ Compilación

- `javac NombreDeClase.java` o `javac *.java` cuando varias clases
- Generación de un archivo Byte-Code « NombreDeClase.class »
- No edición de vínculos (solamente una verificación)



No poner la extensión .class para la ejecución

➤ Ejecución

- `java NombreDeClase`
- Escoger la clase principal a ejecutar

Tipos primitivos de Java

- No son objetos !!!
- Ocupen una place fija en memoria reservada a la declaración
- Tipos primitivos :
 - ❑ Enteros : **byte** (1 octeto) - **short** (2 octetos) - **int** (4 octetos) - **long** (8 octetos)
 - ❑ Flotantes (norma IEEE-754) : **float** (4 octetos) - **double** (8 octetos)
 - ❑ Booleans : **boolean** (true o false)
 - ❑ Caracteres : **char** (implementación Unicode sobre 16 bits)
- Cada tipo simple tiene un alter-ego objeto disponiendo de métodos de conversión (a ver en la parte Clases et Objetos)
- El autoboxing introduce desde la versión 5.0 convierte de manera transparente los tipos primitivos en referencia

Inicialización y constantes

➤ Inicialización

- Una variable puede recibir un valor inicial al momento de su declaración :

```
int n = 15;  
boolean b = true
```

- Este instrucción tiene el mismo papel :

```
int n;  
n = 15;  
boolean b;  
b = true;
```



Pensar a la inicialización al riesgo de una error de compilación

```
int n;  
System.out.println(" n = " + n);
```

```
Output - CursoJavaSE (run)  
Compiling 1 source file to E:\workspace\CursoJavaSE\build\classes  
E:\workspace\CursoJavaSE\src\cursojavase\PrimeroPrograma.java:27:  
variable n might not have been initialized  
    System.out.println(" n = " + n);  
1 error  
BUILD FAILED (total time: 1 second)
```

➤ Constantes

- Son variables cuyas valor cual se puede afectar una sola vez
- No pueden ser modificadas
- Son definidas con la palabra clave **final**

```
final int n = 5;  
final int t;  
...  
t = 8;  
n = 10; // error : n esta declarado final
```

Estructuras de controles

➤ Elección

- Si luego sino : « **if** *condición* {...} **else** {...} »

No hay palabra clave « then »
en la estructura Elección



➤ Iteraciones

- Cerro : « **for** (*inicialización* ; *condición* ; *modificación*) { ... } »
- Mientras : « **while** (*condición*) {...} »
- Hacer hasta : « **do** {...} **while** (*condición*) »

➤ Selección limitada

- Según hacer : « **switch** *identificador* { **case** valor0 : ... **case** valor1 : ... **default**: ... } »
- La palabra clave **break** pide a salir del bloque

Pensar a averiguar si break es
necesario en cada caso



Estructuras de controles

➤ Ejemplo : estructura de controle

```
public class SwitchBreak {  
  
    public static void main(String[] argv) {  
        int n = 2;  
        System.out.println("Valor de n :" + n);  
        switch(n) {  
            case 0 : System.out.println("nulo");  
                break;  
            case 1 :  
            case 2 : System.out.println("pequeño");  
            case 3 :  
            case 4 :  
            case 5 : System.out.println("medio");  
                break;  
            default : System.out.println("grande");  
        }  
        System.out.println("Adiós...");  
    }  
}
```

➤ Hacemos variar n :

Valor de n : 0
nulo
Adiós

Valor de n : 1
pequeño
medio
Adiós

Valor de n : 6
grande
Adiós



Pedirse si break es necesario

Operadores sobre los tipos primitivos

➤ Operadores aritméticos

- Unarios : « +a, -b »
- Binarios : « a+b, a-b, a*b, a%b »
- Incrementación y decrementación : « a++, b-- »
- Afectación ampliada : « +=, -=, *=, /= »

➤ Operadores de comparación

- « a==b, a!=b, a>b, a<b, a>=b, a<=b »

➤ Operadores lógicos

- And : « a && b », « a & b »
- Or : « a || b », « a | b »

➤ Conversión de tipo explicita (cast)

- « (NuevoTipo)variable »



Atención : error

```
boolean t = true;  
if (t = true) {...}
```

Preferir :

```
boolean t = true;  
if (t) {...}
```

Operadores sobre tipos primitivos

➤ Ejemplo de la lotería

- Muestra la utilización de los conceptos precedentes

```
public class OperadoresTiposPrimitivos {  
  
    public static void main(String[] argv) {  
        int contador = 0;  
  
        while(contador != 100) {  
            // Tomar un numero aleatorio  
            double numeroAleatorio = Math.random() * 1000;  
  
            // Establece un indexo de 0 a 10  
            int indexo = contador % 10;  
  
            // Construcción de la visualización  
            System.out.println("Indexo:" + indexo +  
                " Numero aleatorio:" + (int)numeroAleatorio);  
  
            // Incrementación del cerro  
            contador+= 1;  
        }  
    }  
}
```

A ver mas tarde ...

```
Output - CursoJavaSE (run-single)  
Indexo:0 Numero aleatorio:582  
Indexo:1 Numero aleatorio:500  
Indexo:2 Numero aleatorio:411  
Indexo:3 Numero aleatorio:258  
Indexo:4 Numero aleatorio:357
```

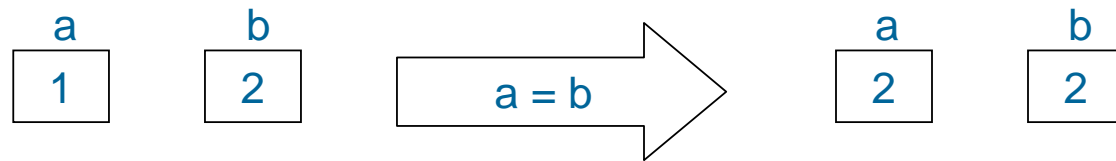
Afectación, copia y comparación

➤ Afectar y copiar un tipo primitivo

- « $a=b$ » significa a toma la valor de b
- a y b son distintos
- Toda modificación de a implica la modificación de b

➤ Comparar un tipo primitivo

- « $a == b$ » devuelve « true » si los valores de a y b son iguales



Las tablas en Java

- Las tablas son considerados como **objetos**
- Proporcionan colecciones ordenadas de elementos
- Los elementos de una tabla pueden ser :
 - ❑ Variables de tipo primitivo (int, boolean, double, char, ...)
 - ❑ Referencias sobre objetos (a ver en la parte Clases y Objetos)
- Creación de una tabla
 - ① Declaración = determinar el tipo de la tabla
 - ② Dimensión = determinar el tamaño de la tabla
 - ③ Inicialización = inicializar cada casilla de la tabla

Las tablas en Java : declaración

① Declaración

- La declaración precisa simplemente el tipo de los elementos de la tabla

```
int[ ] miTabla;
```

```
miTabla null
```

- Puede escribirse también

```
int miTabla[ ];
```



Atención : una declaración de tabla no debe precisar dimensiones

```
int miTabla[5]; // Error
```

Las tablas en Java : dimensión

② Dimensión

- El numero de elementos de la tabla será determinada cuando el objeto tabla será efectivamente creado utilizando la palabra clave **new**
- El tamaño determinada a la creación de la tabla es fija, no podrá ser modificada mas tarde
- Longitud de una tabla : « **miTabla.length** »

```
int[ ] miTabla; // Declaración  
miTabla = new int[3]; // Dimensión
```

- La creación de una tabla por **new**
 - Asigna la memoria en función del tipo de la tabla y del tamaño
 - Inicializa el contenido de la tabla a 0 para los tipos simples



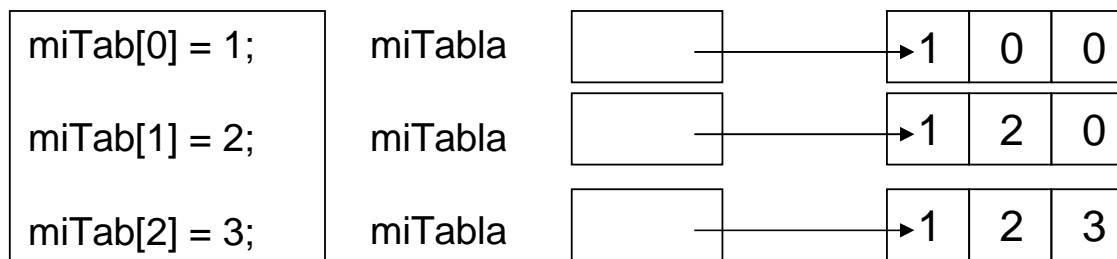
Las tablas en Java : Inicialización

3 Inicialización

- Como en C/C++ los índices empiezan a cero
- El acceso a un elemento de una tabla se efectúa según esta forma

```
miTab[varInt]; // varInt >= 0 y <length
```

- Java verifica automáticamente el índice durante el acceso (crea una excepción)



- Otro método : dando explícitamente la lista de sus elementos entre {...}

```
int[ ] miTab = {1, 2, 3}
```

□ es equivalente a

```
miTab = new int[3];  
miTab[0] = 1; miTab[1] = 2; miTab[2] = 3;
```

Las tablas en Java : síntesis

① Declaración

```
int[] miTabla;
```

② Dimensión

```
miTabla = new int[3];
```

③ Inicialización

```
miTabla[0] = 1;  
miTabla[1] = 2;  
miTabla[2] = 3;
```

o ① ② y ③

```
int[ ] miTab = {1, 2, 3};
```



```
for (int i = 0; i < miTabla.length; i++) {  
    System.out.println(miTabla[i]);  
}
```

Las tablas en Java : pluridimensionales

➤ Tablas cuyas elementos son ellos mismos tablas

➤ Declaración

tipo[][] miTabla;

➤ Tablas rectangulares

□ Dimensión :

miTabla = new tipo[2][3]

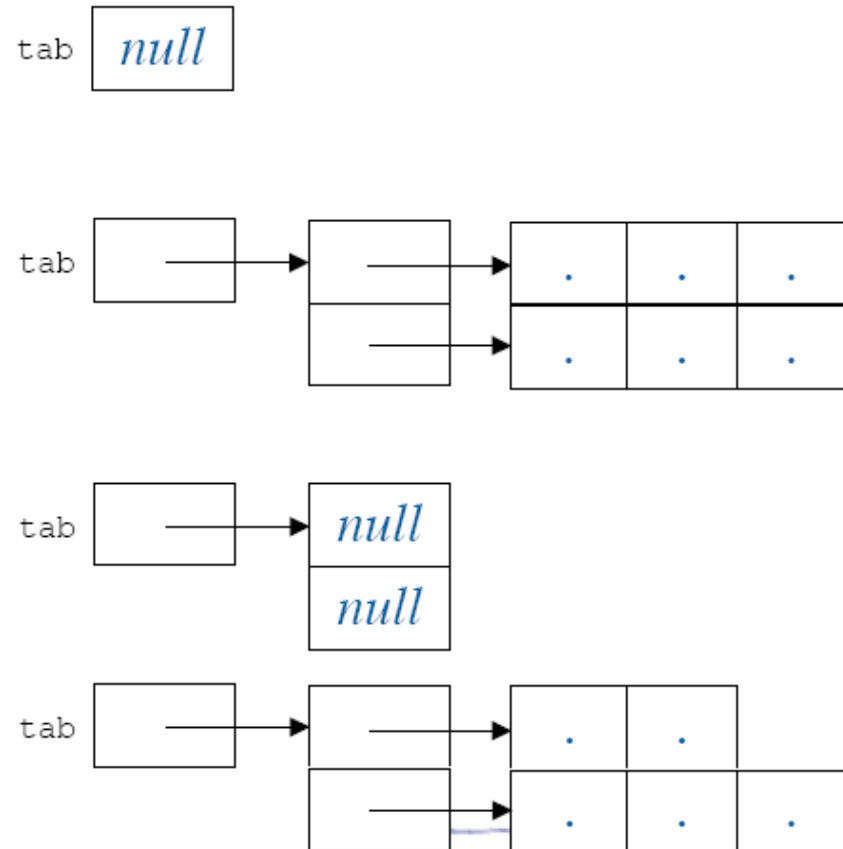
➤ Tablas non-rectangulares

□ Dimensión :

miTabla = new tipo[2]

miTabla[0] = new tipo[2]

miTabla[1] = new tipo[3]



Pequeña precisión del “System.out.println(...)”

➤ Usos : visualización a la pantalla

- « System.out.println(...) » : regresa a la línea
- « System.out.print(...) » : no regresa a la línea

➤ Diferentes salidas posibles

- « out » salida estándar
- « err » salida en caso de error (non temporizada)

➤ Todo lo que se puede visualizar ...

- Objetos, números, booleans, caracteres, ...

➤ Todo que se puede hacer ...

- Concatenación salvaje entre tipos y objetos con el « + »

```
System.out.println("a=" + a + "entonces a < 0 es " + a < 0);
```

Comentarios y puesta en forma

➤ Documentación de los códigos fuentes :



- Utilización des comentarios

// Comentario en una línea completa
int b = 34; // Comentario después el código
/ El inicio del comentario*
*** No puedo continuar a escribir ...*
*hasta que el compilador encuentra eso */*

- Utilización de la herramienta Javadoc (a ver en la parte indispensables)

➤ Puesta en forma

- Facilita la relectura
- Credibilidad asegurada !!!!
- Escotadura a cada nivel de bloque

<pre>if (b == 3) { if (cv == 5) { if (q) { ... } else { ... } ... } ... }</pre>	<pre>if (b == 3) { if (cv == 5) { if (q) { ... } else {...} ... } ... }</pre>
 Preferir	 Evitar



Programación Orientada Objeto Aplicación al lenguaje JAVA

Clases y objetos



Institut de recherche
pour le développement

Jérémie HABASQUE – 2007
mailto:jeremie_habasque@yahoo.fr

Clase y definición

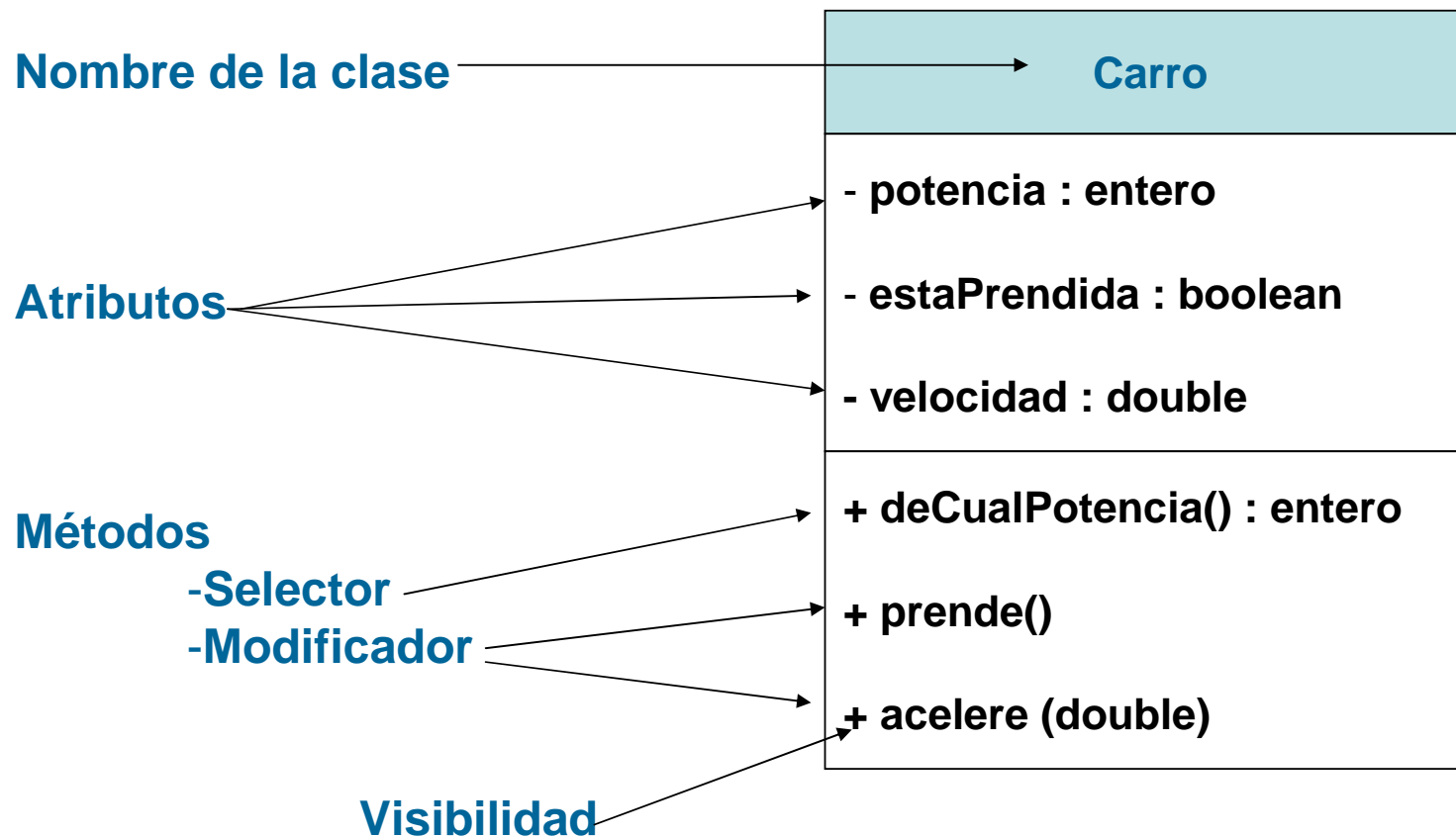
- Una clase esta constituida de:
 - Datos lo que se llaman **atributos**
 - Procedimientos y/o funciones lo que se llaman **métodos**

- Una clase es un modelo de definición para objetos
 - Que tienen misma estructura (mismo conjunto de atributos)
 - Que tienen mismo comportamiento (mismos métodos)
 - Que tienen una semántica comuna

- Los **objetos** son representaciones dinámicas (**instanciation**), del modelo definido para ellos a través de la clase
 - Una clase permite de **instanciar** (crear) varios objetos
 - Cada objeto es instancia de una clase y una sola

Clase y notación UML

Diagrama de clase



Codificación de la clase “Carro”

Nombre de la clase

Atributos

Selector

Modificadores

```
public class Carro {  
    private int potencia;  
    private boolean estaPrendida;  
    private double velocidad;  
  
    public int deCualPotencia() {  
        return potencia;  
    }  
  
    public void prende() {  
        estaPrendida = true;  
    }  
  
    public void acelere(double v) {  
        if (estaPrendida) {  
            velocidad = velocidad + v  
        }  
    }  
}
```

Clase y visibilidad de los atributos

➤ Característica de un atributo

- Variables « globales » de la clase
- Accesibles en todos los métodos de la clase

```
public class Carro {  
  
    private int potencia;  
    private boolean estaPrendida;  
    private double velocidad;  
  
    public int deCualPotencia() {  
        return potencia;  
    }  
  
    public void prende() {  
        estaPrendida = true;  
    }  
  
    public void acelere(double v) {  
        if (estaPrendida) {  
            velocidad = velocidad + v  
        }  
    }  
}
```

Atributos visibles
en los métodos

Distinción entre atributos y variables

➤ Característica de una variable :

- Visible a dentro del bloque que le define

```
public class Carro {  
  
    private int potencia;  
    private boolean estaPrendida;  
    private double velocidad;  
  
    public int deCualPotencia() {  
        return potencia;  
    }  
  
    public void prende() {  
        estaPrendida = true;  
    }  
  
    public void acelere(double v) {  
        if (estaPrendida) {  
            double conTolerancia;  
            conTolerancia = v + 25;  
            velocidad = velocidad + conTolerancia  
        }  
    }  
}
```

Variable visible únicamente a dentro de este método

Variable puede ser definida en cualquier parte del bloque

Algunas convenciones en Java : rigor y clase !

➤ Convenciones de nombres

- EsoEsUnaClase
- esoEsUnMétodo(...)
- yoSoyUnaVariable
- YO_SOY_UNA_CONSTANTE

➤ Un archivo por clase, una clase por archivo

- Clase « Carro » descrita en el archivo Carro.java
- Se puede excepcionalmente tener varias clases por archivo (caso de las *Inner classes*)



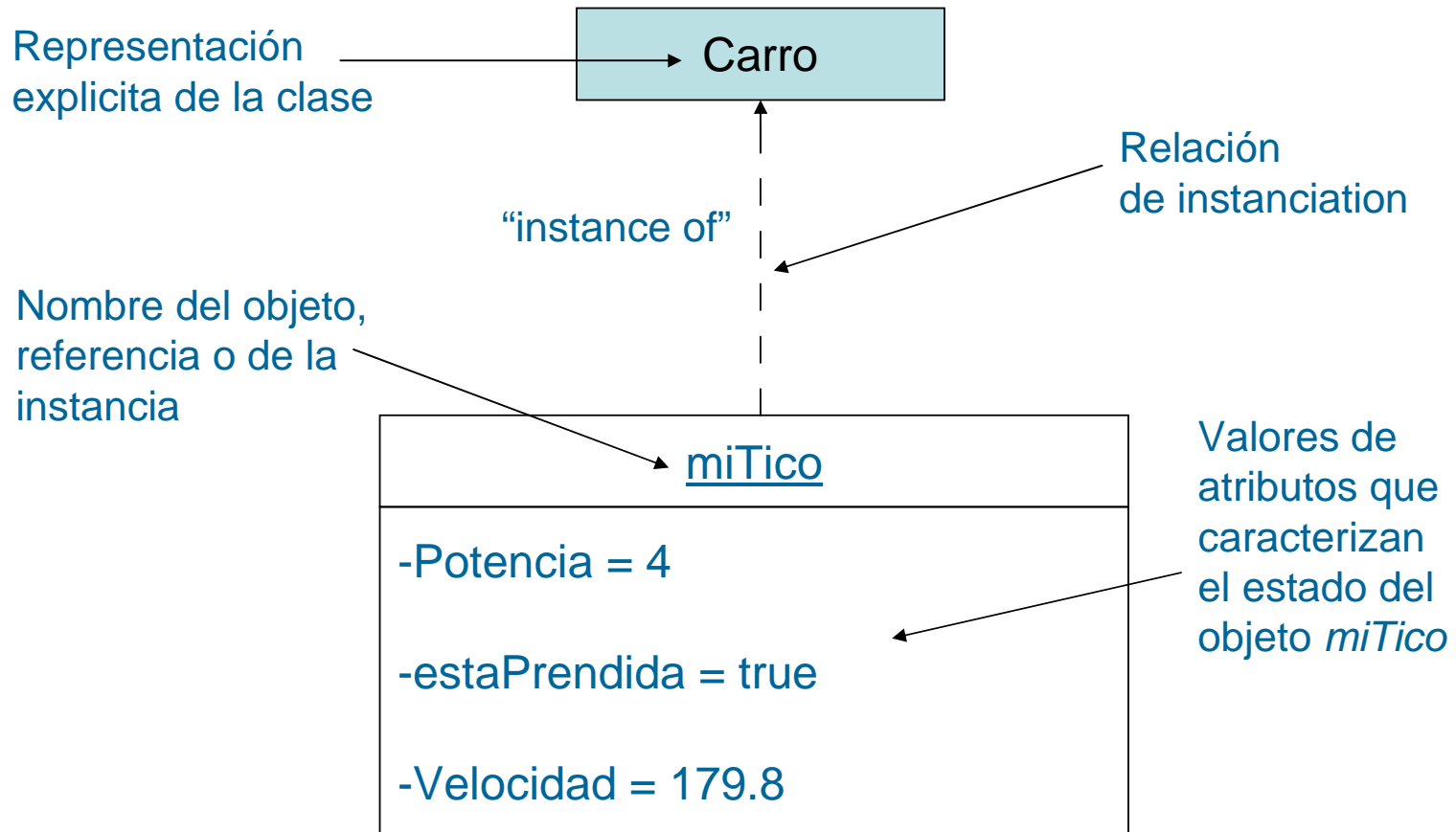
Respectar las minúsculas y las mayúsculas de nombres

Objeto y definición

- Un objeto es **instancia** de una sola clase :
 - ❑ Se conforme a la descripción que ella proporciona
 - ❑ Admite un valor propia al objeto para cada atributo declarado en la clase
 - ❑ Los valores de atributos caracterizan el **estado** del objeto
 - ❑ Posibilidad de aplicarle toda operación (**método**) definida en la clase
- Todo objeto es manipulado y identificado por su referencia
 - ❑ Utilización de indicador oculto (mas accesible que el C++)
 - ❑ Se habla indiferentemente de **instancia**, de **referencia** o de **objeto**

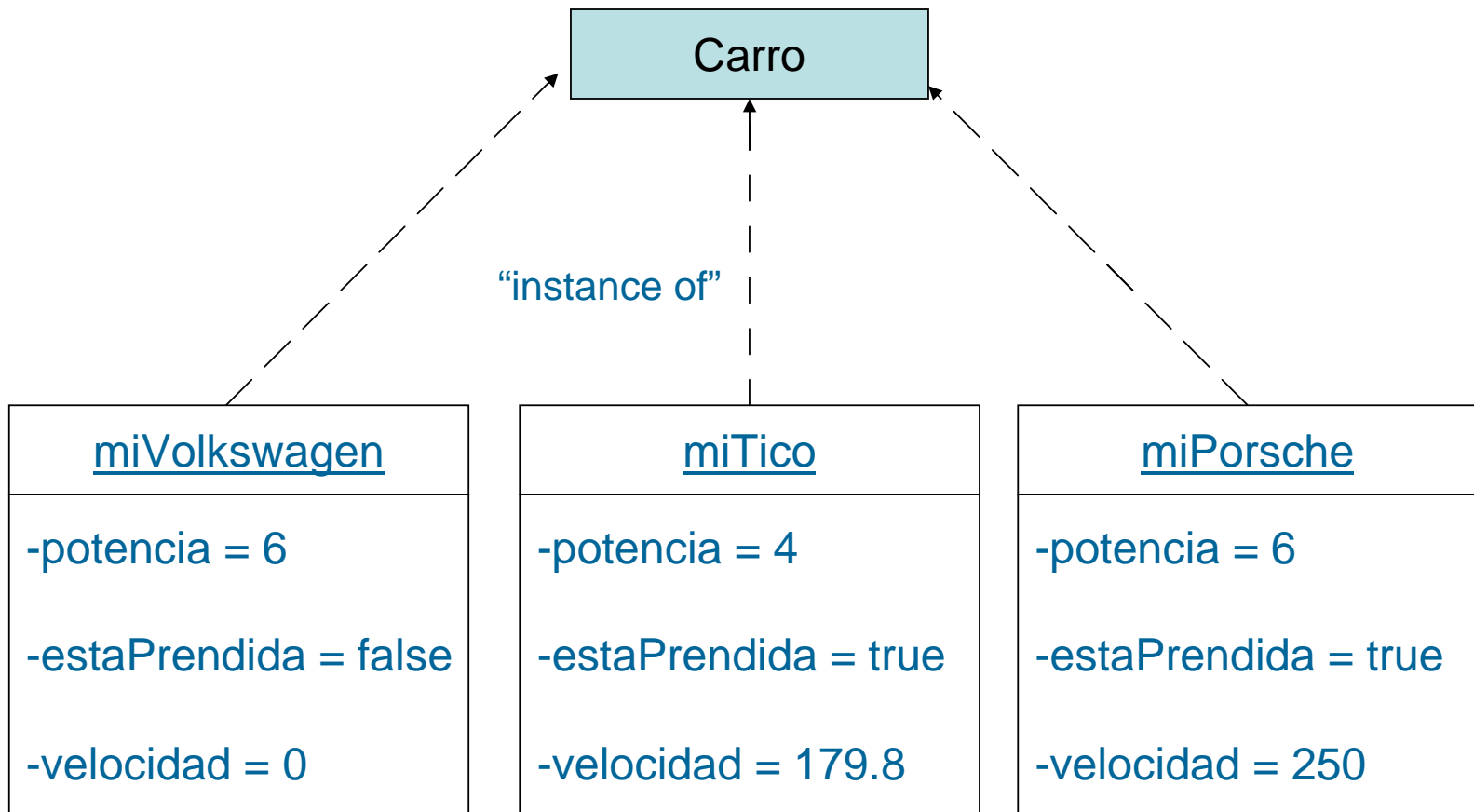
Objeto y notación UML

➤ **miTico es una instancia de la clase *Carro***



Estado de objetos

- Cada objeto que esta una instancia de la clase *Carro* tiene sus propias valores de atributos



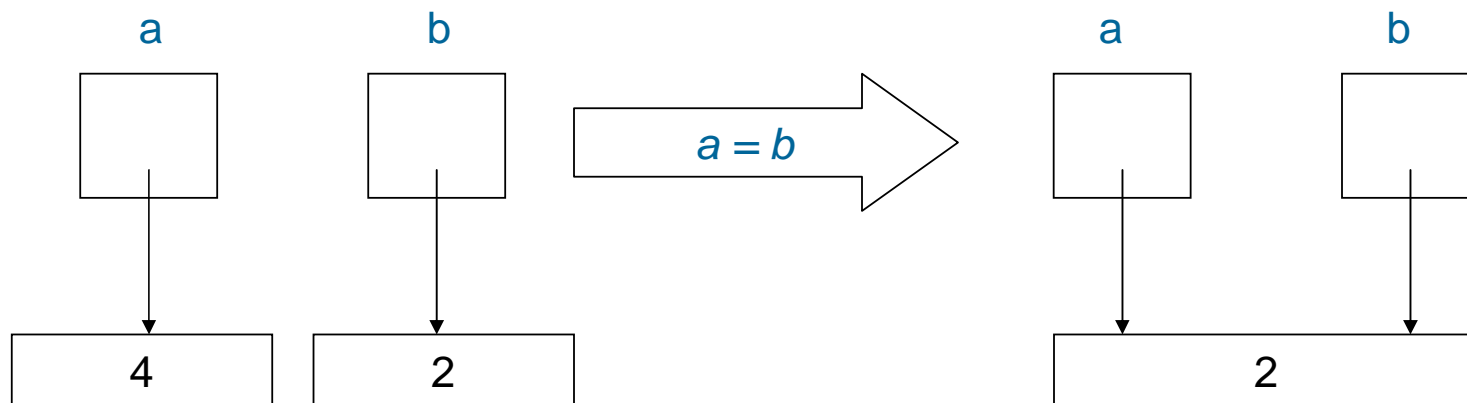
Afectación y comparación

➤ Afectar un objeto

- ❑ « $a = b$ » significa a se vuelve idéntico a b
- ❑ Los dos objetos a et b son idénticos y toda modificación de a implica ella de b

➤ Comparar dos objetos

- ❑ « $a == b$ » devuelve « true » si los dos objetos son idénticos
- ❑ es decir si las referencias son iguales, eso no compara los atributos



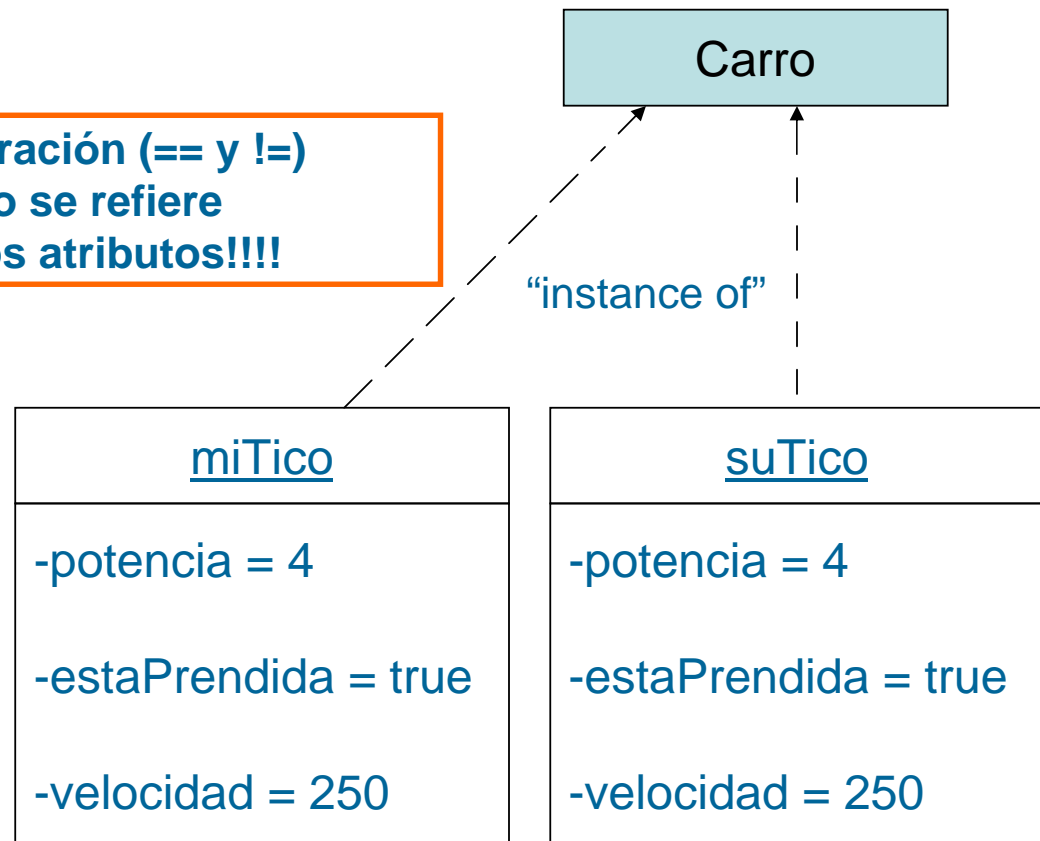
Afectación y comparación

- El objeto **miTico** y **suTico** tienen los mismos atributos (estados idénticos) pero tienen referencias diferentes

□ **miTico != suTico**



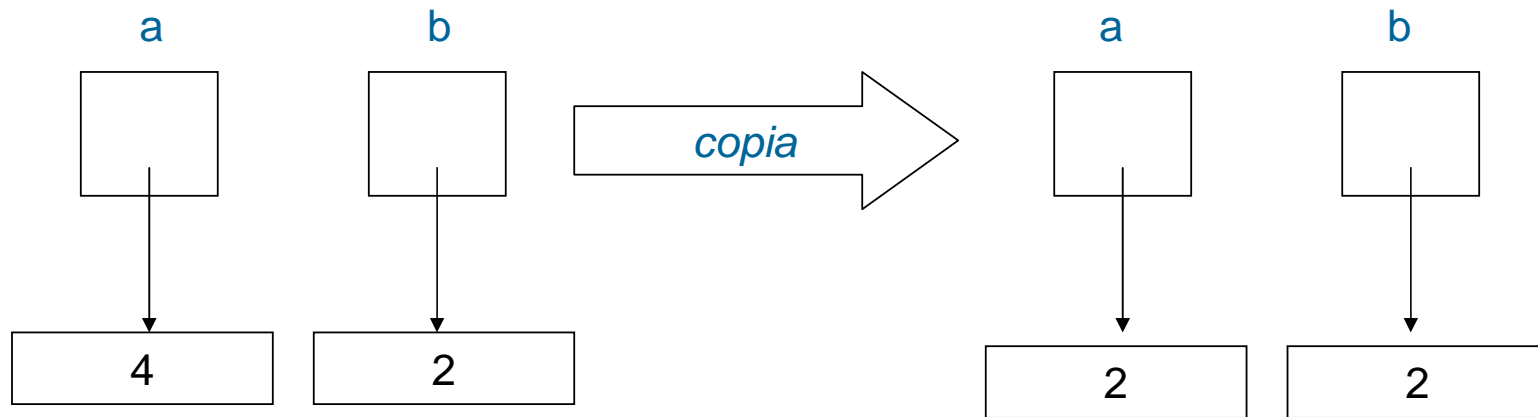
¡La prueba de comparación (== y !=) entre objetos sólo se refiere a la referencia y no los atributos!!!!



Afectación y comparación

➤ Copiar los atributos de un objeto « clone() »

- ❑ Los dos objetos a y b son distintos
- ❑ Toda modificación de a no implica ella de b



➤ Comparar el contenido de objetos : « equals(Object o) »

- ❑ Devolver « true » si los objetos a y b pueden ser considerados como iguales a la vista de sus atributos



Copia y comparación en las partes siguientes

Estructura de objetos

- Un objeto esta constituido de una parte « **estática** » y de una parte « **dinámica** »
- Parte « **estática** »
 - No varía de una instancia clase a otra
 - Permite de activar el objeto
 - Constituida de métodos de la clase
- Parte « **dinámica** »
 - Varía de una instancia clase a otra
 - Varia durante la vida de un objeto
 - Constituida de un ejemplar de cada atributo de la clase

Ciclo de vida de un objeto

➤ Creación

- Uso de un Constructor
- El objeto es creado en memoria y los atributos del objeto son inicializados

➤ Utilización

- Uso de métodos y de atributos (no recomendado)
- Los atributos del objeto pueden ser modificados
- Los atributos (o sus derivados) pueden ser consultados



La utilización de un objeto no construido provoca una excepción de tipo *NullPointerException*

➤ Destrucción y liberación de la memoria cuando :

- Uso (eventual) de un *Pseudo-Destructor*
- Ya no se hace referencia al objeto, el espacio memoria que ocupaba es recuperado

Creación de objetos : proceso

➤ La creación de un objeto a partir de una clase esta llamada una **instanciation**.

➤ El objeto creado es una **instancia** de la clase

➤ Declaración

- Define el nombre y el tipo del objeto
- Un objeto solamente declarado vale « **null** » (palabra reservada del lenguaje)

➤ Creación y asignación de la memoria

- Llama de métodos particulares : los constructores
- La creación reservan la memoria y inicializa los atributos

➤ Vuelta de una referencia sobre el objeto ahora creado

- `miObjeto != null`

miObjeto

null

miObjeto



atributo 1
...
atributo n

Creación de objetos : realización

- La creación de un objeto esta realizada par **new** Constructor(parámetros)
- Existe un constructor por defecto que no tienen parámetro (si ningún otro constructor con parámetro existe)



Los constructores llevan el mismo nombre

```
public class PruebaMiCarro {  
  
    public static void main (String[] argv) {  
  
        // Declaración luego creación  
        Carro miCarro;  
        miCarro = new Carro();  
  
        // Declaración y creación en una única línea  
        Carro miSegundoCarro = new Carro();  
  
    }  
}
```

Declaración

Creación y asignación en memoria

Creación de objetos : realización

➤ Ejemplo : construcción de objetos

```
public class PruebaMiCarro {  
  
    public static void main (String[] argv) {  
  
        // Declaración luego creación  
        Carro miCarro;  
        miCarro = new Carro();  
  
        // Declaración de un segundo carro  
        Carro miCarroCopiado; ←  
        // Atención!! por el momento miCarroCopiado vale null  
  
        // Prueba en las referencias.  
        if (miCarroCopiado == null) {  
  
            // Creación por afectación de otra referencia  
            miCarroCopiado = miCarro ←  
            // miCarroCopiado tiene la misma referencia que miCarro  
        }  
    }  
}
```

Declaración

Afectación
por referencia

El constructor de “Carro”

➤ Hasta ahora

- ❑ Se utilizó el constructor por defecto sin parámetro
- ❑ No se sabe como se construye el “Carro”
- ❑ Los valores de atributos al inicio son indefinidos y idénticos para cada objeto (potencia, etc.)



Los constructores llevan el mismo nombre que la clase y no tienen valor de vuelta

➤ Papel del constructor en Java

- ❑ Efectuar algunas inicializaciones necesarias para el nuevo objeto creado

➤ Toda clase Java tiene al menos un constructor

- ❑ Si una clase no define explícitamente de constructor, un constructor por defecto sin argumentos y que no efectúa ningún inicialización particular esta alegada

El constructor de “Carro”

➤ El constructor de « Carro »

- ❑ Inicializa « velocidad » a cero
- ❑ Inicializa « estaPrendida » a false
- ❑ Inicializa la « potencia » con el valor pasado en parámetro del constructor

```
public class Carro {  
  
    private int potencia;  
    private boolean estaPrendida;  
    private double velocidad;  
  
    public Carro(int p) {  
        potencia = p;  
        estaPrendida = false;  
        velocidad = 0;  
    }  
    ...  
}
```

Constructor
con un parámetro

Construir un “Carro” de 7CV

➤ El constructor de « Carro »

- ❑ Declaración de la variable « miCarro »
- ❑ Creación del objeto con el valor 7 en parámetro del constructor

```
public class PruebaMiCarro {  
    public static void main(String[] argv) {  
        // Declaración luego creación  
        Carro miCarro;  
        miCarro = new Carro(7);  
        Carro miSegundoCarro;  
        // Implicado que existe  
        // explícitamente un constructor : Carro(int)  
        miSegundoCarro = new Carro(); // Error  
    }  
}
```

Declaración

Creación y asignación memoria con Carro(int)

Constructor sin argumentos

➤ Utilidad :

- Cuando se debe crear un objeto sin poder decidir los valores los valores de sus atributos en el momento de la creación
- Sustituye al constructor por defecto que se volvió inutilizable en cuanto se definió a un constructor (con parámetros) en la clase

```
public class Carro {  
  
    private int potencia;  
    private boolean estaPrendida;  
    private double velocidad;  
  
    public Carro() {  
        potencia = 4;  
        estaPrendida = false;  
        velocidad = 0;  
    }  
  
    public Carro(int p) {  
        potencia = p;  
        estaPrendida = false;  
        velocidad = 0;  
    } ...  
}
```

```
public class PruebaMiCarro {  
  
    public static void main (String[] argv) {  
  
        // Declaración luego creación  
        Carro miCarro;  
        miCarro = new Carro(7);  
        Carro miSegundoCarro;  
        miSegundoCarro = new Carro(); // OK  
    }  
}
```

Constructor múltiples

➤ Intereses

- ❑ Posibilidad de inicializar un objeto de varias maneras diferentes
- ❑ Se habla entonces de **sobrecarga** (overloaded)
- ❑ El compilador distingue los constructores en función :
 - ❑ de la posición de argumentos
 - ❑ del numero
 - ❑ del tipo



Cada constructor tiene el mismo nombre (el nombre de la clase)

```
public class Carro {  
    ...  
    public Carro() {  
        potencia = 4; ...  
    }  
  
    public Carro(int p) {  
        potencia = p; ...  
    }  
  
    public Carro(int p, boolean estaPrendida) {  
        ...  
    }  
}
```

Acceso a los atributos

- Para acceder a los datos de un objeto, se utiliza una notación puntada

IdentificaciónObjeto.nombreMetodo

```
public class PruebaMiCarro {  
  
    public static void main (String[ ] argv) {  
  
        // Declaración luego creación  
        Carro v1 = new Carro();  
        Carro v2 = new Carro();  
  
        // Acceso a los atributos en escritura  
        v1.setPotencia(110);  
  
        // Acceso a los atributos en lectura  
        System.out.println("Potencia de v1 = " + v1.getPotencia() );  
    }  
}
```


Envió de mensaje : llamada de métodos

➤ Para « pedir » a un objeto de efectuar un tratamiento es necesario **enviar un mensaje**

➤ Un mensaje esta formado de 3 partes

- ❑ Una referencia permitiendo designar el objeto a quien el mensaje se envía
- ❑ El nombre del método o del atributo a ejecutar
- ❑ Los eventuales parámetros de la método

IdentificaciónObjeto.nombreDeMétodo(« Parámetros eventuales »)

➤ Envío de mensaje similar a una llamada de función

- ❑ El código definido en el método es ejecutado
- ❑ El control se da la vuelta al programa que llama

Envió de mensaje : llamada de métodos



No olvidar los paréntesis para las llamadas a los métodos

```
public class PruebaMiCarro {  
  
    public static void main (String[] argv) {  
  
        // Declaración luego creación  
        Carro miCarro = new Carro();  
  
        // El carro prende  
        miCarro.prende();  
  
        if (miCarro.deCualPotencia() <= 4) {  
            System.out.println("No muy rápido...");  
        }  
  
        // El carro acelere  
        miCarro.acelere (123.5);  
  
    }  
}
```

Carro
- ...
+ deCualPotencia() : entero + prende() + acelere (double) + ...

Envió de un mensaje
al objeto *miCarro*
Llamada de un modificador

Envió de un mensaje
al objeto *miCarro*
Llamada de un selector

Envió de mensaje : paso de parámetros

➤ Un parámetro de un método puede ser

- Una variable de tipo simple
- Una referencia de un objeto caracterizado por cualquiera clase

➤ En Java todo es pasado por valor

- Les parámetros efectivos de un método
- El valor de vuelta de un método (si diferente de « void »)

➤ Los tipos simples

- Su valor es copiada
- Su modificación en el método no implica la del original

➤ Los objetos

- Su referencia es copiada y no los atributos
- Su modificación en el método implica la del original!!!

Envió de mensaje : paso de parámetros

➤ Ejemplo

```
public class PruebaMiCarro {  
  
    public static void main (String[] argv) {  
  
        // Declaración luego creación  
        Carro miCarro = new Carro();  
  
        // Declaración luego creación 2  
        Carro miSegundoCarro = new Carro();  
  
        // Llamada del método compara(Carro) que  
        // devuelve el nombre de diferencia  
        int p = miCarro.compara(miSegundoCarro);  
  
        System.out.println("Numero diferencia :" + p);  
  
    }  
}
```

Adición de un selector con
paso de referencia

Referencia como parámetro

Carro
- ...
+ acelere (double) + compara (Carro) : entero + ...

El objeto “corriente” : this

- El objeto « corriente » es designado con la palabra clave **this**
 - ❑ Permite de designar el objeto en el cual se encuentra
 - ❑ **self** o **current** en otros lenguajes
 - ❑ Designa una referencia particular que es un miembro oculto



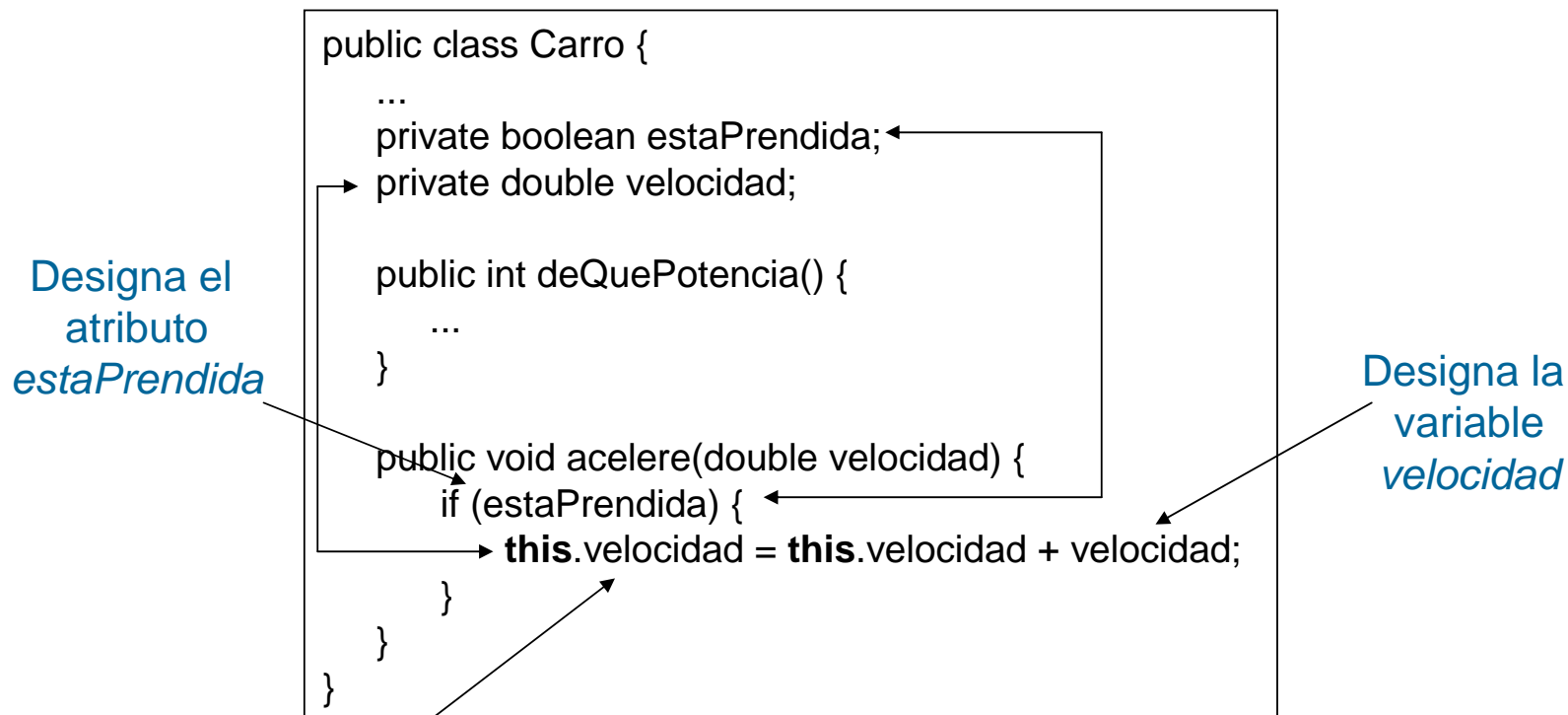
¡No intentar afectar un nuevo valor a this!!!!

`this = ... ; // No pensar a eso`

- Utilidad del objeto « corriente »
 - ❑ Volver explícito el acceso a los propios atributos y métodos definidos en la clase
 - ❑ Pasar en parámetro de un método la referencia del objeto corriente

El objeto “corriente” : atributos y métodos

➤ Designa variables o métodos definidas en una clase



this no es necesario cuando los identificadores de variables no presentan ningún equívoco

El come back de UML

➤ Asociación : los objetos son semánticamente vinculados

□ Ejemplo : un Carro esta manejado por el Conductor

➤ Agregación : ciclo de vida independiente

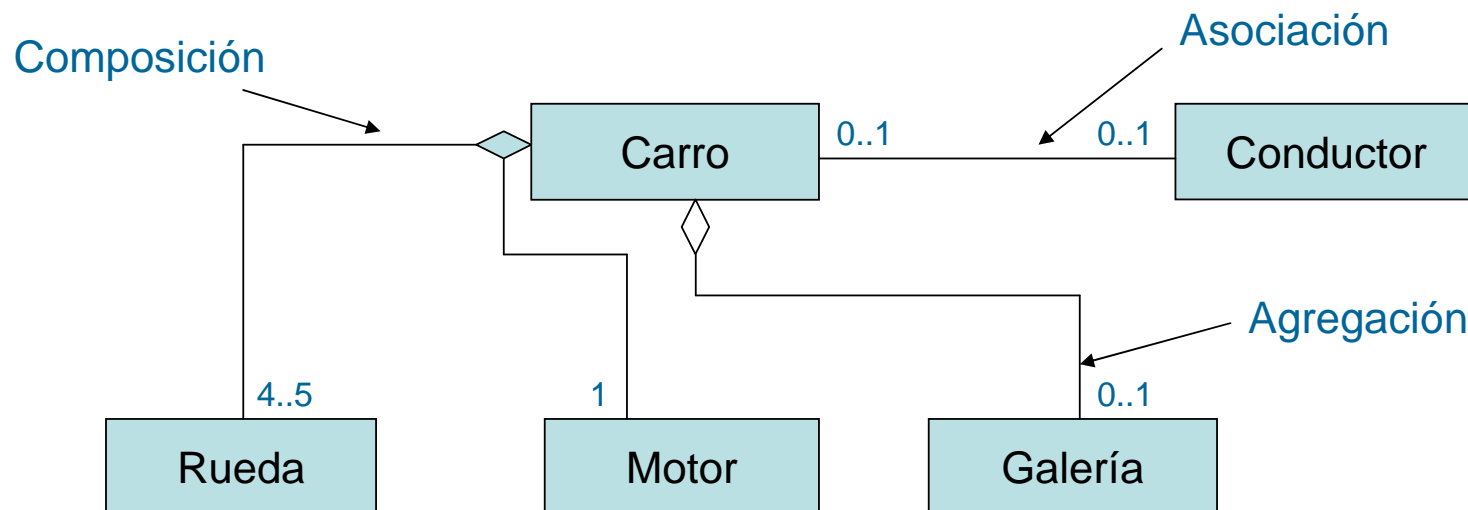
□ Ejemplo : un Carro y una Galería

➤ Composición : ciclo de vida idénticos

□ Ejemplo : un Carro tiene un Motor que dura la vida del Carro

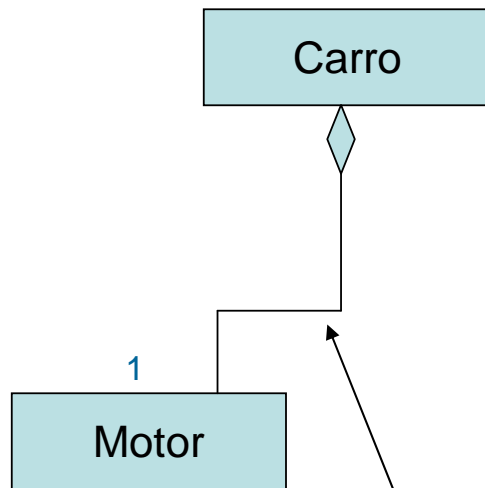


Los rombos se ligan a la clase que contiene



Codificación del asociación : composición

- El objeto de clase *Carro* puede enviar mensajes al objeto de clase *Motor*: Solución 1



A discutir : si el motor de un carro esta muerto, hay la posibilidad de cambiarlo

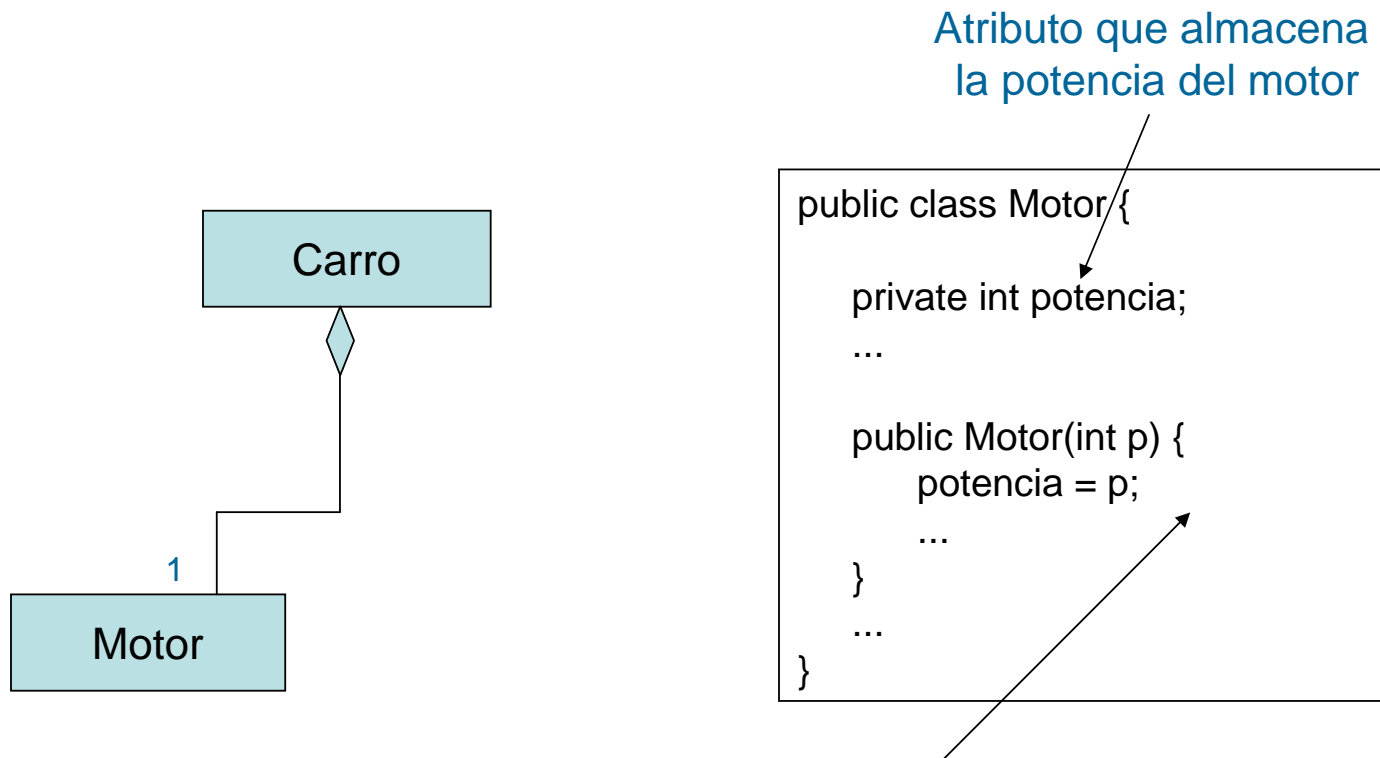
Atributo que almacena la referencia del motor

```
public class Carro{
    private Motor elMotor;
    ...
    public Carro(int p) {
        elMotor = new Motor(p);
        ...
    }
    ...
}
```

Creación del objeto Motor

Codificación del asociación : composición

- El objeto de clase *Motor* no envía mensajes al objeto de clase *Carro* :
Solución 1



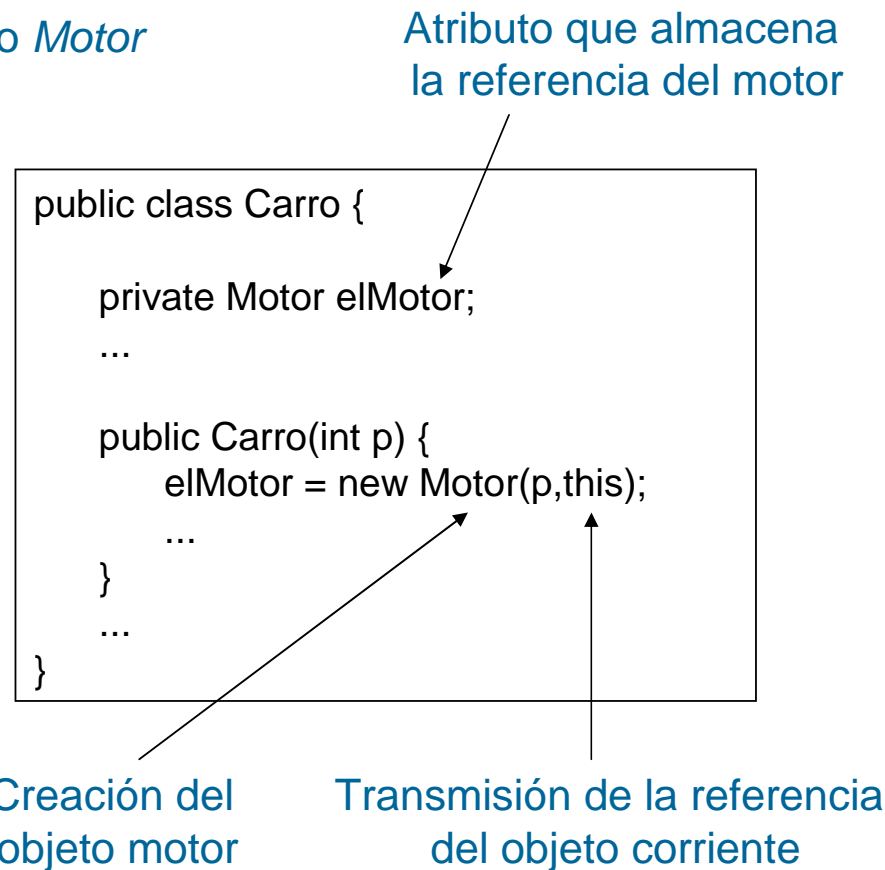
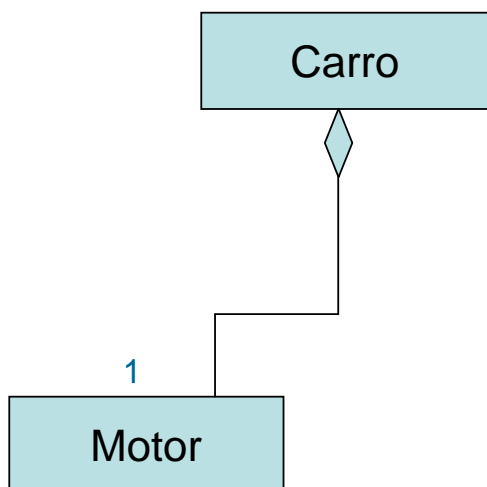
Atributo que almacena la potencia del motor

La potencia esta dado durante la construcción

Codificación del asociación : composición

➤ Puede ser necesario que los dos objetos en composición se intercambian mensajes: solución 2

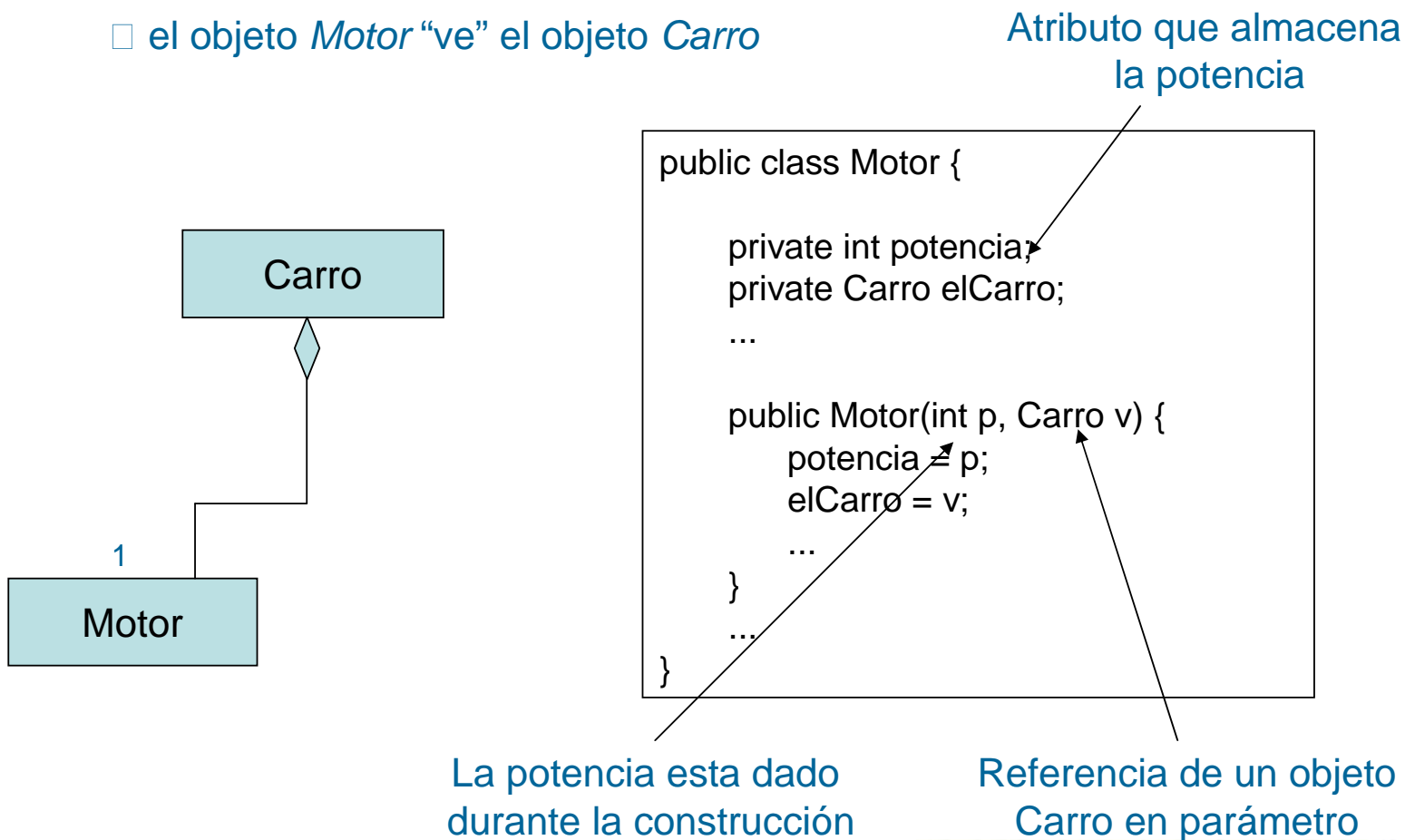
□ el objeto *Carro* “ve” el objeto *Motor*



Codificación del asociación : composición

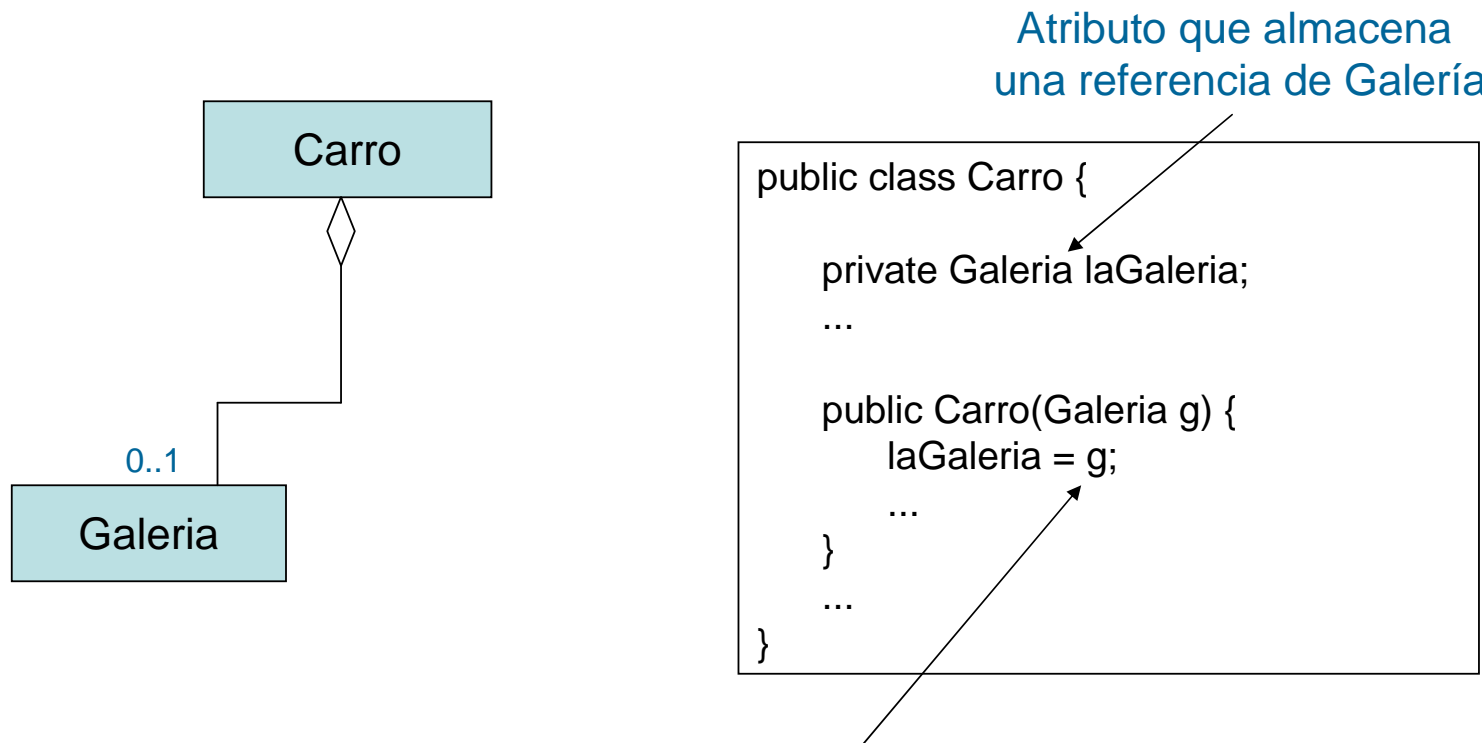
➤ Puede ser necesario que los dos objetos en composición se intercambian mensajes: solución 2

□ el objeto *Motor* “ve” el objeto *Carro*



Codificación del asociación : agregación

- El objeto de clase *Galería* no envía mensaje al objeto de clase *Carro*



Un objeto *Galería* esta transmitido al momento de la construcción del *Carro*

Destrucción y garbage collector

- La destrucción de los objetos se hace de manera implícita
- El garbage collector se pone activa
 - Automáticamente
 - Así más ninguna variable referencia el objeto
 - Si el bloque en el cual se definía se termina
 - Si el objeto fue afectado con “nulo”
 - Manualmente
 - A petición explícita del programador por la instrucción “System.gc ()”
- Un pseudo destructor “*protected void finalize ()*” puede ser definido explícitamente por el programador
 - él esta llamado exactamente antes de la liberación de la memoria por la máquina virtual, pero no se sabe cuando. Conclusión: ¡no muy seguro!!!!



Preferir definir un método y de alegarlo antes que todo objeto ya no esté hecho referencia: *destruido ()*

Destrucción y garbage collector

```
public class Carro {  
  
    private boolean estaPrendida;  
  
    protected void finalize() {  
        estaPrendida = false;  
        System.out.println("Motor parado");  
    }  
}
```



Para ser seguro que finalice se ejecuta bien, se debe absolutamente llamar explícitamente *System.gc()*

Fuerza el programa por terminarse

```
public class PruebaMiCarro {  
  
    public static void main(String[] argv) {  
        // Declaración luego creación de miCarro  
        Carro miCarro = new Carro();  
        miCarro.prende();  
        // miCarro no me sirve ya a nada  
        miCarro = null;  
  
        // Llamada explicita del garbage collector  
        System.gc();  
  
        // Fin del programa  
        System.exit(0);  
        System.out.println("Mensaje no visible");  
    }  
}
```

Output - CursoJavaSE (run-single)

```
compile-single:  
run-single:  
Motor parado  
BUILD SUCCESSFUL (total time: 1 second)
```

Gestión de objetos

- **Indicar su tipo y su sitio memoria: `System.out.println ()`**

```
System.out.println(miCarro) // Carro@119c082
```

- **Recuperación su tipo: método “`Class getClass ()`”**

```
miCarro.getClass(); // Devuelve un objeto de tipo Class  
Class classCarro = miCarro.getClass(); // Class es una clase!!!
```

- **Probar su tipo: operador “`instanceof`” o palabra clave “`class`”**

```
if (miCarro instanceof Carro) {...} // Es verdad
```

o

```
if (miCarro.getClass() == Carro.class) {...} // Es verdad también
```

Sobrecarga

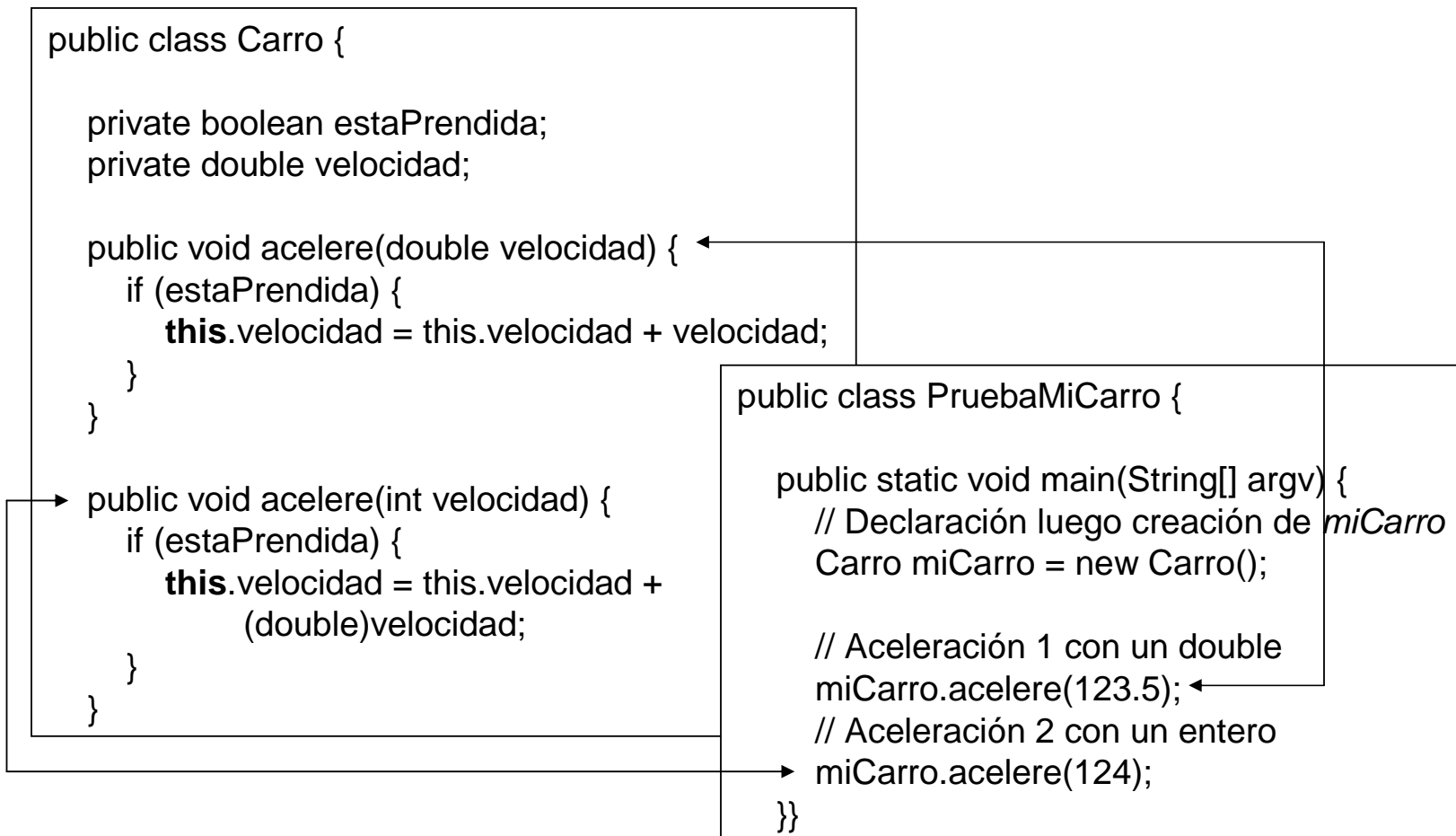
- La sobrecarga (*overloading*) no se limita a los constructores, es posible también para cualquier método
- Posibilidad de definir de los métodos que poseen el mismo nombre pero de los que los argumentos difieren
- Cuando se alega un método sobrecargado el compilador selecciona automáticamente el método cuyo número es el tipo de los argumentos corresponden al número y al tipo de los parámetros pasados en la llamada del método



Métodos sobrecargados pueden tener tipos de vuelta diferentes a condición que tengan argumentos diferentes

Sobrecarga

➤ Ejemplo



Constructores múltiples

➤ Llamada explícita de un constructor de la clase dentro de otro constructor

- Debe hacerse como primera instrucción del constructor
- Utiliza la palabra clave “this(parámetros efectivos)”

➤ Ejemplo

- Implantación del constructor sin parámetro de “Carro” a partir del constructor con parámetros

```
public class Carro {
    ...
    public Carro() {
        ← this(7, new Galeria());
    }
    public Carro(int p) {
        ← this(p, new Galeria());
    }
    public Carro(int p, Galeria g) {
        potencia = p;
        motor = new Motor(potencia);
        galería = g;
        ...
    }
}
```

Encapsulación

- Posibilidad de acceder a los atributos de una clase Java pero no recomendada ya que contraria al principio de encapsulación
 - Los datos deben ser protegidos y accesibles para el exterior por selectores
- Posibilidad de actuar sobre la visibilidad de los miembros (atributos y métodos) de una clase frente a otras clases
- Varios niveles de visibilidad pueden definirse precediendo de un modificador la declaración de un atributo, método o constructor
 - Private
 - Public
 - Protected ← A ver en la parte siguiente

Encapsulación : visibilidad de miembros de una clase

	+ public	- private
Clase	La clase puede ser utilizada por cualquiera clase	Utilizable solamente por las clases definidas dentro de una otra clase. Una clase privada es utilizable por su clase que engloba
Atributo	Atributo accesible por todas partes donde su clase es accesible. No se recomienda de un punto de vista encapsulación	Atributo limitado a la clase donde se hace la declaración
Método	Método accesible por todas partes donde su clase es accesible.	Método accesible dentro de la definición de la clase

Encapsulación

➤ Ejemplo



Un método privado no puede ya alegarse fuera del código de la clase donde se define

```
public class Carro {  
    private int potencia;  
    ...  
    public void prende() {  
        ...  
    }  
    private void hacerCombustión() {  
        ...  
    }  
}
```

```
public class PruebaMiCarro {  
    public static void main (String[] argv) {  
        // Declaración luego creación de miCarro  
        Carro miCarro = new Carro();  
  
        // Prendida de miCarro  
        miCarro.prende();  
  
        miCarro.hacerCombustión(); // Error  
    }  
}
```

Las cadenas de caracteres “String”

➤ Son objetos tratados como tipos simples...

➤ Inicialización

```
String miCadena = "Hola!"; // Eso parece a un tipo simple
```

➤ Longitud

```
miCadena.length(); // Con las paréntesis porque eso es un método
```

➤ Comparación

```
miCadena.equals("Hola!"); // Devuelve true
```

➤ Concatenación

```
String prueba = "prue" + "ba";  
String prueba = "prue".concat("ba");
```



**Atención a la comparación
de cadenas de caracteres.
miCadena == "toto";
¡Comparación sobre las referencias!!**

Las cadenas modificables “StringBuffer”

- Son modificables por inserción, adiciones, conversiones, etc.

- Se obtiene un « StringBuffer » con sus constructores

```
StringBuffer mCM = new StringBuffer(int length);  
StringBuffer mCM = new StringBuffer(String str);
```

- Se puede transformarles en cadenas normales « String » por :

```
String s = mCM.toString();
```

- Se puede añadir cualquier cosa (sobrecarga) por :

```
mCM.append(...); // String, int, long, float, double
```

- Se puede insertarlos cualquier cosa (sobrecarga) por :

```
mCM.insert(int offset, ...); // String, int, long, float, double
```

Las cadenas descomponibles “StringTokenizer”

- Permiten la descomposición en palabras o elementos siguiendo un delimitador

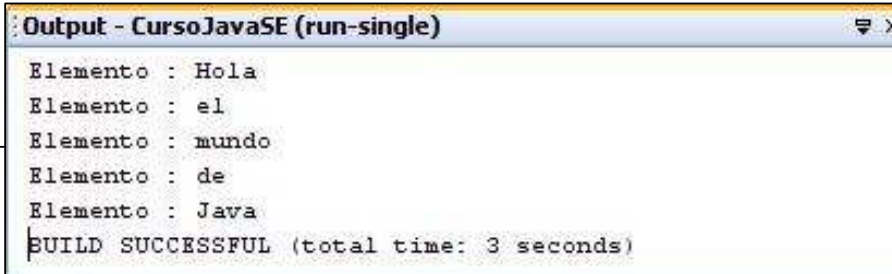
this is a test => this
 is
 a
 test

- Se obtiene un « StringTokenizer » con sus constructores

```
StringTokenizer mCM = new StringTokenize(String str); // Delimitador = espacio  
StringTokenizer rMCM = new StringTokenizer(String str, String delim);
```

- Un ejemplo :

```
java.util.StringTokenizer st =  
    new java.util.StringTokenizer("Hola,  
    el mundo|de|Java", "|");  
while(st.hasMoreElements()) {  
    System.out.println("Elemento : " + st.nextElement());  
}
```



```
Output - CursoJavaSE (run-single)  
Elemento : Hola  
Elemento : el  
Elemento : mundo  
Elemento : de  
Elemento : Java  
BUILD SUCCESSFUL (total time: 3 seconds)
```


Variables de clase

- Puede ser útil de definir para una clase de atributos independientemente de las instancias : numero de Carro creadas
- Utilización des variables de clase comparables a las « variables globales »
- Uso de las **variables de clase**
 - ❑ Variables de las cuales no existe más que un único ejemplar asociado a su clase de Definición
 - ❑ Variables existen independientemente del número de instancias de la clase que han sido creados
 - ❑ Variables utilizables aunque ninguna instancia de la clase existe

Variables de clase

- Son definidas como los atributos pero con la palabra clave **static**

```
public static int numeroCarroCreados;
```



Atención a la encapsulación.
Es peligroso dejar esta variable
de clase en public.

- Para acceder, es necesario utilizar no un identificador sino el nombre de la clase

```
Carro.numeroCarroCreados = 3;
```



**No está prohibido utilizar una
variable de clase como un atributo
(por medio de un identificador)
pero muy desaconsejado**

Constantes de clase

➤ Uso

- ❑ Son constantes vinculadas a una clase
- ❑ Son escritas en MAYUSCULAS



Una constante de clase es generalmente siempre visible

➤ Además, son definidas con la palabra clave final

```
public class Galeria {  
    public static final int MASA_MAX = 150;  
}
```

➤ Para acceder, es necesario utilizar no un identificador de objeto sino el nombre de la clase (igual variables de clase)

```
if (miCarro.getWeightLimite() <= Galerie.MASA_MAX) {...}
```

Variables y constantes de clase : ejemplo

➤ Ejemplo

```
public class Carro {  
  
    public static final int PESO_TOTAL_MAX = 3500;  
    private int peso;  
    public static int numeroCarroCreados;  
    ...  
  
    public Carro(int peso, ...) {  
        this.peso = peso;  
        ...  
    }  
}
```

Peligroso porque posibilidad
de modificación exterior

Utilización de variables
y constantes de clase
por el nombre de la clase
Carro

```
public class PruebaMiCarro {  
  
    public static void main (String[] argv) {  
        // Declaración luego creación de miCarro  
        Carro miCarro = new Carro(2500);  
        ...  
  
        System.out.println("Peso maxi:" +  
        Carro.PESOTOTAL_MAX);  
        System.out.println(Carro.numeroCarroCreados);  
        ...  
    }  
}
```

Métodos de clase

➤ Uso

- ❑ Esto son métodos que no se interesan por un objeto particular
- ❑ Útil para cálculos intermedios internos a una clase
- ❑ Útil también para devolver el valor de una variable de clase en visibilidad *private*

➤ Se definen como los métodos de instancias, pero con la palabra clave **static**

```
public static double velocidadMaxTolerada() {  
    return velocidadMaxTolerada*1.10;  
}
```

➤ Para acceder, es necesario utilizar no un identificador de objeto sino el nombre de la clase (igual variables de clase)

Carro.velocidadMaxTolerada()

Métodos de clase : ejemplo

➤ Ejemplo

```
public class Carro {  
    private static int numeroCarroCreados;  
    ...  
    public static int getNumeroCarroCreados(){  
        return Carro.numeroCarroCreados;  
    }  
}
```

Declaración de una variable de clase privada. Respecto de principios de encapsulación.

Declaración de un método de clase para acceder al valor de la variable de clase.

```
public class PruebaMiCarro {  
    public static void main (String[] argv) {  
        // Declaración luego creación de miCarro  
        Carro miCarro = new Carro(2500);  
        ...  
        System.out.println("Numero Instancias :" +  
            Carro.getNumeroCarroCreados());    }  
}
```

Métodos de clase : error clásica

➤ Ejemplo

```
public class Carro {  
    private Galeria laGaleria;  
    ...  
    public Carro(Galeria g) {  
        laGaleria = g;  
        ...  
    }  
    public static boolean isGaleriaInstall() {  
        return (laGaleria != null)  
    }  
}
```

Declaración de un objeto
Galería non estático

Error : utilización
de un atributo non
estático en un zona
estática



No se puede utilizar
variables de instancia en un
método de clase!!!!

Métodos de clase

➤ Recuerdo : cada uno de los tipos simples (int, double, boolean, char) tiene un alter-ego objeto que dispone de métodos de conversión

➤ Por ejemplo la clase *Integer* « encapsula » el tipo **int**

□ Constructor a partir de un int o de una cadena de caracteres

```
public Integer(int value);  
public Integer(String s);
```

□ Disponibilidad de métodos que permiten la conversión en tipo simple

```
Integer valorObjeto = new Integer(123);  
int valorPrimitivo = valorObjeto.intValue();  
○  
int valorPrimitivo = valueObjeto; (AutoBoxing)
```



Atención a los errores de conversión.
Vuelta de una excepción.
A ver en la última parte del curso

□ Métodos de clase muy útiles que permiten a partir de una cadena de caracteres de transformar en tipo simple o tipo object

```
String miValorCadena = new String("12313");  
int miValorPrimitivo = Integer.parseInt(miValorCadena);
```


Las tablas en Java : aplicación objetos

① Declaración

```
Carro[] miTabla;
```

② Dimensión

```
miTabla = new Carro[3];
```

③ Inicialización

```
miTabla[0] = new Carro(5);  
miTabla[1] = new Carro(7);  
miTabla[2] = new Carro(8);
```

o ① ② y ③

```
Carro[ ] miTab = {  
    new Carro(5),  
    new Carro(7),  
    new Carro(8)  
};
```



```
for (int i = 0; i < miTabla.length; i++) {  
    System.out.println(miTabla[i].prende());  
}
```



Programación Orientada Objeto Aplicación al lenguaje JAVA

Herencia



Institut de recherche
pour le développement

Jérémie HABASQUE – 2007
mailto:jeremie_habasque@yahoo.fr

Definición y intereses

➤ Herencia

- Técnica oferta por los lenguajes de programación para construir una clase a partir de una (o varias) otra clase compartiendo sus atributos y operaciones

➤ Intereses

- **Especialización, enriquecimiento** : una nueva clase reutiliza los atributos y las operaciones de una clase añadiendo y/o de las operaciones particulares a la nueva clase
- **Redefinición** : una nueva clase redefine los atributos y operaciones de una clase de manera a cambiar el sentido y/o el comportamiento para el caso particular definido por la nueva clase
- **Reutilización** : evita de reescribir código existente y a veces no se poseen las fuentes de la clase que debe heredarse

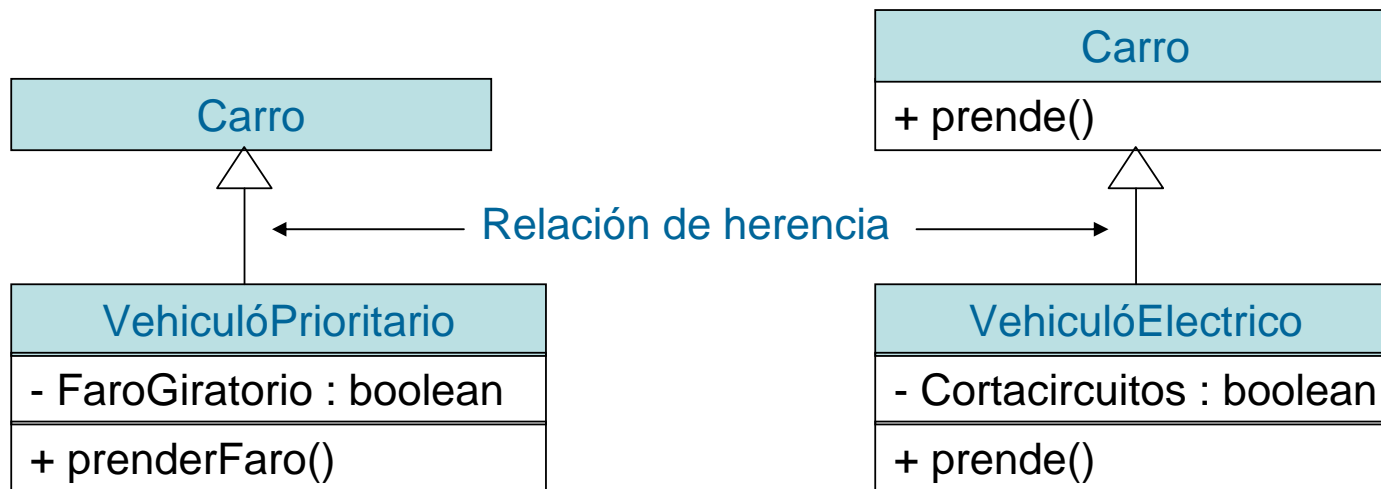
Especialización de la clase “Carro”

➤ Un vehiculó prioritario es un carro con giró faro

- ❑ Un vehiculó prioritario responde a los mismos mensajes que el carro
- ❑ Se puede encender el giró faro de un vehiculó prioritario

➤ Un carro eléctrico es un carro cuya operación de prendida es diferente

- ❑ Un carro eléctrico responde a los mismos mensajes que el carro
- ❑ Se puede prender un carro eléctrico activando un cortacircuitos



Clases y sub clases

➤ Un objeto de la clase *VehículoPrioritario* o *CarroElectrico* es también un objeto de la clase *Carro* entonces dispone de todos los atributos y operaciones de la clase *Carro*

VehiculóPrioritario	
Herede de Carro	- FaroGiratorio : boolean
	+ prenderFaro()
	- potencia : entero
	- estaPrendida : boolean
	- velocidad : float
	+ deQuePotencia() : entero
	+ prende()
	+ acelere(float)

VehiculóElectrico	
Herede de Carro	- CortaCircuitos : boolean
	+ prender ()
	- potencia : entero
	- estaPrendida : boolean
	- velocidad : float
	+ deQuePotencia() : entero
	+ prende()
	+ acelere(float)

Clases y sub clases : terminología

➤ Definiciones

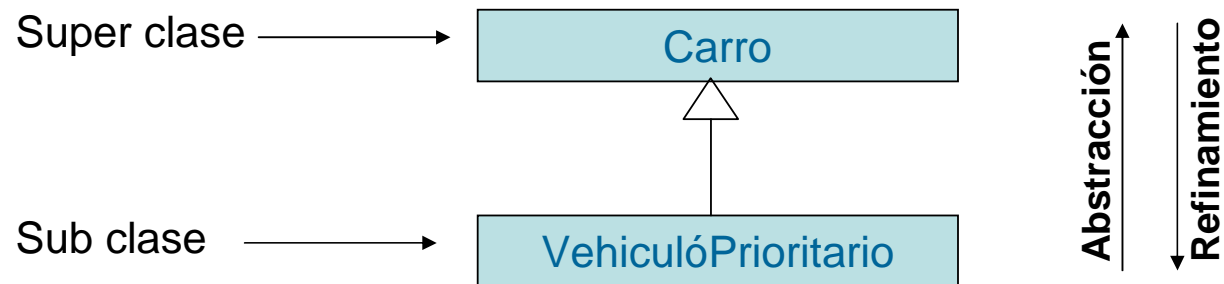
- La clase *VehiculoPrioritario* **herede** de la clase *Carro*
- *Carro* es la **clase madre** y *VehiculoPrioritario* la **clase hija**
- *Carro* es la **super-clase** de la clase *VehiculoPrioritario*
- *VehiculoPrioritario* es una **sub-clase** de *Carro*

➤ Atención

- Un objeto de la clase *VehiculoPrioritario* o *CarroElectrico* es obligatoriamente un objeto de la clase *Carro*
- Un objeto de la clase *Carro* no es obligatoriamente un objeto de la clase *VehiculoPrioritario* o *CarroElectrico*

Generalización y especialización

- La generalización expresa una relación « **es-uno** » entre una clase y su super-clase

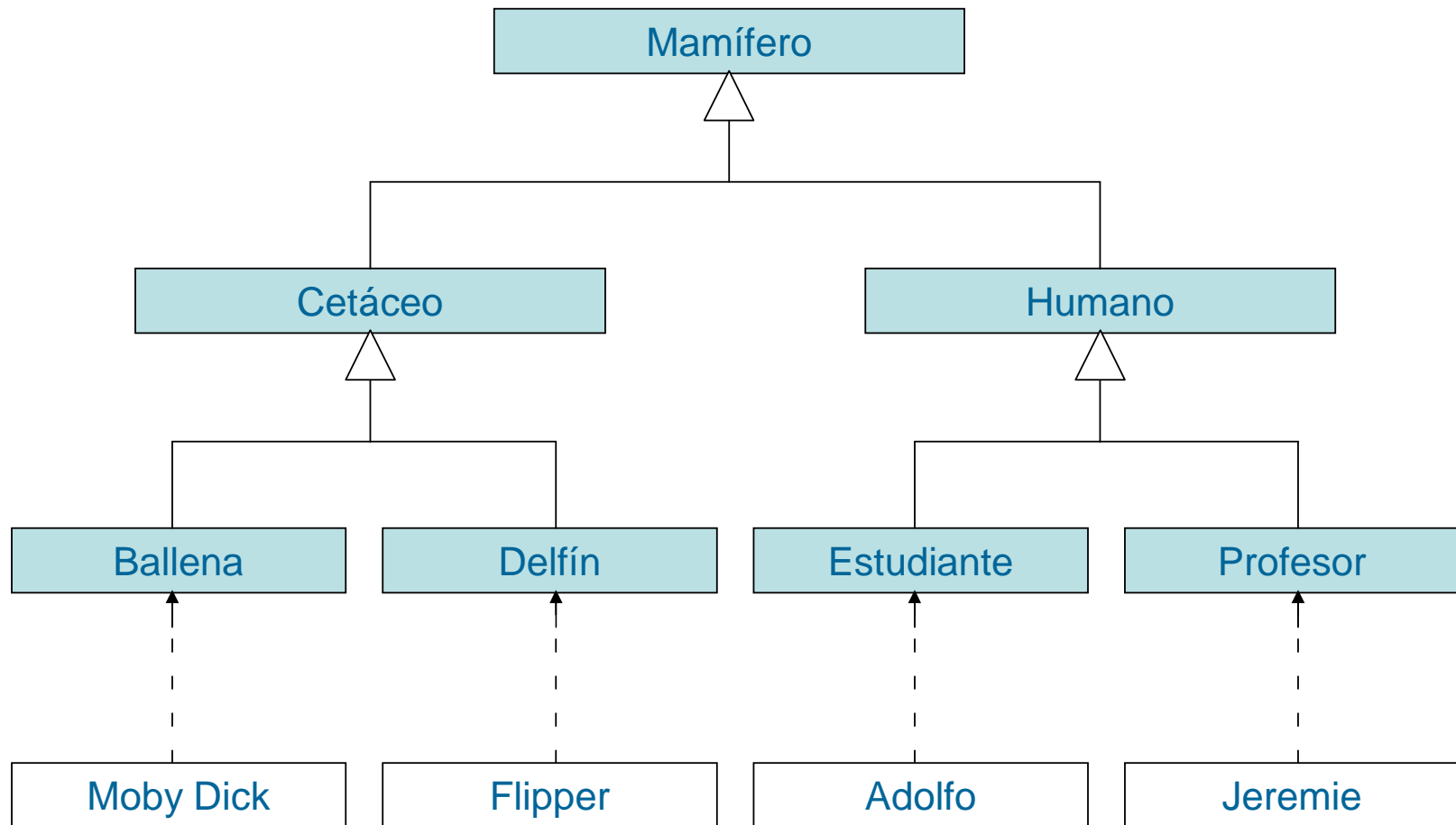


VehiculoPrioritario es un Carro

- La herencia permite :
 - de **generalizar** en el sentido abstracción
 - de **especializar** en el sentido refinamiento

Ejemplo de herencia

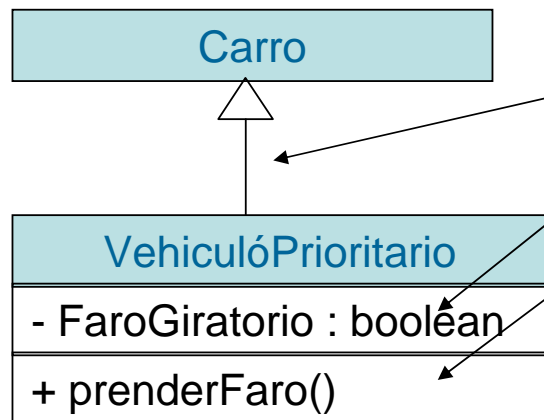
➤ Ejemplo



Herencia y Java

➤ Herencia simple

- Una clase puede heredar solamente de una otra clase
- El algunos lenguajes (ex : C++) posibilidad de herencia múltiple
- Utilización de la palabra clave **extends** después el nombre de la clase

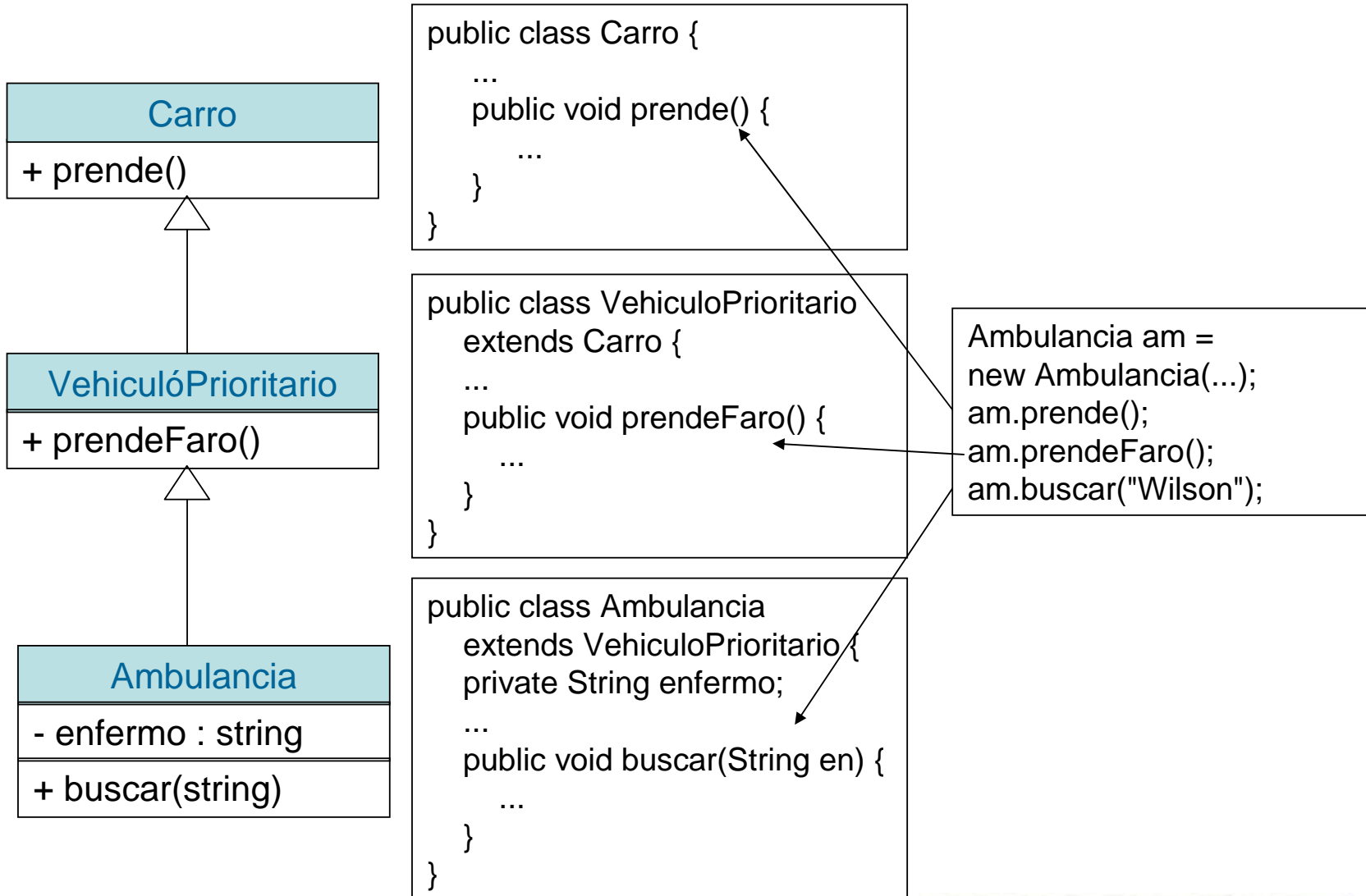


```
public class VehiculoPrioritario extends Carro {
    private boolean faroGiratorio;
    ...
    public void prenderFaro() {
        faroGiratorio = true;
    }
    ...
}
```



No intentar de heredar de varias clases (extends *Carro, Camioneta,...*) eso no funciona

Herencia a varios niveles



Sobrecarga y redefinición

➤ La herencia

- Una sub-clase puede añadir nuevos atributos y/o métodos a los de los que hereda (sobrecarga es una parte)
- Una sub-clase puede redefinir (redefinición) los métodos cuyos ella hereda y proporcionar implementaciones específicas para ellos

➤ **Recuerdo de la *sobrecarga* : posibilidad de definir métodos que tienen el mismo nombre pero cuyos argumentos (parámetros y valor de vuelta) difieren**



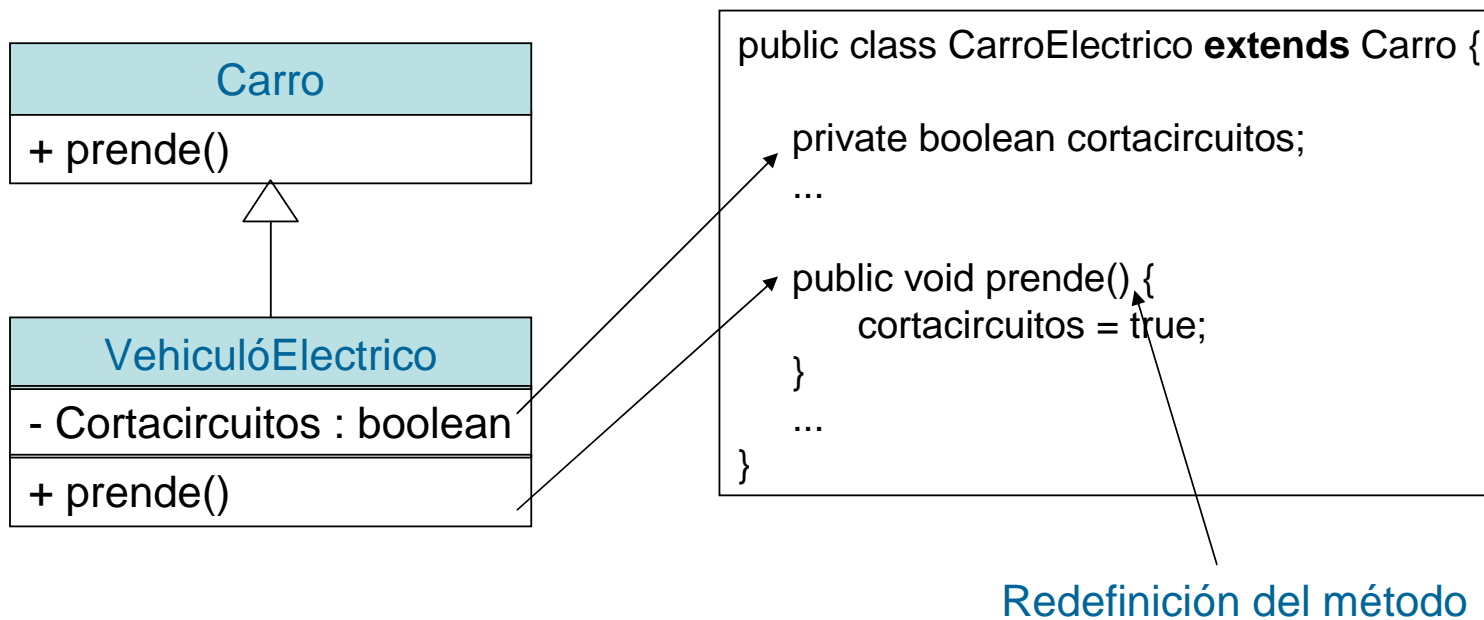
Métodos sobrecargados pueden tener diferentes tipos de vuelta a condición que los métodos tienen argumentos diferentes

➤ ***Redefinición (overriding)* : cuando la sub-clase define un método cuyo el nombre, los parámetros y el tipo de vuelta son idénticos**

Sobrecarga y redefinición

➤ Un carro eléctrico es un carro cuyo la operación de prender es diferente

- Un carro eléctrico responde a los mismos mensajes que el *Carro*
- Se prende un carro eléctrico activando un cortacircuitos



Sobrecarga y redefinición

```
public class Carro {  
    ...  
    public void prende() {  
        ...  
    }  
}
```



**No confundir sobrecarga y redefinición.
En el caso de la sobrecarga, la sub clase
añade métodos mientras que la redefinición
“especializa” de los métodos existentes**

Redefinición

Sobrecarga

```
public class CarroElectrico  
    extends Carro {  
    public void prende() {  
        ...  
    }  
}
```

CarroElectrico tiene « a lo sumo » un método de menos que *VehiculoPrioritario*

```
public class VehiculoPrioritario  
    extends Carro {  
    public void prende(int codigo) {  
        ...  
    }  
}
```

CarroElectrico tiene « a lo sumo » un método más que *VehiculoPrioritario*

Redefinición con reutilización

➤ Interés

- ❑ La redefinición de un método « aplaste » el código del método heredada
- ❑ Posibilidad de reutilizar el código del método heredado por la palabra clave **super**
- ❑ **super** permite así la designación explicita del instancia de una clase cuya el tipo es el de la clase madre
- ❑ Acceso a los atributos y métodos redefinidos por la clase corriente pero que queremos utilizar

super.nombreSuperClaseMetodoLlamado(...);

➤ Ejemplo del Carro : los limites a solucionar

- ❑ La llamada al método *prende()* de *CarroElectrico* no modifica que el atributo Cortacircuitos

Redefinición con reutilización

➤ Ejemplo

```
public class Carro {  
    private boolean estaPrendida;  
    ...  
    public void prende() {  
        estaPrendida = true;  
    }  
}
```

Actualización del atributo *estaPrendida*

```
public class CarroElectrico extends Carro {  
    private boolean Cortacircuitos;  
    ...  
    public void prende() {  
        cortacircuitos = true;  
        super.prende();  
    }  
}
```

```
public class PruebaMiCarro {  
    public static void main (String[] argv) {  
        // Declaración luego creación  
        VehiculoElectrico carroEl =  
            new VehiculoElectrico(...);  
        carroEl.prende();  
    }  
}
```

Envió de un mensaje por llamada de por *prende*



Aquí la posición de *super* no tiene importancia

Uso de constructores : continuación

- Posibilidad como los métodos de reutilizar el código de los constructores de la super-clase
- Llamada explícita de un constructor de la clase madre a dentro de un constructor de la clase hija



La llamada al constructor de la superclase debe hacerse absolutamente en primera instrucción

- Utiliza la palabra clave **super**

super(parámetros del constructor);

- Llamada implícita de un constructor de la clase madre es efectuada cuando no existe llamada explícita. Java inserte implícitamente la llamada **super()**

Uso de constructores : continuación

➤ Ejemplo

```
public class Carro {  
    ...  
    public Carro() {  
        this(7, new Galeria());  
    }  
  
    public Carro(int p) {  
        this(p, new Galeria());  
    }  
  
    public Carro(int p, Galeria g) {  
        potencia = p;  
        motor = new Motor(potencia);  
        galería = g;  
        ...  
    }  
}
```



La llamada al constructor de la superclase debe hacerse absolutamente en primera instrucción

Implantación del constructor de CarroPrioritario a partir de Carro

```
public class CarroPrioritario extends Carro {  
  
    private boolean faroGiratorio;  
  
    public CarroPrioritario(int p, Galeria g) {  
        super(p, null);  
        this.faro = false;  
    }  
}
```

Uso de constructores : continuación

➤ Ejemplo : armadura de constructores

```
public class ClaseA {  
    → public ClaseA() {  
        System.out.println("Clase A");  
    }  
}
```

```
public class ClaseB extends ClaseA{  
    private String mensaje;  
    → public ClaseB(String mensaje) {  
        super(); // Llamada implicita  
        System.out.println("Clase B");  
        System.out.println(mensaje);  
    }  
}
```

```
public class ClaseC extends ClaseB{  
    → public ClaseC(String inicio) {  
        super("Mensaje resultante de C " + inicio);  
        System.out.println("Clase C");  
        System.out.println("Fin");  
    }  
}
```

```
public class ClasePrueba {  
    public static void main(String[] argv) {  
        new ClaseC("Mensaje del main");  
    }  
}
```

Output - CursoJavaSE (run-single)

```
run-single:  
Clase A  
Clase B  
Mensaje resultante de C Mensaje del main  
Clase C  
Fin  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Uso de constructores : continuación

➤ **Recuerdo : si una clase no define explícitamente un constructor, entonces ella tiene un constructor por defecto**

- ❑ Sin parámetro
- ❑ Que no hace nada
- ❑ Inútil si un otro constructor esta definido explícitamente

```
public class A {  
    public void visualizarInformación() {  
        System.out.println("Informaciones...");  
    }  
}
```

```
public A() {  
    super();  
}
```

```
public class B {  
    private String plnfo;  
    public B(String plnfo) {  
        this.plnfo = plnfo;  
    }  
}
```

```
super();
```

```
public class Prueba {  
    public static void main (String[ ] argv) {  
        new B("Mensaje del main");  
    }  
}
```

Uso de constructores : continuación

➤ Ejemplo

```
public class Carro {  
    ...  
    public Carro(int p) {  
        this(p, new Galeria());  
    }  
  
    public Carro(int p, Galeria g) {  
        potencia = p;  
        motor = new Motor(potencia);  
        galería = g;  
        ...  
    }  
}
```

Constructores explícitos
desactivación del constructor
por defecto



**Error : no existe en *Carro*
constructor sin parámetro**

```
public class CarroPrioritario extends Carro {  
  
    private boolean faroGiratorio;  
  
    public CarroPrioritario(int p, Galeria g) {  
        this.faroGiratorio = false;  
    }  
}}
```

La clase **Object** : el misterio resuelto

➤ La clase **Object** es la clase más arriba de nivel en la jerarquía de herencia

- Toda clase otro que **Object** posee una superclase
- Toda clase hereda directamente o indirectamente de la clase **Object**
- Una clase que no define cláusula **extends** hereda de la clase **Object**

```
public class Carro extends Object{  
    ...  
    public Carro(int p, Galeria g) {  
        potencia = p;  
        motor = new Motor(potencia);  
        galería = g;  
        ...  
    }  
    ...  
}
```

Object
+ Class getClass() + String toString() + boolean equals(Object) + int hashCode() ...



No es necesario de escribir explícitamente extends Object

La clase Object : el misterio resuelto

```
public class Carro {  
    ...  
    public Carro(int p) {  
        this(p, new Galeria());  
    }  
}
```

```
public class Prueba {  
    public static void main (String[] argv) {  
        Carro miCarro = new Carro(3);  
        System.out.println(miCarro);  
    }  
}
```

```
public String toString() {  
    return (this.getClass().getName() +  
            "@" + this.hashCode());  
}
```

```
Output - CursoJavaSE (run-single)  
run-single:  
ClassObject.Carro@19821f  
BUILD SUCCESSFUL (total time: 0 seconds)
```

```
public class Carro {  
    ...  
    public Carro(int p) {  
        this(p, new Galeria());  
    }  
  
    public String toString(){  
        return(« Potencia:" + p);  
    }  
}
```

```
public class Prueba {  
    public static void main (String[] argv) {  
        Carro miCarro = new Carro(3);  
        System.out.println(miCarro);  
    }  
}
```

`.In(miCarro.toString());`

```
Output - CursoJavaSE (run-single)  
run-single:  
Potencia:3  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Redefinición del método *String toString()*

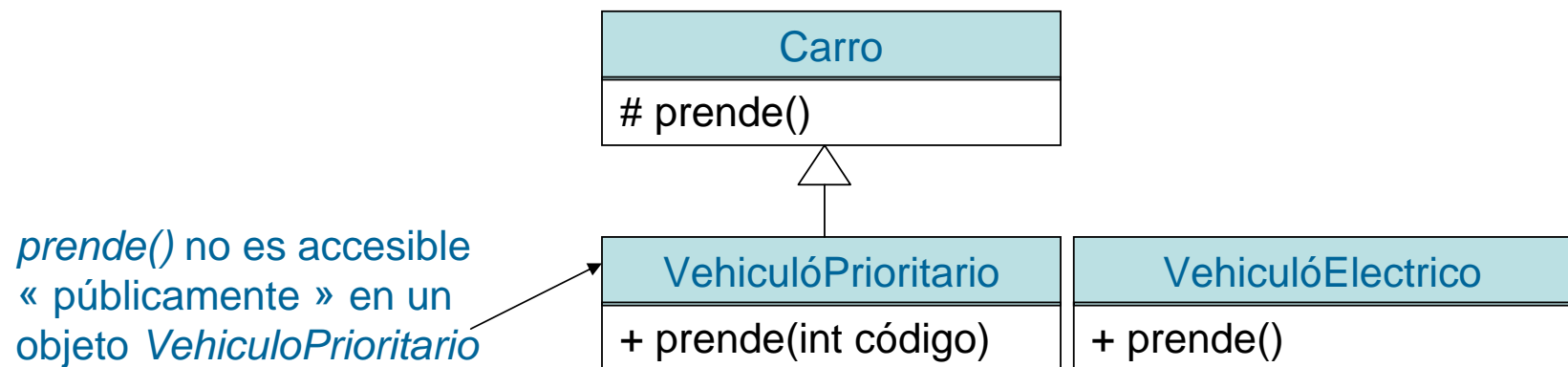
Derechos de acceso a los atributos y métodos

➤ Ejemplo del Carro : los limites a solucionar

- ❑ El método *prende()* es disponible en la clase *VehiculoPrioritario*
Es decir que se puede prender sin dar el código !!!
- ❑ Solución : proteger el método *prende()* de la clase Carro

➤ Realización

- ❑ Utilización de la palabra clave **protected** ante la definición de los métodos y atributos
- ❑ Los miembros son accesibles en la clase donde esta definido, en todas sus sub clases



Derechos de acceso a los atributos y métodos

➤ Ejemplo

```
public class Carro {  
  
    private boolean estaPrendida;  
    ...  
    protected void prende() {  
        estaPrendida = true;  
    }  
}
```

```
public class CarroPrioritario  
    extends Carro {  
  
    private int codigoCarro;  
  
    public void prende(int codigo) {  
        if (codigoCarro == codigo) {  
            super.prende();  
        }  
    }  
}
```

```
public class PruebaMiCarro {  
  
    public static void main (String[] argv) {  
        // Declaración luego creación de miCarro  
        VehiculoElectrico vElectrico = new VehiculoElectrico(...);  
        vElectrico.prende(); // Llama prende de VehiculoElectrico  
  
        VehiculoPrioritario bombero = new VehiculoPrioritario(...);  
        bombero.prende(1234); // Llama prende de CarroPrioritario  
        bombero.prende(); // Error porque prende no es public  
    }  
}
```


Métodos y clases finales

➤ Definición

- Utilización de la palabra-clave **final**
- Método : prohibir una eventual redefinición de un método

```
public final void prende();
```

- Clase : prohibir toda especialización o herencia de la clase concernida

```
public final class CarroElectrico extends Carro {  
    ...  
}
```



La clase *String* por ejemplo es final



Programación Orientada Objeto Aplicación al lenguaje JAVA

Herencia y polimorfismo



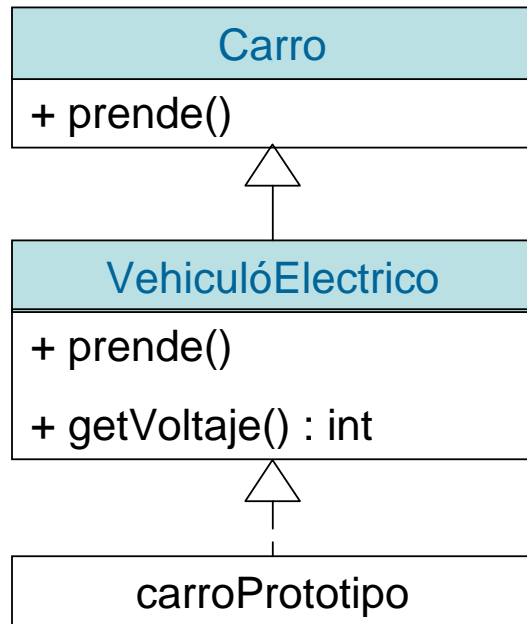
Institut de recherche
pour le développement

Jérémie HABASQUE – 2007
mailto:jeremie_habasque@yahoo.fr

Definición del polimorfismo

➤ Definición

- Un lenguaje orientado objeto esta dicho polimorfico, si ofrece la posibilidad de poder recibir un objeto como instancia de clases variadas, según las necesidades
- **Una clase B que hereda de la clase A puede ser vista como un sub-tipo del tipo definido por la clase A**



➤ Recuerdo

- carroPrototipo es una instancia de la clase *VehiculoElectrico*

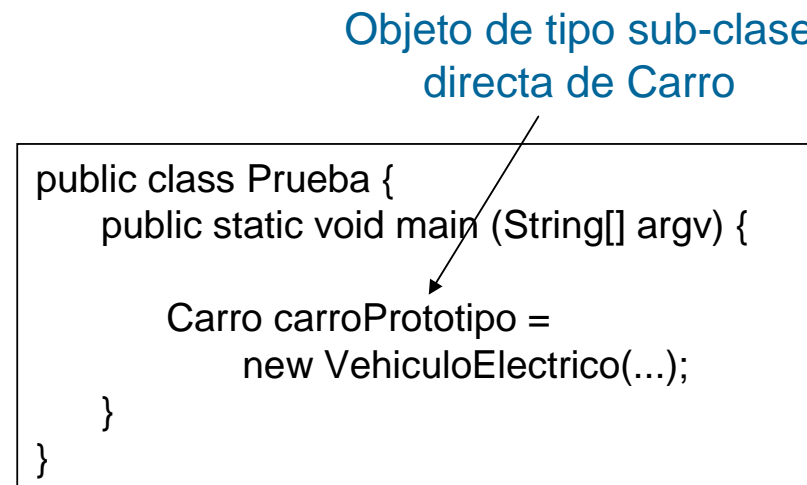
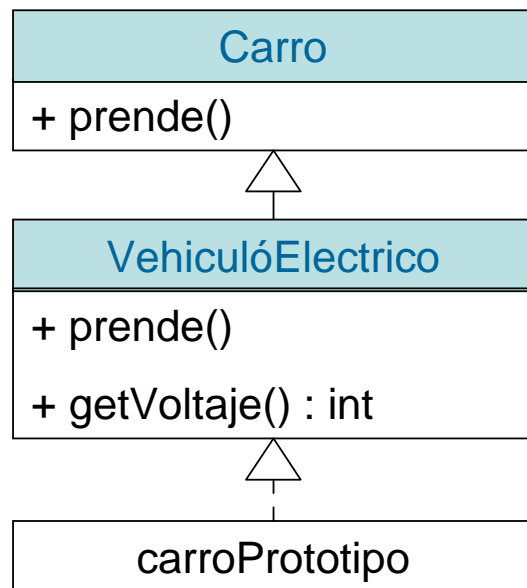
➤ Pero también

- carroPrototipo es una instancia de la clase *Carro*

Polimorfismo y Java : upcasting

➤ Java es polimorfo

- A una referencia declarada de la clase *Carro*, es posible de afectar un valor que es una referencia hacia un objeto de la clase *CarroElectrico*
- Se habla de **upcasting**
- Mas generalmente a una referencia de un tipo dado, o sea A, es posible de afectar un valor que corresponde a una referencia hacia un objeto cuyo el tipo efectivo es cualquiera sub clase directa o indirecta de A



Polimorfismo y Java : upcasting

➤ A la compilación

- Cuando un objeto con « upcast », es visto por el compilador como un objeto de tipo de la referencia utilizada para designarle
- Sus funcionalidades se limitan entonces a las propuestas por la clase del tipo de la referencia

```
public class Prueba {  
    public static void main (String[] argv) {  
  
        // Declaración et creación de un objeto Carro  
        Carro carroPrototipo = new CarroElectrico(...);  
  
        // Utilización de un método de la clase Carro  
        carroPrototipo.prende();  
  
        // Utilización de un método de la clase CarroElectrico  
        System.out.println(carroPrototipo.getVoltaje()); // Error  
    }  
}
```

El método *getVoltaje()*
no es disponible
en la clase Carro!!!



Examinar el tipo de la referencia

Polimorfismo y Java : upcasting

```
public class Prueba {
    public static void main (String[] argv) {

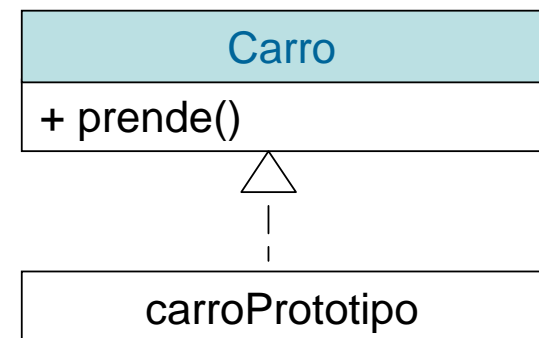
        // Declaración et creación de un objeto Carro
        Carro carroPrototipo = new CarroElectrico(...);

        // Utilización de un método de la clase Carro
        carroPrototipo.prende(); ✓

        // Utilización de un método de la clase CarroElectrico
        System.out.println(carroPrototipo.getVoltaje());

    }
}
```

Observación : Cual será el código efectivamente ejecutado cuando el mensaje `prende()` esta enviado a `carroPrototipo` ??

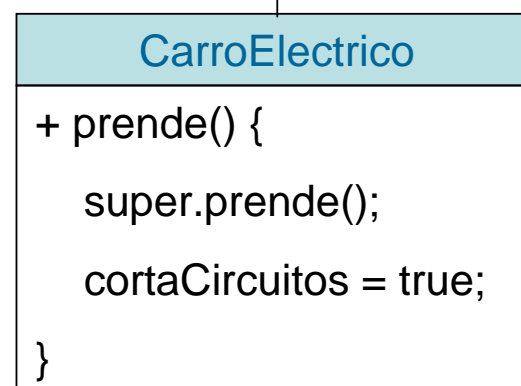
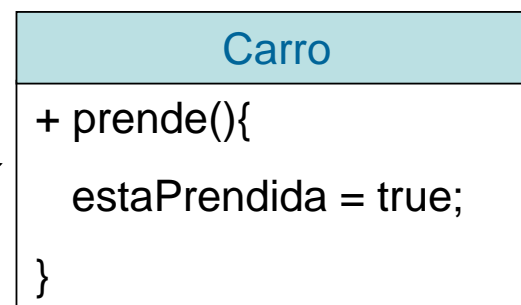


Polimorfismo y Java : vinculo dinámico

```
public class Prueba {  
    public static void main (String[] argv) {  
        Carro carroPrototipo = new CarroEléctrico(...);  
  
        carroPrototipo.prende();  
    }  
}
```

El objeto carroPrototipo
inicializa los atributos
de la clase CarroElectrico

carroPrototipo.prende()



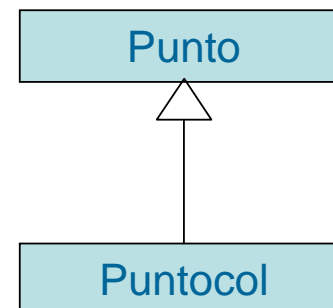
Observación : Es el método *prende()* de *CarroElectrico* que esta llamada. Entonces, el método llama (por super) el método de la superclase

Polimorfismo y Java : vinculo dinámico

```
public class Punto {
    private int x,y;
    public Punto(int x, int y) { this.x = x; this.y = y; }
    public void desplazar(int dx, int dy) { x += dx; y+=dy; }
    public void visualización() { System.out.println("Estoy en "+ x + " " + y);}
}
```

```
public class Puntocol extends Punto {
    private byte color;
    public Puntocol(int x, int y, byte color) {
        super(x,y);
        this.color = color;
    }
    public void visualización() {
        super.visualización();
        System.out.println("y mi color es : " + color);
    }}
```

```
public class Prueba {
    public static void main (String[] argv) {
        Punto p = new Punto(23,45);
        p.visualización();
        Puntocol pc = new Puntocol(5,5,(byte)12);
        p = pc;
        p.visualización();
        p = new Punto(12,45);
        p.visualización();
    }
}
```

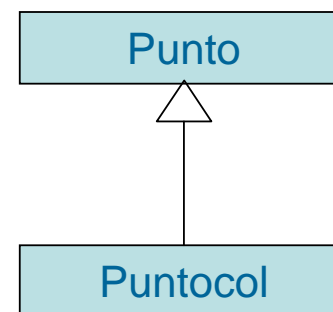


Polimorfismo y Java : vinculo dinámico

```
public class Punto {
    private int x,y;
    public Punto(int x, int y) { this.x = x; this.y = y; }
    public void desplazar(int dx, int dy) { x += dx; y+=dy; }
    public void visualización() {
        this.identifica();
        System.out.println("Estoy en "+ x + " " + y);}
    }
    public void identifica() {System.out.println("Soy un punto");}
}
```

```
public class Puntocol extends Punto {
    private byte color;
    public Puntocol(int x, int y, byte color) {...}
    public void visualización() {
        super.visualización();
        System.out.println("y mi color es : " + color);
    }
    public void identifica() {System.out.println("Soy un punto colorado");}
}
```

```
public class Prueba {
    public static void main (String[] argv) {
        Punto p = new Punto(23,45);
        p.visualización();
        Puntocol pc = new Puntocol(5,5,(byte)12);
        p = pc;
        p.visualización();
        p = new Punto(12,45);
        p.visualización();    }}
```



Polimorfismo y Java : vinculo dinámico

➤ Al ejecución

- Cuando un método de un objeto esta accedado a través de una referencia con « upcast », es el método como esta definido al nivel de la clase efectiva del objeto que esta invocada y ejecutada
- El método a ejecutar es determinado al ejecución y no a la compilación
- Se habla de relación tardía, vinculo dinámico, dynamic binding, late-binding o run-time binding

Polimorfismo y Java : balance

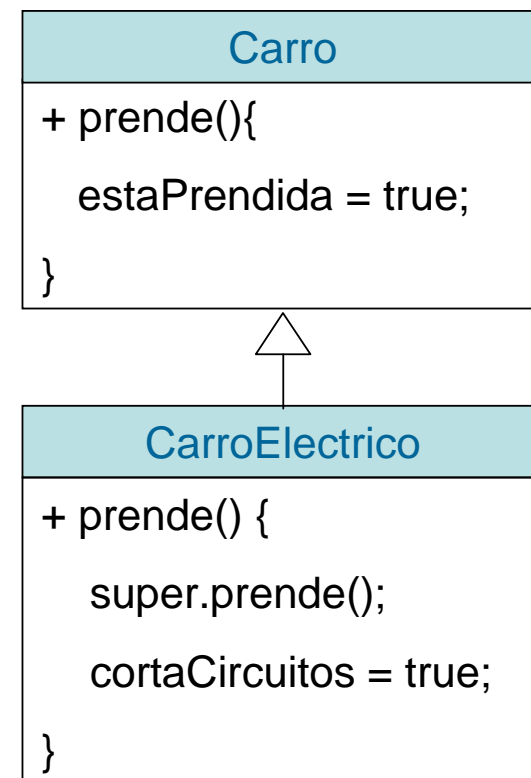
```
public class Prueba {  
    public static void main (String[] argv) {  
        Carro carroPrototipo = new CarroElectrico(...);  
        carroPrototipo.prende();  
    }  
}
```

➤ Upcasting (compilación)

- ❑ Una variable *carroPrototipo* esta declarada como una referencia hacia un objeto de la clase *Carro*
- ❑ Un objeto de la clase *CarroElectrico* esta creado
- ❑ Para el compilador *carroPrototipo* es una referencia de un objeto de la clase *Carro*, y impide de acceder a los métodos especificas a *CarroElectrico*

➤ Vinculo dinámico (ejecución)

- ❑ Una variable *carroPrototipo* es una referencia hacia un objeto de la clase *CarroElectrico*



Polimorfismo: OK, pero por qué hacer?

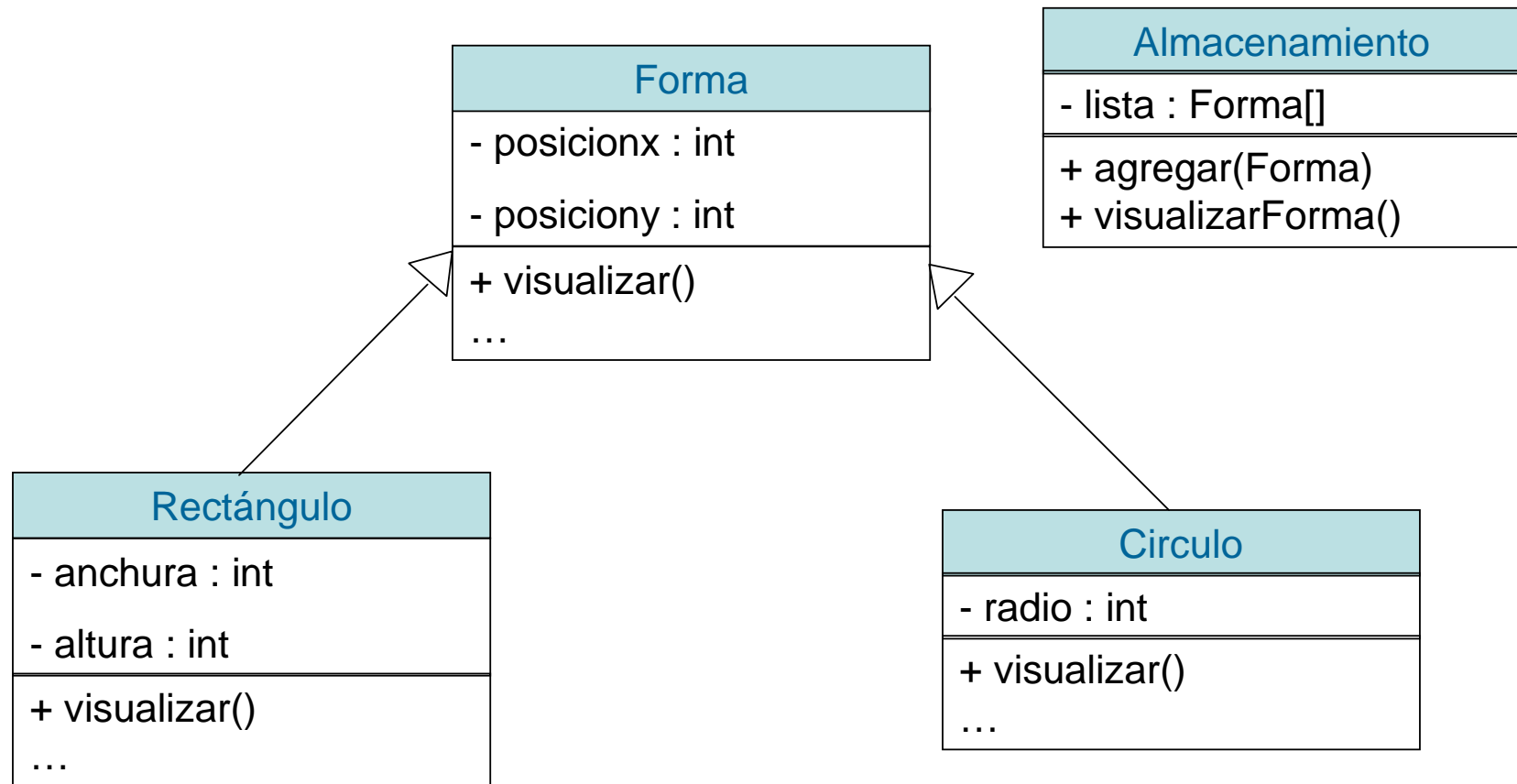
➤ Ventajas ...

- No es necesario de distinguir diferentes casos en función de la clase de los objetos
- El polimorfismo constituye la tercera característica esencial de un lenguaje orientado objeto después la abstracción de datos (encapsulación) y la herencia
- Una facilidad mas grande de evolución del código. Posibilidad de definir nuevas funcionalidades heredando de nuevos tipos de datos a partir de una clase de base común sin necesitar de modificar el código que manipula la clase de base
- Desarrollo **mas rápido**
- Mas grande **simplicidad** y **mejor organización** del código
- Programas mas fácilmente **extensibles**
- Mantenimiento del código **mas fácil**

Polimorfismo: un ejemplo típico

➤ Ejemplo sobre geometría

- Almacenar unas Forma de cualquier tipo (Rectángulo o Circulo) luego visualizarlos



Polimorfismo: un ejemplo típico

```
public class Almacenamiento {
    private Forma[] lista;
    private int tamaño;
    private int i;

    public Almacenamiento(int tamaño) {
        this.tamaño = tamaño;
        lista = new Forma[this.tamaño];
        i = 0;
    }

    public void agregar(Forma f) {
        if (i < tamaño) {
            lista[i] = f;
            i++;
        }
    }

    public void visualizarForma() {
        for (int i = 0; i < tamaño; i++) {
            lista[i].visualizar();
        }
    }
}
```



Si un nuevo tipo de Forma
esta definido, el código de la clase
Almacén no esta modificado

```
public class Prueba {
    public static void main (String[] argv) {
        Almacenamiento miAlmacen = new Almacenamiento(10);
        miAlmacen.agregar(new Circulo(...));
        miAlmacen.agregar(new Rectangulo(...));

        Rectangulo miRect = new Rectangulo(...);
        Forma tuRect = new Rectangulo(...);
        miAlmacen.agregar(miRect);
        miAlmacen.agregar(tuRect);
    }
}
```

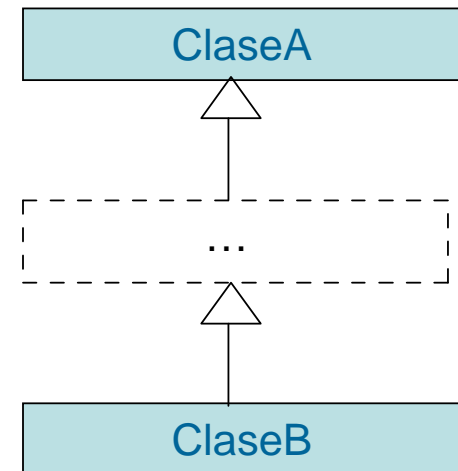
Polimorfismo: downcasting

➤ Interés

- ❑ Fuerza un objeto “por liberar” las funcionalidades ocultas por el dominio
- ❑ Se habla de conversión de tipo explícita (cast). Ya en vista de para los tipos primitivos

```
ClaseA miObj = ...  
ClaseB a = (ClaseB) miObj
```

- ❑ Para que el “cast” funciona, es necesario que a la ejecución el tipo efectivo de *miObj* sea “compatible” con el tipo ClaseB
- ❑ Compatible : se puede probar la compatibilidad por la palabra clave **instanceof**



obj instanceof ClaseB

Devuelve true o false

Polimorfismo: downcasting

➤ Ejemplo

```
public class Prueba {
    public static void main (String[] argv) {
        Forma miForma = new Rectangulo(...);
        //No puedo utilizar los métodos de la clase Rectángulo

        //Declaración de un objeto de tipo Rectángulo
        Rectángulo miRect;
        if (miForma instanceof Rectángulo) {
            miRect = (Rectangulo)miForma;
            //Utilización posible de métodos específicas de Rectángulo
        }
    }
}
```



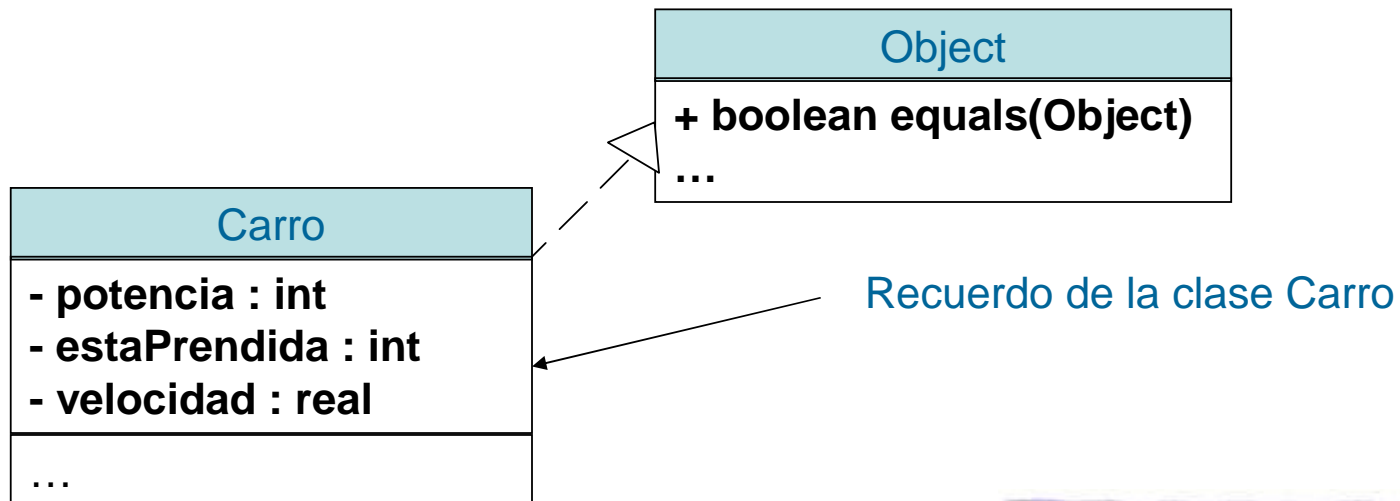
Atención si la compatibilidad es falsa y si el cast esta efectuado una excepción de tipo *ClassCastException* se aumenta

Realización de la conversión del objeto de tipo Forma en objeto de tipo Rectángulo

El método “equals()”

➤ Dos posibilidades para comparar un objeto de sus clases

- ❑ Crear un método « `public boolean comparar(MiClase c) {...}` » que compara los atributos
- ❑ Redefinir el método « `public boolean equals(Object o)` » para guardar la compatibilidad con las otras clases de Java.
 - ❑ Re-implementar el método « `public boolean equals(Object o)` » comparando los atributos (utilizando una conversión de tipo explícito)



El método “equals()”

```
public class Carro extends Object {  
    public boolean equals(Object o) {  
        if (!o instanceof Carro) {  
            return false;  
        }  
  
        Carro miCarro = (Carro)o;  
        return this.potencia == miCarro.potencia && this.estaPrendida ==  
            miCarro.estaPrendida && this.velocidad == miCarro.velocidad;  
    }  
    ...  
}
```

Redefinición del método equals
de la clase Object

```
public class Prueba {  
    public static void main (String[] argv) {  
        Carro miCarro = new Carro(...);  
        CarroElectrico miCarroElec = new CarroElectrico(...);  
  
        miCarro.equals(miCarroElec); --> TRUE  
    }  
}
```

Mismos valores
de argumentos

**Atención : la igualdad de referencia ==
comprueba si las referencias son las mismas,
eso no compara los atributos**



Clases abstractas : intereses

➤ No se conoce siempre el comportamiento por defecto de una operación común a varias subclases

- Ejemplo: techo de un convertible. Se sabe que todas las convertibles pueden guardar su techo, pero el mecanismo es diferente de convertible
- Solución: se puede declarar el método “abstracta” en la clase madre y no darle implantación por defecto

➤ Método abstracta y consecuencias : 3 reglas a saber

- Si una sola de los métodos de una clase es abstracta, luego la clase se vuelve también abstracta
- No se puede instanciar una clase abstracta porque al menos una de sus métodos no tiene implementación
- Todas las clases hijas heredando de la clase madre abstracta deben implementar todos los métodos abstractas o sino son también abstractas

Clases abstractas y Java

➤ Se utiliza la palabra clave **abstract** para especificar abstracta una clase

➤ Una clase abstracta se declara así :

```
public abstract class NombreMiClase {  
    ...  
}
```

➤ Un método abstracto se declara así :

```
public abstract void miMetodo(...);
```



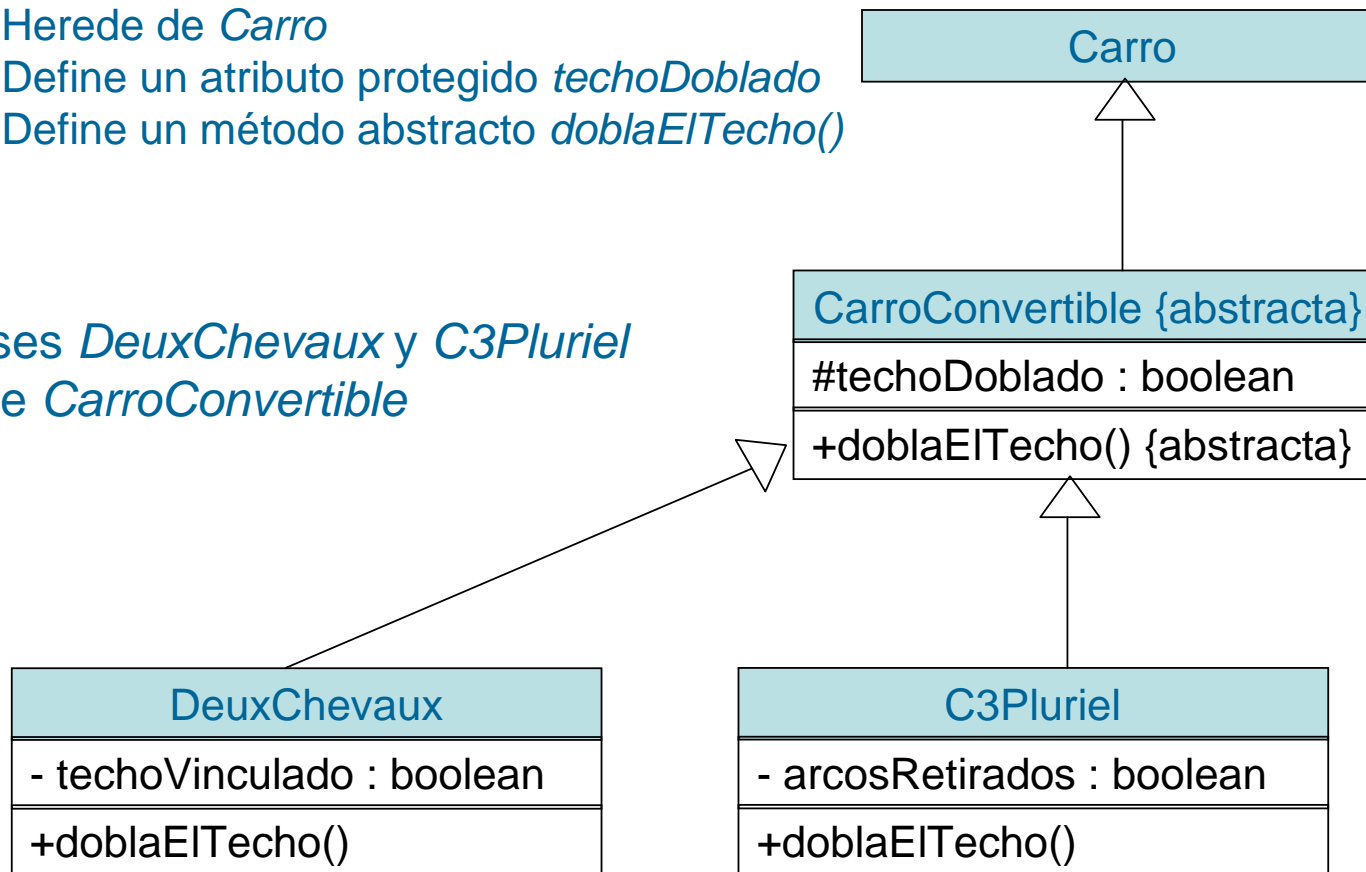
Para crear un método abstracto, se declara su firma (nombre y parámetros) sin especificar el cuerpo y añadiendo la palabra clave **abstract**

Clases abstractas : ejemplo CarroConvertible

➤ La clase *CarroConvertible*

- ❑ Herede de *Carro*
- ❑ Define un atributo protegido *techoDoblado*
- ❑ Define un método abstracto *doblaElTecho()*

➤ Las clases *DeuxChevaux* y *C3Pluriel* hereden de *CarroConvertible*



Clases abstractas : ejemplo CarroConvertible

Clase abstracta

```
public abstract class CarroConvertible
    extends Carro {
    protected boolean techoDoblado;

    public abstract void doblaEITecho();
}
```

Método abstracto

```
public class DeuxChevaux extends CarroConvertible{
    private boolean techoVinculado;

    public void doblaEITecho() {
        this.techoDoblado = true;
        this.techoVinculado = true;
    }
}
```

```
public class C3Pluriel extends CarroConvertible {
    private boolean arcosRetirados;

    public void doblaEITecho() {
        this.techoDoblado = true;
        this.arcosRetirados = true;
    }
}
```



**Atención : no es redefinición.
Se habla de implementación
de método abstracta**

Clases abstractas : ejemplo Carro Convertible

```
public class Prueba {  
    public static void main (String[] argv) {  
        // Declaración y creación de una DeuxChevaux  
        CarroConvertible carroAntiguo = new DeuxChevaux(...);  
        // Envió de mensaje  
        carroAntiguo.doblaEITecho();  
  
        // Declaración y creación de una C3Pluriel  
        CarroConvertible carroReciente = new C3Pluriel(...);  
        // Envió de mensaje  
        carroReciente.doblaEITecho();  
  
        // Declaración y creación de un CarroConvertible  
        CarroConvertible carroConvertible =  
        new CarroConvertible(...); // Error  
    }  
}
```



**Atención : la clase
CarroConvertible
no puede ser instanciada
porque es abstracta**

```
Output - CursoJavaSE (run-single)  
Compiling 1 source file to E:\workspace\CursoJavaSE\build\classes  
E:\workspace\CursoJavaSE\src\ClasesAbstractas\Prueba.java:31:  
ClasesAbstractas.CarroConvertible is abstract; cannot be instantiated  
    new CarroConvertible(); // Error  
1 error  
BUILD FAILED (total time: 1 second)
```

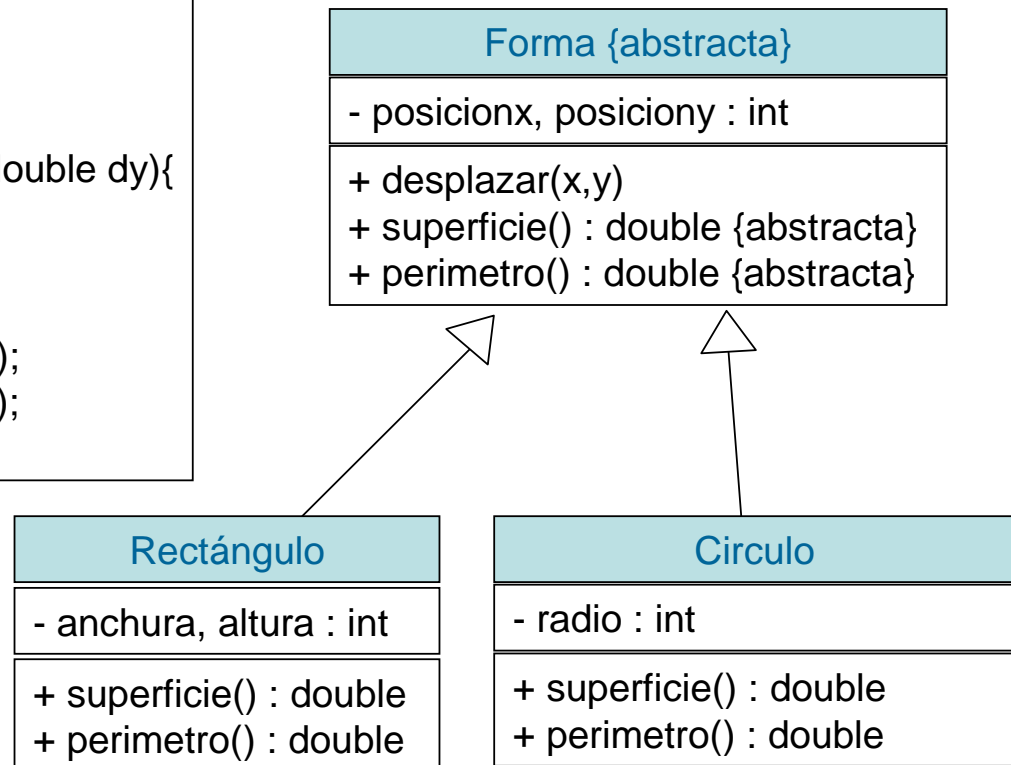
Clases abstractas : ejemplo Forma

➤ La clase *Forma*

- ❑ Los métodos *superficie()* y *perímetro()* son abstractos
- ❑ Estos métodos tienen « sentido » solamente para las sub-clases *Circulo* y *Rectángulo*

```
public abstract class Forma {  
    private int posicionx, posiciony;  
  
    public void desplazar(double dx, double dy){  
        x += dx; y += dy;  
    }  
  
    public abstract double perímetro();  
    public abstract double superficie();  
}
```

No implementación !!



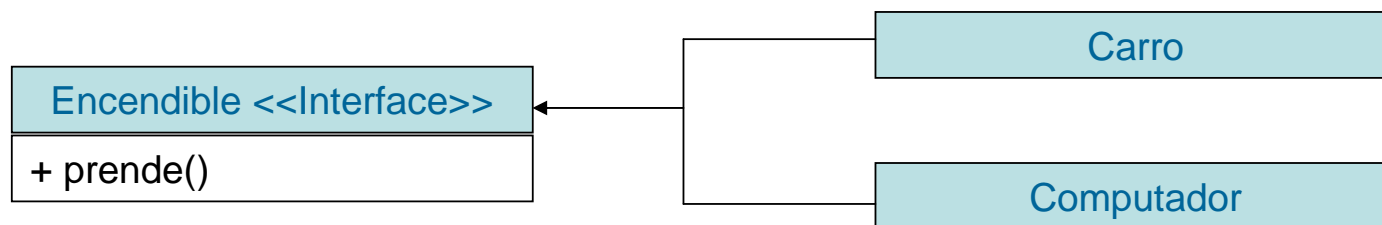
Noción de interface

➤ Un interfase es un modelo para una clase

- ❑ Cuando todos los métodos de una clase son abstractos y no hay ningún atributo : se habla de interfase
- ❑ Una interfase define la firma de los métodos que deben aplicarse en las clases que respetan este modelo
- ❑ Toda clase que implementa el interfase debe implementar todos los métodos definidos por el interfase
- ❑ Todo objeto instancia de una clase que implementa el interfase puede declararse como del tipo de este interfase
- ❑ Les interfaces podrán derivarse

➤ Ejemplo :

- ❑ Las cosas « Encendible » deben tener un método « prende() »



Noción de interface y Java

➤ Aplicación de una interface

- La definición de una interface se presenta como la definición de una clase. Se utiliza simplemente la palabra clave **interface** en cambio de **class**

```
public interface NombreInterface {  
    ...  
}
```



**Interfase : no confundir
con Interfaces graficas**

- Cuando se define una clase, se puede precisar que implementa uno o más interfaces utilizando una vez la palabra clave **implements**

```
public class NombreClases implements Interface1,Interface3, ... {  
    ...  
}
```

- Si una clase hereda de otra clase puede también implementar uno o más interfaces

```
public class NombreClases extends SuperClase implements Interface1, ... {  
    ...  
}
```

Noción de interfase y Java

➤ Aplicación de una interfase

- Una interfase no posee atributo
- Una interfase puede poseer constantes

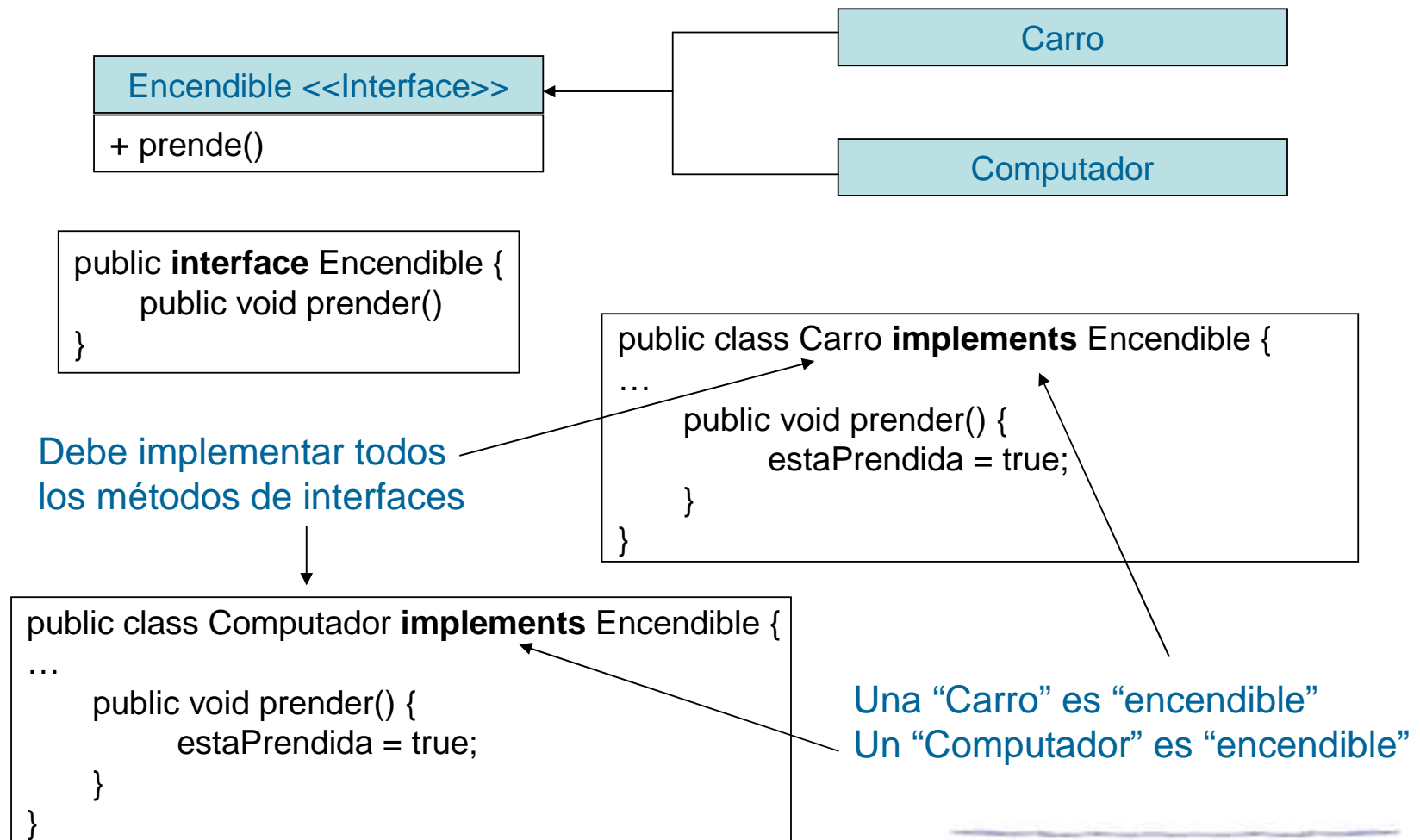
```
public interfaz NombreInterface {  
    public static final int CONST = 2;  
}
```

- Una interfase no posee palabra clave **abstract**
- Los interfaces no son instanciables (Mismo razonamiento con las clases abstractas)

```
NombreInterface intento = new NombreInterface(); // Error!!
```

Noción de interfase y Java

- Toda clase que implementa la interfase debe implementar todos los métodos definidos por el interfase



Noción de interfase y Java

- **Todo objeto instancia de una clase que aplica el interfaz puede declararse como del tipo de este interfaz**

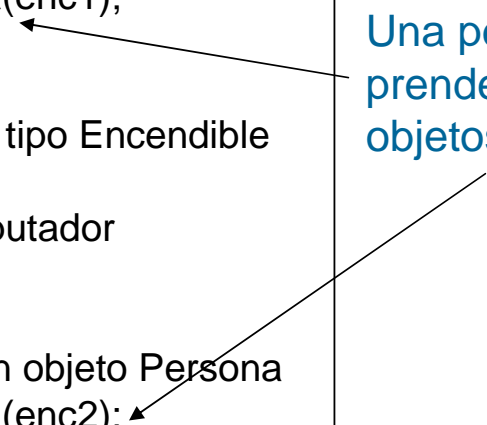
```
public class Prueba {
    public static void main (String[] argv) {
        // Declaración de un objeto de tipo Encendible
        Encendible enc1;
        // Creación de un objeto Carro
        enc1 = new Carro();

        // Declaración y creación de un objeto Persona
        Persona pers1 = new Persona(enc1);
        pers1.ponerEnMarcha();

        // Declaración de un objeto de tipo Encendible
        Encendible enc2;
        // Creación de un objeto Computador
        enc2 = new Computador();

        // Declaración y creación de un objeto Persona
        Persona pers2 = new Persona(enc2);
        pers2.ponerEnMarcha();
    }
}
```

Una persona puede prender todos los objetos "Encendible"



Noción de interfase y Java

- Un “carro” y un “computador” son objetos “encendibles”

```
public class Persona {  
  
    private Encendible objetoEncendible;  
  
    public Persona(Encendible enc) {  
        objetoEncendible = enc;  
    }  
  
    public void ponerEnMarcha() {  
        objetoEncendible.prender();  
    }  
  
}
```

**Una persona puede prender
Carro y Computador
sin conocer su natura exacta**

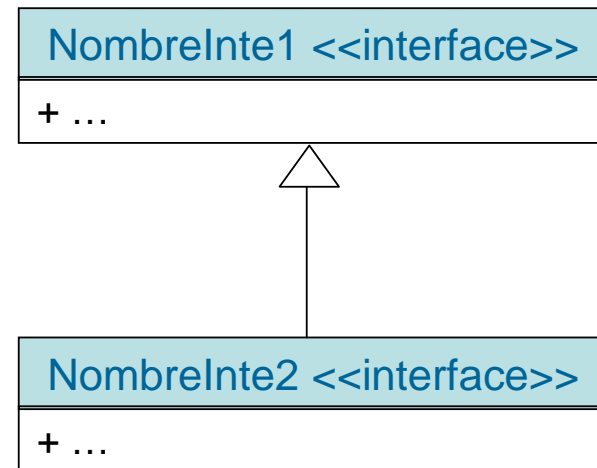
Noción de interfase y Java

➤ Las interfaces podrán derivarse

- Una interfase puede heredar de otra interface : « extends »

➤ Consecuencias

- La definición de métodos del interfaz madre (NombreInte1) se reanuda en el interfaz hija (NombreInte2). Toda clase que implementa la interfase hija debe dar una implementación a todos los métodos y también a los métodos heredadas



➤ Utilización

- Cuando un modelo puede definirse en varios sub-modelos complementarios

Clases abstractas versus interfaces

➤ Las clases

- Se implementan completamente
- Otra clase puede heredar de una clase

➤ Las clases abstractas

- Se implementan parcialmente
- Otra clase puede heredar de una clase abstracta pero debe dar una implementación a los métodos abstractos
- Ellas no pueden ser instanciadas pero pueden tener un constructor

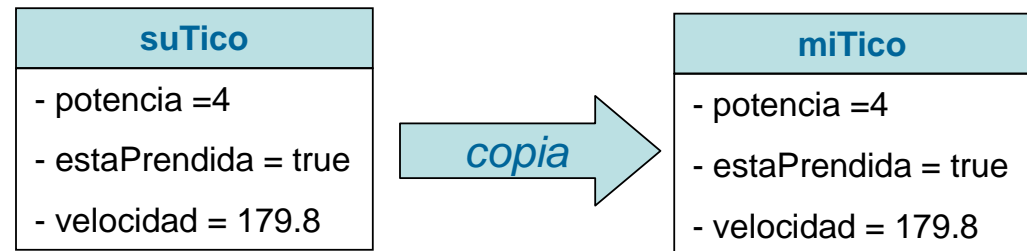
➤ Las interfaces

- No son implementadas
- Toda clase que implementa una o más interfaces debe implementar todos sus métodos (abstractas)

El interface “Cloneable”

➤ Dos posibilidades para duplicar un objeto de sus clases

- Crear un método « `public MiClase duplicar()` » que devuelve una copia del objeto creando a una nueva instancia e inicializado los atributos (utilizando el constructor)



- Utilizar el interface « Cloneable » para conservar la compatibilidad con las otras clases de Java

- Implementar el método « `protected Object clone()` » del interface Cloneable

```
public class Carro implements Encendible, Cloneable {  
    protected Object clone() {  
        Carro copia;  
        copia = new Carro(this.potencia, (Galeria)laGaleria.clone());  
        return copia;  
    }  
}
```

Las clases internas “Inner classes”

➤ Regla de base en Java

- Una clase por archivo y un archivo por clase

➤ Clases locales o internas

- Definidas a dentro de otras clases (Motor en Carro)

```
public class Carro {  
    ...  
    class Motor {  
        ...  
    }  
}
```

➤ Clases anónimas

- Son instancias de clases y implementaciones de una clase abstracta o de una interfase
- La o los métodos abstractas deben ser implementadas al momento del instanciacion

```
Encendible unaInstancia =  
    new Encendible(){  
        public void prende() {  
            // Código aquí  
        }  
    };
```



Las clases anónimas son muchas utilizadas para el desarrollo de IHM con Java/Swing

Las clases internas “Inner classes”

➤ Código fuente : 1 archivo

- clase
- clase anónima
- clase interna

➤ Generación de byte-code : 3 archivos

- clase « Carro.class »
- anónima « Carro\$1.class »
- interne « Carro\$Motor.class »

Clase anónima, implementa el interfase ClaseA

```
public class Carro {
    public Carro() {
        Motor miMotor = new Motor(...);
        ClaseA milnit = new ClaseA() {
            public void inicialización() {
                ...
            }
        };
    }
}

class Motor {
    ...
    public Motor(...) {
        ...
    }
}
```

Clase interna

Nombre	Tamaño	Tipo
Carro\$1.class	1 KB	Archivo CLASS
Carro\$Motor.class	1 KB	Archivo CLASS
Carro.class	1 KB	Archivo CLASS
Carro.java	1 KB	Archivo JAVA

Los archivos .class que tienen en sus nombres un \$ no son archivos temporales!!!



Programación Orientada Objeto Aplicación al lenguaje JAVA

Los indispensables



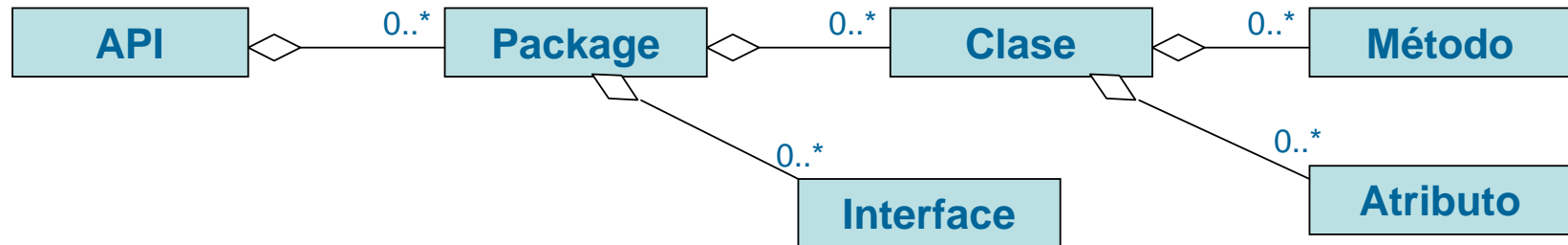
Institut de recherche
pour le développement

Jérémie HABASQUE – 2007
mailto:jeremie_habasque@yahoo.fr

Los packages

➤ El lenguaje Java propone una definición muy clara del mecanismo que permite clasificar y administrar los API externos

➤ Los API son constituidas :



➤ Un package es un grupo de clases asociadas a una funcionalidad

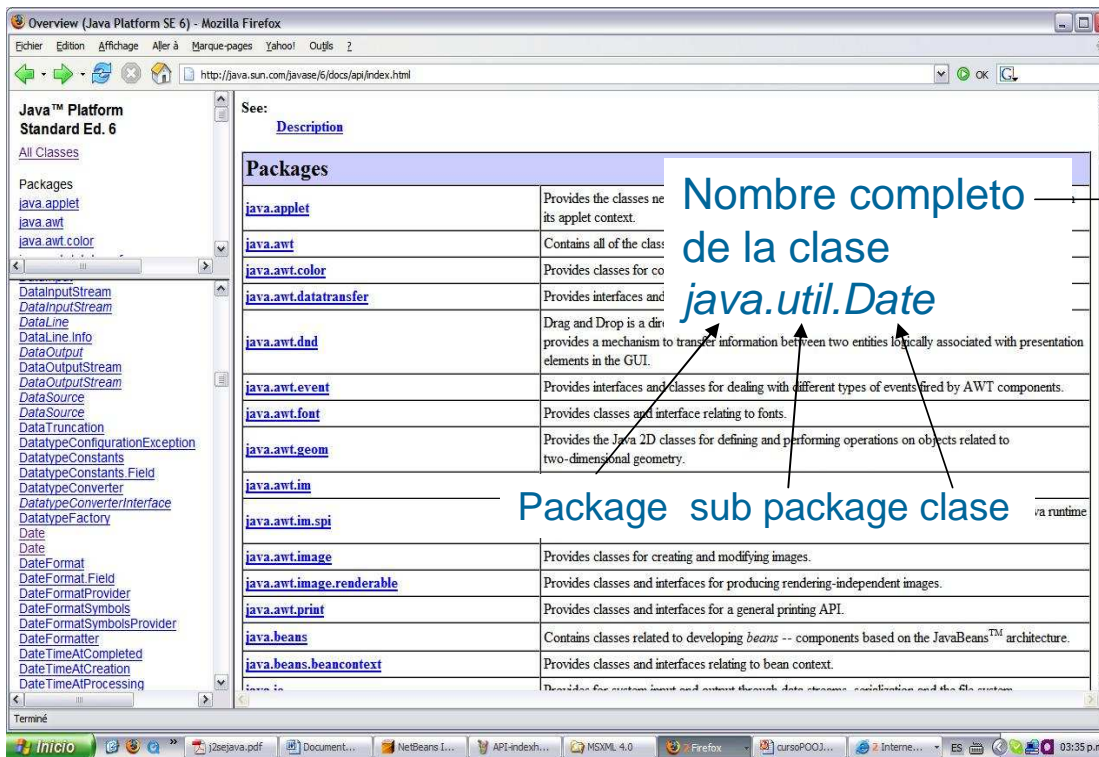
➤ Ejemplos de packages

- java.lang* : reúna las clases de base Java (*Object*, *String*, *System*, ...)
- java.util* : reúna las clases utilitarias (*Collections*, *Date*, ...)
- java.io* : lectura y escritura
- ...

Los packages : OK, pero para que hacer ?

➤ La utilización de los packages de medidas permite agrupar las clases para organizar librerías de clases Java

➤ Ejemplo : la clase *Date* esta definida dos veces



java.sql
Class Date

java.lang.Object
└─ java.util.Date

└─ java.sql.Date

All Implemented Interfaces:
Serializable, Cloneable, Comparable<Date>

java.util
Class Date

java.lang.Object
└─ java.util.Date

All Implemented Interfaces:
Serializable, Cloneable, Comparable<Date>

Direct Known Subclasses:
Date, Time, Timestamp

Los packages : utilización de clases

➤ Cuando, en un programa, hay una referencia a una clase, el compilador la búsqueda en el package por defecto (*java.lang*)

➤ Para los otros, es necesario de proporcionar explícitamente la información para saber donde se encuentra la clase :

□ Utilización de **import** (clase o package)

```
import misclases.Punto;  
import java.lang.String; // No sirve a nada porque por defecto  
import java.io.ObjectOutput;
```



```
import misclases.*;  
import java.lang.*; // No sirve a nada porque por defecto  
import java.io.*;
```

□ Nombre del package con el nombre de la clase

```
java.io.ObjectOuput toto = new java.io.ObjectOuput(...)
```

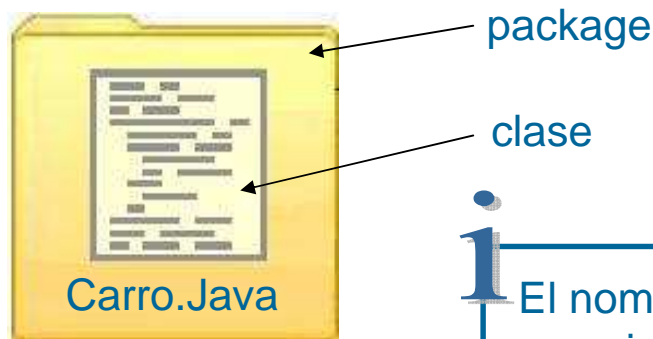


Escritura muy pesada
preferir la solución con
la palabra clave import

Los packages : su “existencia” física

- A cada clase Java corresponde un archivo
- A cada package (sub-package) corresponde una carpeta

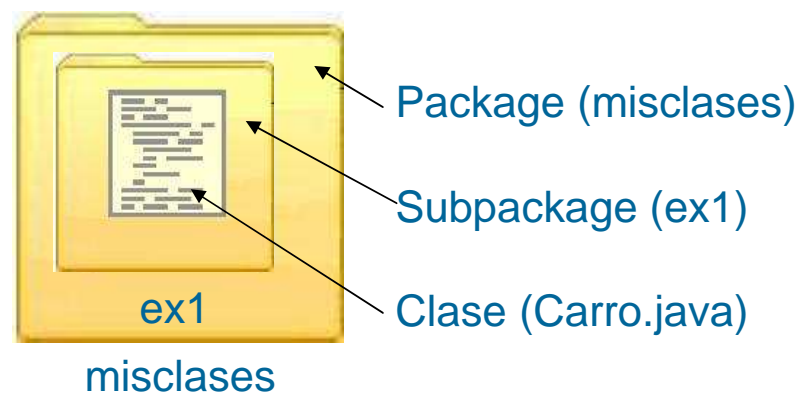
Ejemplo : *misclases.Carro*



i El nombre de los packages siempre se escribe en minúscula

- Un package puede contener
 - Clases o interfaces
 - Un otro package (sub-package)

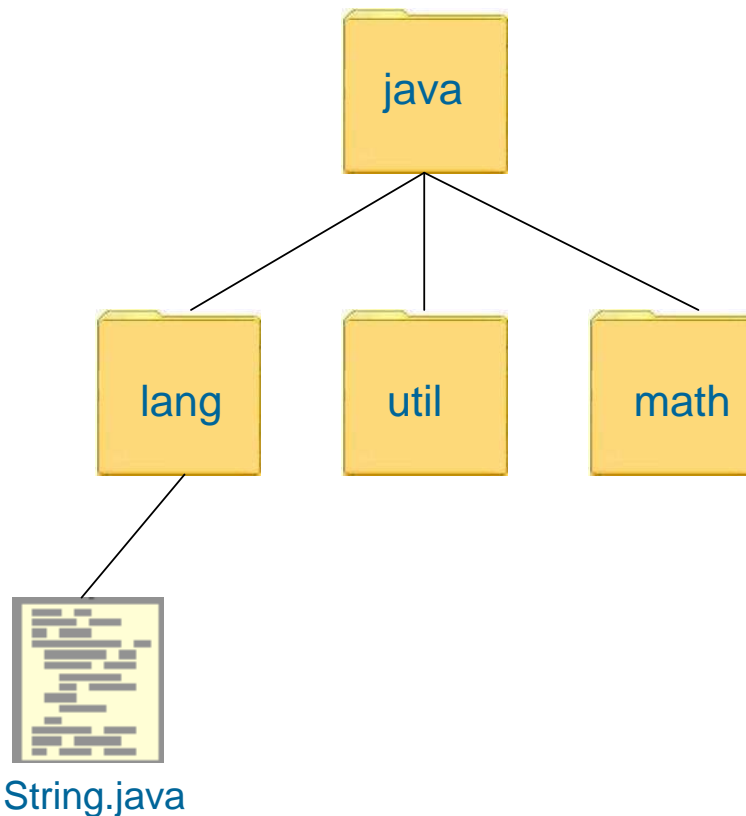
Ejemplo : *misclases.ex1.Carro*



Los packages : jerárquica de packages

➤ A una jerárquica de packages corresponde una jerárquica de carpetas cuyas nombres coinciden con los componentes de los nombres de package

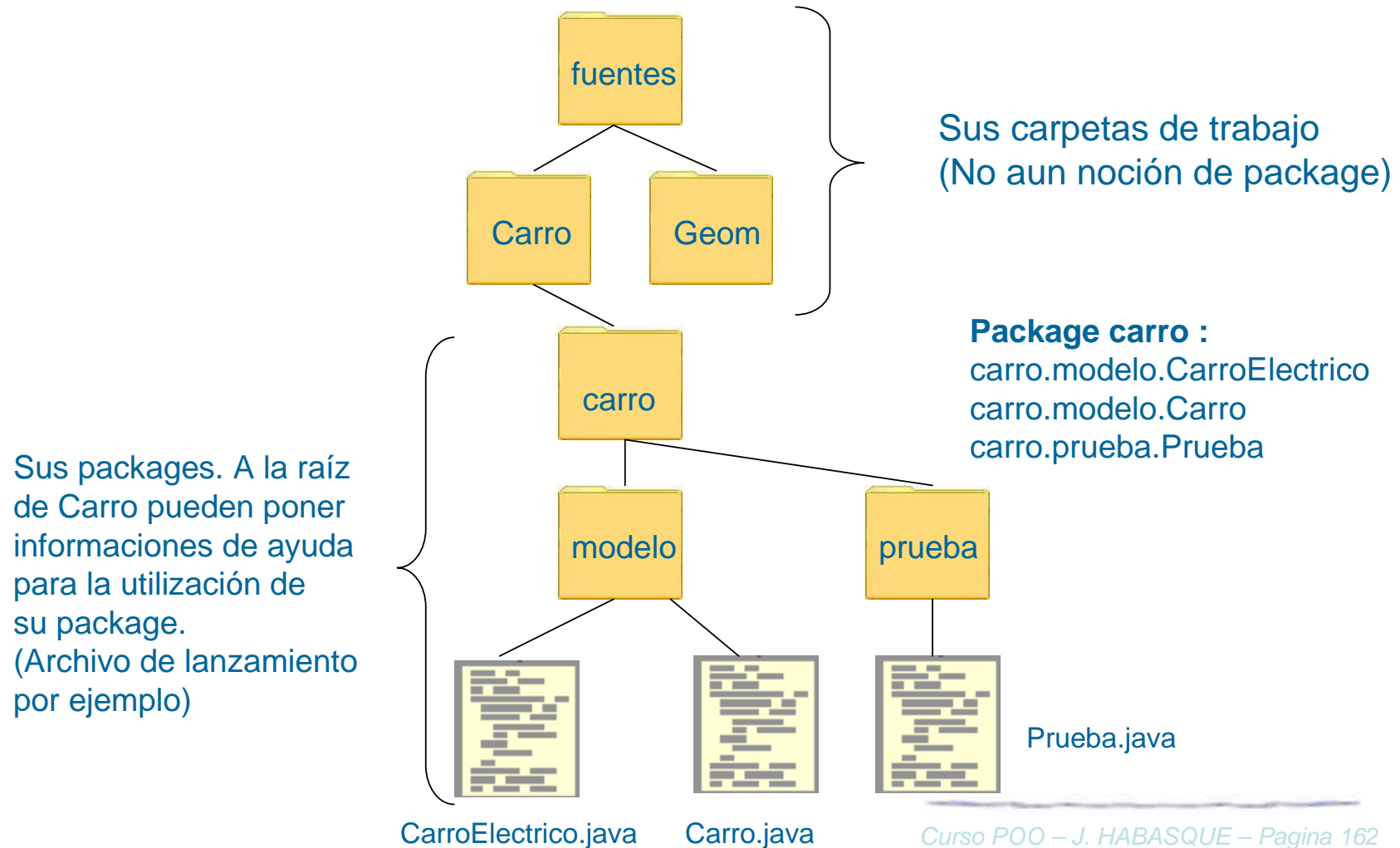
➤ Ejemplo : la clase *String*



- Biblioteca pura Java
- Las fuentes (*.java) se encuentran en la carpeta *src* de la carpeta Java
- Los bytecodes (*.class) se encuentran en el archivo *rt.jar* de la carpeta Java

Los packages : creación y consejos

- Cuando crean un proyecto, nombrar el package de mas alto nivel (carro por ejemplo) al nombre del proyecto (*Carro* por ejemplo)



Los packages : creación y consejos


- Para especificar a una clase que pertenece a una clase, utilizar la palabra clave **package**

```
package carro.modelo;
public class CarroElectrico {
    ...
}
```


```
package carro.modelo;
public class Carro {
    ...
}
```

```
package carro.prueba;
import carro.modelo.CarroElectrico;
import carro.modelo.Carro;
import ...

public class Prueba1 {
    public static void main(String[] argv) {
        ...
    }
}
```



La palabra clave package siempre se coloca en primera instrucción de una clase



No confundir herencia y package. No es la misma cosa. *CarroElectrico* es en el mismo package que *Carro*

Los packages : compilación y ejecución

- Estar en la raíz de la carpeta Carro



- La compilación debe tener en cuenta las direcciones de los packages

```
javac carro\modelo\*.java carro\prueba\*.java
```

- La ejecución se hace indicando la clase principal con su dirección

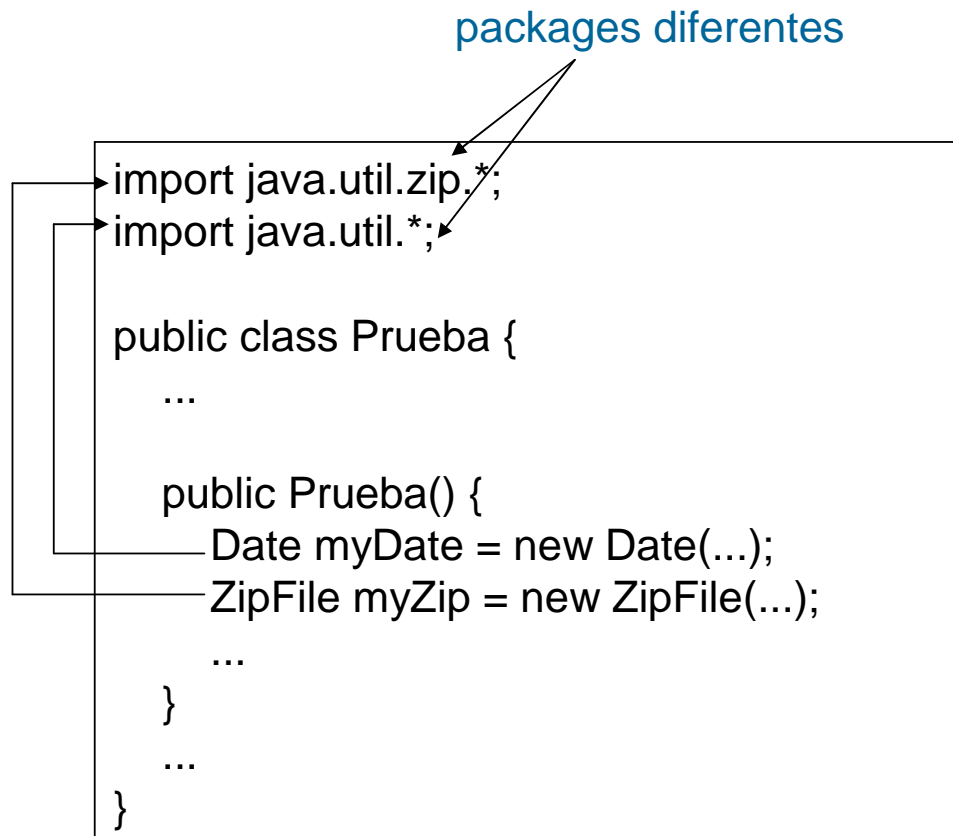
```
java carro.prueba.Prueba
```



La separación entre package, sub-packages y clases se hace con ayuda de un punto « . » y no de un anti-slash « \ »

Los packages : compilación y ejecución

- La instrucción `import nombrePackage.*` no concierne que las clases del package indicado. No se aplica a los clases de las sub-clases



Prueba utiliza las clases *Date* del package *java.util* y *ZipFile* del package *java.util.zip*

Javadoc y los comentarios

➤ Dos tipos de comentarios

- ❑ Comentarios de tratamientos : precisión sobre el código el mismo
- ❑ Comentarios de documentación (herramienta **javadoc** del JDK : generación automática de paginas HTML)

➤ Clases, constructores, métodos y campos

- ❑ Incluido entre `/**` y `*/`
 - ❑ Première línea : únicamente `/**`
 - ❑ Siguietes : un espacio seguido de una estrella
 - ❑ Última línea : únicamente `*/` precedido de un espacio

```
/**
 * Descripción del método
 * Otras características
 */
public Carro(...) {
    ...
}
}
```



**Añadir sentido y precisión a sus códigos.
¡Explicar no es traducir!!**

Javadoc y los comentarios

➤ Javadoc y intereses

- **Javadoc** es a las clases lo que son las paginas de manual (**man**) son a **Unix** o lo que es **Windows Help** es a los aplicaciones **MS Windows**
- Redacción de la documentación técnica de las clases durante el desarrollo de estas mismas clases luego generación final del HTML

➤ Utilización

- La entidad documentada esta precedida por su comentario
- Seguir la presentación precedente de descripción de los métodos, clases, ...
- Utilización de tags definidos por **javadoc** permitiendo de caracterizar algunas informaciones (utilización posible de baliza HTML)

@author	■	Nombre del o de los autores
@version	■	Identificador de versión
@param	■	Nombre y significación del argumento (métodos únicamente)
@since	■	Versión del JDK donde aprecio (utilizado par SUN)
@return	■	Valor de vuelta
@throws	■	Clase del excepción y condiciones de lanzamiento
@deprecated	■	Provoque las advertencias de desaprobación
@see	■	Referencia cruzada

Javadoc y los comentarios

➤ Ejemplo con la fuente de la clase *Object*

```
package java.lang;
/**
 * Class Object is the root of the class hierarchy.
 * Every class has Object as a superclass. All objects,
 * including arrays, implement the methods of this class.
 *
 * @author unascribed
 * @version 1.58, 12/03/01
 * @see java.lang.Class
 * @since JDK1.0
 */
public class Object {
/**
 * Returns the runtime class of an object. That Class
 * object is the object that is locked by static synchronized
 * methods of the represented class.
 *
 * @return the object of type Class that represents the
 * runtime class of the object.
 */
public final native Class getClass();
...
}
```

➤ Generación del código HTML a partir de la herramienta **javadoc**



Para obtener las
informaciones de javadoc
`javadoc -help`

javadoc [options] nombreDeLasClasesJava.java

Javadoc y los comentarios

- Ejemplo de la pagina HTML de la descripción de la clase *Object* generada con javadoc

java.lang
Class Object
java.lang.Object

public class Object

Class `Object` is the root of the class hierarchy. Every class has `Object` methods of this class.

Since:
JDK1.0

See Also:
[Class](#)

Constructor Summary

[Object\(\)](#)

Method Summary

protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this <code>Object</code> .
int	hashCode() Returns a hash code value for the object.

Method Detail

getClass

public final [Class](#)<?> [getClass\(\)](#)

Returns the runtime class of this `Object`. The returned `Class` object is the object that is locked by `static synchronized` methods of the represented class.

The actual result type is `Class<? extends |X|>` where `|X|` is the erasure of the static type of the expression on which `getClass` is called. For example, no cast is required in this code fragment:

```
Number n = 0;  
Class<? extends Number> c = n.getClass();
```

Returns:
The `Class` object that represents the runtime class of this object.

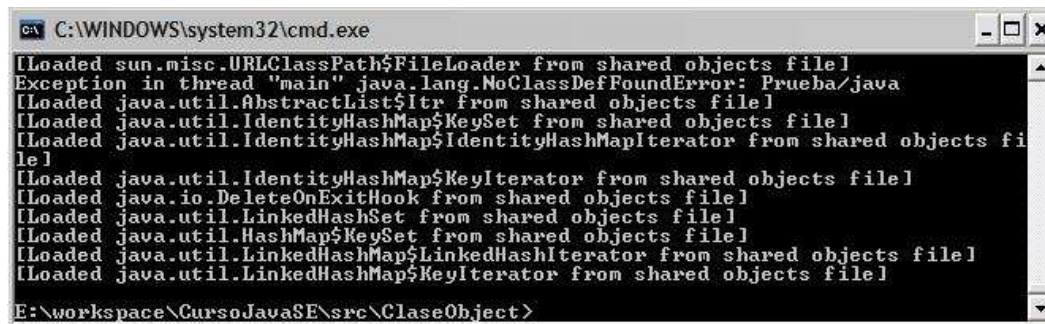
See Also:
[The Java Language Specification, Third Edition \(15.8.2 Class Literals\)](#)

Jar

➤ Jar y intereses

- **jar** es la herramienta estándar para construir los archivos que tienen el mismo objetivo que las bibliotecas de programas utilizadas por algunos lenguajes de programación (lib por ejemplo)

java **-verbose** HelloWorld

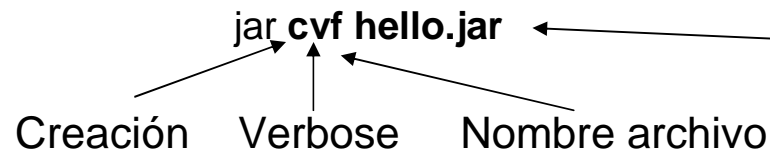


```
C:\WINDOWS\system32\cmd.exe
[Loaded sun.misc.URLClassPath$FileLoader from shared objects file]
Exception in thread "main" java.lang.NoClassDefFoundError: Prueba/java
[Loaded java.util.AbstractList$Itr from shared objects file]
[Loaded java.util.IdentityHashMap$KeySet from shared objects file]
[Loaded java.util.IdentityHashMap$IdentityHashMapIterator from shared objects file]
[Loaded java.util.IdentityHashMap$KeyIterator from shared objects file]
[Loaded java.io.DeleteOnExitHook from shared objects file]
[Loaded java.util.LinkedHashSet from shared objects file]
[Loaded java.util.HashMap$KeySet from shared objects file]
[Loaded java.util.LinkedHashMap$LinkedHashMapIterator from shared objects file]
[Loaded java.util.LinkedHashMap$KeyIterator from shared objects file]
E:\workspace\CursoJavaSE\src\ClaseObject>
```

Muestra los archivos
Utilizados para ejecutar
el programa HelloWorld

➤ Utilización para la creación

- Utilización de la herramienta **jar**
- Para crear un archivo **.jar** conteniendo todos los archivos de la carpeta corriente



El . indica la carpeta corriente

Jar

➤ Utilización para la creación

- ❑ Utilización de un archivo manifiesto (*MANIFEST.MF*) que precise un conjunto de atributos para ejecutar en buenas condiciones la aplicación
- ❑ El atributo *Main-class* por ejemplo permite de conocer la clase principal a ejecutar

```
Manifest-Version: 1.0
Created-By: 1.4.1_01 (Sun Microsystems Inc.)
Main-class: HelloWorld
```



MANIFEST.MF

- ❑ Creación del jar con un archivo manifiesto :

```
jar cvfm hello.jar MANIFEST.MF .
```

- ❑ Utilización para la ejecución

```
java -jar hello.jar
```

Esta opción permite de ejecutar a partir de un jar código Java



La clase HelloWorld se encarga por medio del archivo MANIFEST.MF

Excepción

➤ Definición

- Una excepción es una señal que indica que algo de excepcional (como un error) se produjo. Este excepción para la ejecución normal del programa.

➤ A que sirve eso ?

- Administrar los errores es indispensable :
 - Mala gestión puede tener consecuencias importantes
- Mecanismo simple et lisible :
 - Reagrupación del código reservado al tratamiento de los errores (no hay el mezcla” con el algoritmo)
 - Posibilidad de « recuperar » un error a varios niveles de una aplicación (propagación en la pila de las llamadas de métodos)

➤ Vocabulario

- Lanzar o desencadena (**throw**) una excepción una excepción consiste en indicar los errores
- Capturar o coger (**catch**) una excepción permite de tratar los errores

Excepción

➤ Primer ejemplo : lanzar y coger una excepción

```
public class Punto {  
    ... // Declaración de los atributos  
  
    ... // Otros métodos y constructores  
  
    public Punto(int x, int y) throws ErrConst {  
        if ((x < 0) || (y < 0)) throw new ErrConst();  
        this.x = x ; this.y = y;  
    }  
  
    public void visualizar() {  
        System.out.println("Datos : " + x + " " + y);  
    }  
}
```

No se define aún
la clase *ErrConst*.
A ver más tarde

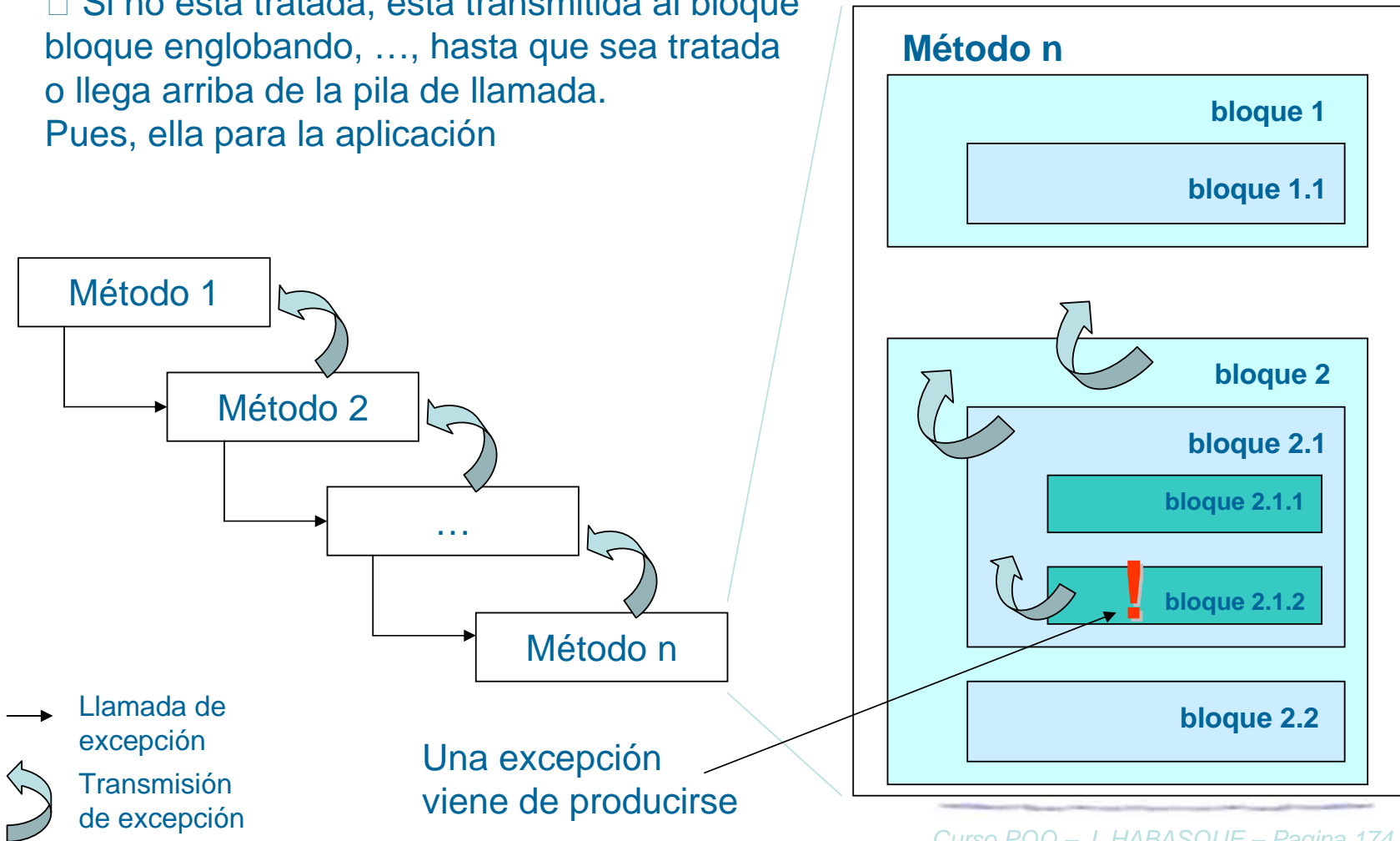
```
public class Prueba {  
    public static void main(String[] argv) {  
        try {  
            Punto a = new Punto(1,4);  
            a.visualizar();  
            a = new Punto(-2, 4);  
            a.visualizar();  
        } catch (ErrConst e) {  
            System.out.println("Error Construcción");  
            System.exit(-1);  
        }  
    }  
}
```

```
Output - CursoJavaSE (run-single)  
run-single:  
Datos : 1 4  
Error Construcción  
Java Result: -1  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Excepción : mecanismo

➤ Explicación

- ❑ Cuando se encuentra una situación excepcional, una excepción esta lanzada
- ❑ Si no esta tratada, esta transmitida al bloque englobando, ..., hasta que sea tratada o llega arriba de la pila de llamada. Pues, ella para la aplicación



Excepción : lanzar o activar

- Un método declara que ella puede lanzar una excepción por la palabra clave **throws**

```
public Punto(int x, int y) throws ErrConst {  
    ...  
}
```

Permite al constructor *Punto* de lanzar una excepción *ErrConst*

- O el método lanza una excepción, creando una nueva valor (un objeto) de excepción utilizando la palabra clave **throw**

```
public Punto(int x, int y) throws ErrConst {  
    if ((x < 0) || (y < 0)) throw new ErrConst();  
    this.x = x ; this.y = y;  
}
```

Creación de una nueva valor de excepción

- O el método llama código que lanza la excepción

```
public Punto(int x, int y) throws ErrConst {  
    checkXYValue(x,y);  
    this.x = x ; this.y = y;  
}
```

```
private void checkXYValue(in x, int y)  
throws ErrConst {  
    if ((x < 0) || (y < 0))  
        throw new ErrConst();  
}
```

Excepción : capturar o coger

- Se habla aquí de gestor de excepción. Se trata de tratar por acciones la situación excepcional
- Se delimita un conjunto de instrucciones susceptibles de activar una excepción por bloques **try {...}**

```
try {  
    Punto a = new Punto(1,4);  
    a.visualizar();  
    a = new Punto(-2, 4);  
    a.visualizar();  
}
```

- La gestión de riesgos es obtenida por bloques **catch(TipoExcepcion e) {...}**

```
} catch (ErrConst e) {  
    System.out.println("Error Construcción");  
    System.exit(-1);  
}
```

- Estos bloques permiten capturar las excepciones cuyas el tipo se especifica y realizar acciones adecuadas

Excepción : capturar o coger

➤ Comprensión del mecanismo de captura

```
public class Prueba {  
    public static void main(String[] argv) {  
        try {  
            Punto a = new Punto(1,4);  
            a.visualizar();  
            a = new Punto(-2, 4);  
            a.visualizar();  
        } catch (ErrConst e) {  
            System.out.println("Error Construcción");  
            System.exit(-1);  
        }  
        ...  
    }  
}
```

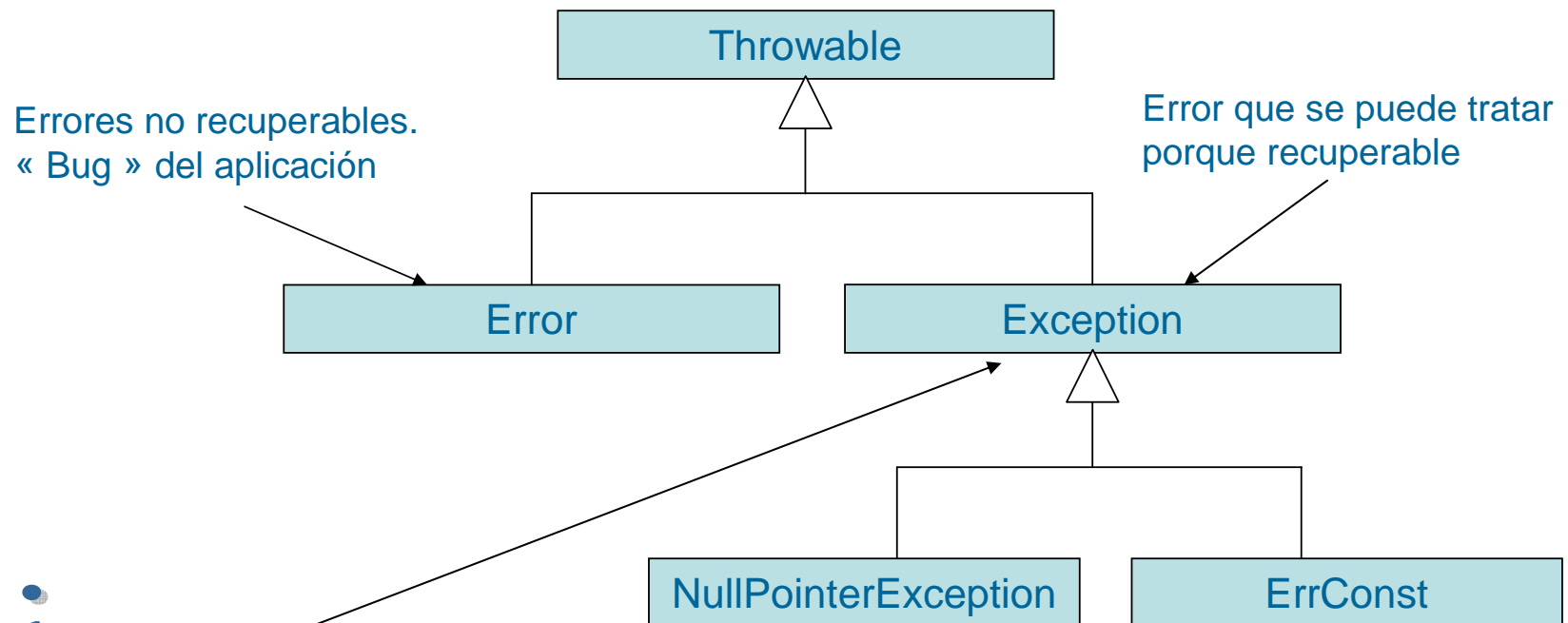
El error excepcional esta
tratada por el bloque **catch**

Luego, hay continuación de la ejecución
fuera del bloque **try catch**

Observación: si error el programa se
para(System.exit(-1))

Excepción : modelización

- Las excepciones en Java son consideradas como objetos
- Toda excepción debe ser una instancia de una sub-clase de la clase *java.lang.Throwable*

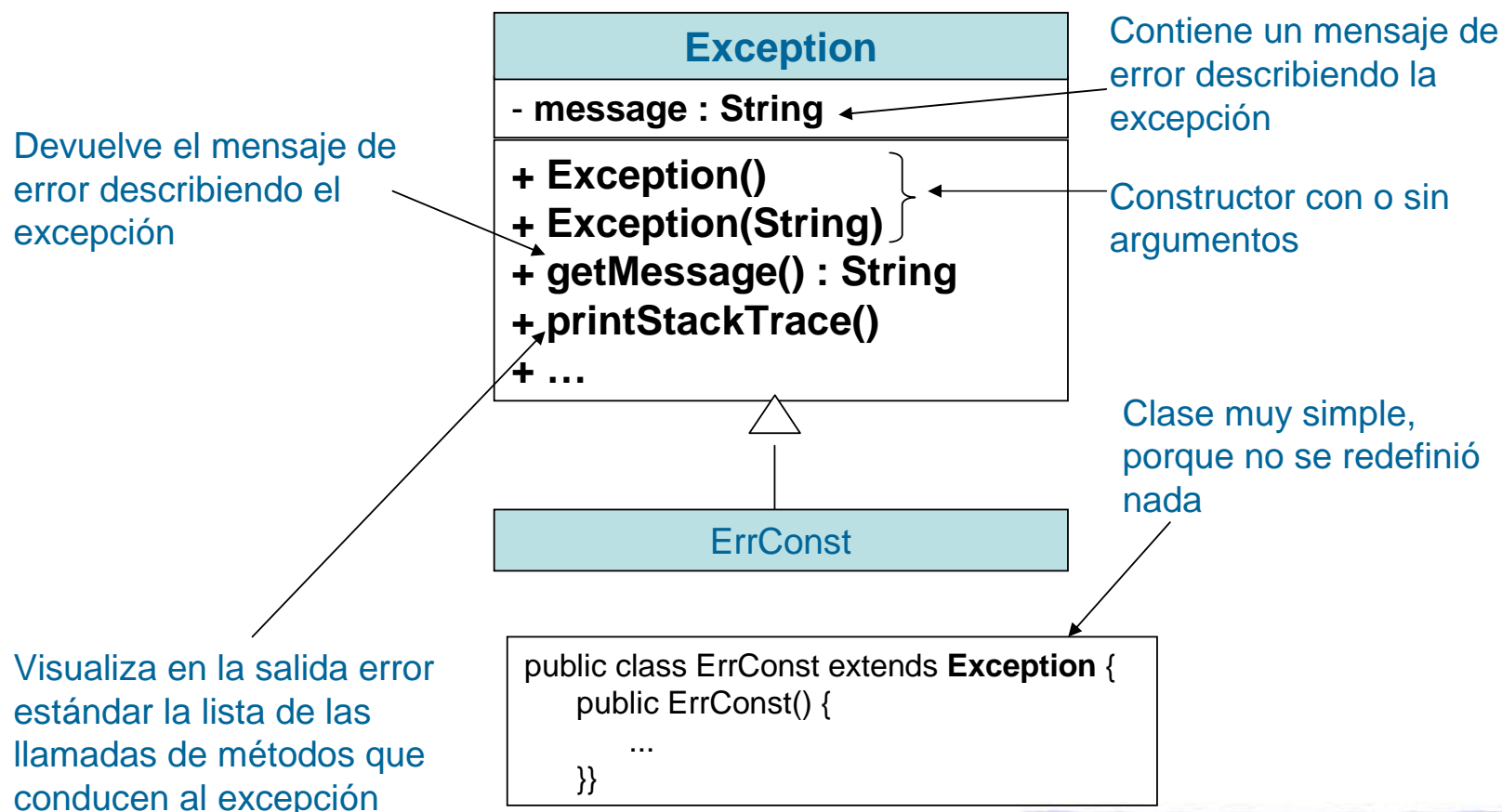


i Para definir nuevos tipos de excepción, se clasificará la clase *Exception*

Excepción : modelización

➤ Las excepciones son objetos, entonces podemos definir

- Atributos particulares
- Métodos



Excepción : modelización

➤ Utilización del objeto *ErrConst*

```
public class Prueba {
    public static void main(String[] argv) {
        try {
            ...
        } catch (ErrConst e) {
            System.out.println("Error Construcccion");
            System.out.println(e.getMessage());
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

Error de tipo *ErrConst* que hereda de Exception

Visualización del error

Visualización de la lista de los métodos

```
: Output - CursoJavaSE (run-single)
run-single:
Datos : 1 4
Error Construcccion
excepcion.ErrConst
null
|
|   at excepcion.Punto.<init>(Punto.java:25)
|   at excepcion.Prueba.main(Prueba.java:22)
Java Result: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Excepción : cogerlos todos...

- Es posible de capturar mas que una excepción. Un bloque **try** y varios bloques **catch**

```
public class Punto {
    public void desplazar(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) < 0)) throw new ErrDepl();
        x += dx ; y +=dy;
    }

    public Punto(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst();
        this.x = x ; this.y = y;
    }
}
```

Definición de un nuevo método que lanza una excepción

Coge la nueva excepción de tipo *ErrDepl*

```
... public class Prueba {
    public static void main(String[] argv) {
        try {
            ... // Bloque en que se desea detectar las excepciones ErrConst y ErrDepl
        } catch (ErrConst e) {
            System.out.println("Error Construccion");
            System.exit(-1);
        } catch (ErrDepl e) {
            System.out.println("Error Desplazamiento");
            System.exit(-1);
        }
    }
}
```

Excepción : cogerlos todos...

➤ Todo método susceptible de activar una excepción debe :

- ❑ O sea cogerla (bloque **try catch**)
- ❑ O declara explícitamente que ella puede lanzar una excepción (palabra clave **throws**)

➤ Las excepciones declaradas en la cláusula **throws** de un método son:

Las excepciones activadas en el método (*Punto*) y no cogidas por el método

```
public Punto(int x, int y) throws ErrConst {  
    if ((x < 0) || (y < 0)) throw new ErrConst();  
    this.x = x ; this.y = y;  
}
```

Las excepciones activadas en los métodos (*checkXYValue*) llamadas por el método (*Punto*) y no cogidas por el método

```
public Punto(int x, int y) throws ErrConst {  
    checkXYValue(x,y);  
    this.x = x ; this.y = y;  
}
```

```
private void checkXYValue(in x, int y) throws ErrConst {  
    if ((x < 0) || (y < 0))  
        throw new ErrConst();  
}
```

Excepción : cogernos todos...

- Es necesario garantizar que las excepciones están bajo controles

```
public class Punto {  
    public void desplazar(int dx, int dy) throws ErrDepl {  
        if (((x+dx) < 0) || ((y+dy) < 0)) throw new ErrDepl();  
        x += dx ; y +=dy;  
    }  
  
    public void transformar() {  
        ...  
        this.desplazar(...);  
    }  
}
```

```
public class ErrDepl extends Exception {  
    public ErrDepl() {  
        ...  
    }  
}
```

```
Output - CursoJavaSE (compile-single)  
Compiling 1 source file to E:\workspace\CursoJavaSE\build\classes  
E:\workspace\CursoJavaSE\src\excepcion\Prueba.java:23: unreported  
exception excepcion.ErrDepl; must be caught or declared to be thrown  
    a.desplazar(4,5);  
  
1 error  
BUILD FAILED (total time: 0 seconds)
```



**No olvidar tratar una excepción
sino el compilador
no lupa ustedes !**

Excepción : cogelos todos...

➤ Para garantizar una buena compilación, dos soluciones :

```
public class Punto {
    public void desplazar(int dx, int dy) throws ErrDepl {
        if (((x+dx) < 0) || ((y+dy) < 0)) throw new ErrDepl();
        x += dx ; y +=dy;
    }

    public void transformar() {
        ...
        this.desplazar(...);
    }
}
```

O añadiendo explícitamente
el instrucción **throws** al
método *transformar* de manera
a redirigir el error

```
public void transformar()
    throws ErrDepl {
    ...
    this.desplazar(...);
}
```

O rodeando de un bloque
try ... catch el método que
puede plantear problema

```
public void transformar() {
    try {
        ...
        this.desplazar(...);
    } catch (ErrDepl e) {
        e.printStackTrace();
    }
}
```


Excepción : transmisión de información

- Posibilidad de enriquecer la clase *ErrConst* añadiendo atributos y métodos de manera a comunicar

```
public class Punto {
    public Punto(int x, int y) throws ErrConst {
        if ((x < 0) || (y < 0)) throw new ErrConst(x,y);
        this.x = x ; this.y = y;
    }
    ...
}
```

```
public class ErrConst extends Exception {
    private int abs, ord;

    public ErrConst(int x, int y) {
        this.abs = x;
        this.ord = y;
    }

    public int getAbs() { return this.abs; }
    public int getOrd() { return this.ord; }
}
```

```
public class Prueba {
    public static void main(String[] argv) {
        try {
            ...
            a = new Punto(-2, 4);
        } catch (ErrConst e) {
            System.out.println("Error Construcción punto");
            System.out.println("Datos deseadas : "
                + e.getAbs() + " " + e.getOrd());
            System.exit(-1);
        }
    }
    ...
}
```

ErrConst

- abs, ord :int

+ ErrConst(x,y)

+ getAbs : int

+ getOrd : int

ErrConst permite de conocer los valores que hicieron fallar la construcción de Punto

Excepción : finally

- Bloc finally : es una instrucción opcional que puede servir de « limpieza ». Esta ejecutada cualquier que sea el resultado del bloque del bloque try (es decir haya activado una excepción o no)
- Permite de especificar código cuyo la ejecución esta garantía en cualquier caso
- Interés doble :
 - Reunir en un único bloque un conjunto de instrucciones que sino que ser duplicadas
 - Efectuar tratamientos después del bloque try, aunque una excepción fue aumentada y no cogida por los bloques catch

Excepción : finally

```
public class Prueba {
    public static void main(String[] argv) {
        try {
            ... // Bloque en el cual se desea detectar
                las excepciones ErrConst et ErrDepl
        } catch (ErrConst e) {
            System.out.println("Error Construcción punto");
            System.out.println("Fin del programa");
            System.exit(-1);
        } catch (ErrDepl e) {
            System.out.println("Error Desplazamiento punto");
            System.out.println("Fin del programa");
            System.exit(-1);
        }
    }
}
```

Estos instrucciones están llamadas varias veces

Por medio de la palabra clave **finally**, es posible descomponer en factores

```
public class Prueba {
    public static void main(String[] argv) {
        try {
            ... // Bloque en el cual se desea detectar
                las excepciones ErrConst et ErrDepl
        } catch (ErrConst e) {
            System.out.println("Error Construcción punto");
        } catch (ErrDepl e) {
            System.out.println("Error Desplazamiento punto");
        } finally {
            System.out.println("Fin del programa");
            System.exit(-1);
        }
    }
}
```

Excepción : para o contra ?

```
errorTipica leerArchivo() {
    int codigoError = 0;
    // Abrir el archivo
    if (isFileIsOpen()) {
        // Determine la longitud del archivo
        if (getFileSize()) {
            // Verificación del asignación de la memoria
            if (getEnoughMemory()) {
                // Leer el archivo en memoria
                if (readFailed()) {
                    codigoError = -1;
                }
            } else {
                codigoError = -2;
            }
        } else {
            codigoError = -3;
        }
        // Cierre del archivo
        if (closeTheFileFailed()) {
            codigoError = - 4;
        }
    } else {
        codigoError = - 5;
    }
}
```

La gestión de los errores se vuelve muy difícil

Difícil de administrar las vueltas de funciones

El código se vuelve cada vez más consiguiente

Excepción : para o contra ?

➤ El mecanismo de excepción permite

- ❑ La concisión
- ❑ La legibilidad

```
void leerArchivo() {
    try {
        // Abrir el archivo
        // Determine la longitud del archivo
        // Verificación del asignación de la memoria
        // Leer el archivo en memoria
        // Cerrar el archivo
    } catch (FileOpenFailed) {
        ...
    } catch (FileSizeFailed) {
        ...
    } catch (MemoryAllocFailed) {
        ...
    } catch (FileReadFailed) {
        ...
    } catch (FileCloseFailed) {
        ...
    }
}
```

i

Preferir este solución a la precedente. Programación limpia y profesional

Excepción : las excepciones corrientes

- Java proporciona numerosas clases predefinidas derivadas de la clase `Exception`
- Estas excepciones estándares se clasifican en dos categorías
 - ❑ Las excepciones explícitas (aquellas que estudiamos), mencionadas por la palabra clave **throws**
 - ❑ Las excepciones implícitas que no están mencionadas por la palabra clave **throws**
- Lista de algunas excepciones
 - ❑ *ArithmeticException* (división por cero)
 - ❑ *NullPointerException* (referencia no construida)
 - ❑ *ClassCastException* (problema de cast)
 - ❑ *IndexOutOfBoundsException* (problema de rebasamiento de índice de tabla)

Los flujos

- Para obtener datos, un programa abre un flujo de datos sobre un fuente de datos (archivo, teclado, memoria, etc.)
- De la misma manera para escribir datos en un archivo, un programa abre un flujo de datos
- Java proporciona un package *java.io* que permite de administrar los flujos de datos de entrada y de salida, en forma de caracteres (ejemplo archivos texto) o en forma binaria

Los flujos

- En Java, el numero de clases de manipulación des flujos es importante (mas que 50)
- Java proporciona cuatro jerárquicas de clases para administrar los flujos de datos
 - Para los flujos binarios :
 - La clase *InputStream* y su sub clases para leer octetos (*FileInputStream*)
 - La clase *OuputStream* y su sub clases para escribir octetos (*FileOuputStream*)
 - Para los flujos de caracteres :
 - La clase *Reader* y su sub clases para leer caracteres (*BufferedReader, FileReader*)
 - La clase *Writer* y su sub clases (*BufferedWriter, FileWriter*)

Los flujos de caracteres

➤ Ejemplo : escribir texto en un archivo

FileWriter herede de *Writer* y permite de manipular un flujo texto asociado a un archivo

```
public class PruebaIO {  
  
    public static void main(String[] argv) {  
  
        FileWriter myFile = new FileWriter("a_escribir.txt");  
  
        myFile.write("Ahí es la primera línea de un archivo");  
  
        myFile.close();  
  
    }  
}
```

Cierre del flujo *myFile* hacia el archivo *a_escribir.txt*

Escritura de una línea de texto en un archivo *a_escribir.txt*

Los flujos de caracteres

➤ Ejemplo : leer la entrada estándar

```
public class PruebaIO {
    public static void main(String[] argv) {
        System.out.println("Ingresan su nombre :");

        String inputLine = " ";
        try {
            BufferedReader is = new BufferedReader(new InputStreamReader(System.in));
            inputLine = is.readLine();

            is.close();
        } catch (Exception e) {
            System.out.println("Interceptado : " + e);
        }

        if (inputLine != null)
            System.out.println("Su nombre es :" + inputLine);
        }
    }
}
```

“Convierte” un objeto de tipo *InputStream* en *Reader*

Lee la línea hasta el próximo Enter

```
Output - CursoJavaSE (run-single)
Ingresan su nombre :
Juan Carlos
Su nombre es :Juan Carlos
BUILD SUCCESSFUL (total time: 8 seconds)
```

Cadena ingresada

Los flujos de caracteres

➤ Ejemplo : copia de archivo utilizando los caracteres

FileReader et *FileWriter* heredan de *Reader* y *Writer* y permiten de manipular un flujo texto asociado a un archivo texto

```
public class PruebaIO {
    public static void main(String[] argv) {
        FileReader in = new FileReader("a_leer.txt");
        FileWriter out = new FileWriter("a_escribir.txt");
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
        in.close();
        out.close();
    }
}
```

Transferencia de datos hasta que *in* no proporcione nada

Cierre de flujos y los archivos respectivos

Los flujos binarios

➤ Ejemplo : copia de archivo utilizando los binarios

Mismo razonamiento que para los caracteres excepto...

```
public class PruebaIO {  
    public static void main(String[] argv) {  
        FileInputStream in = new FileInputStream("a_leer.txt");  
        FileOutputStream out = new FileOutputStream("a_escribir.txt");  
        int c;  
  
        while ((c = in.read()) != -1) {  
            out.write(c);  
        }  
  
        in.close();  
        out.close();  
    }  
}
```

La clase File

- Java dispone de una clase *File* que ofrece funcionalidades de gestión de archivos
- La creación de un objeto de tipo *File*

```
File miArchivo = new File("cosa.dat");
```



Atención : no confundir la creación del objeto con la creación de archivo físico

File
- name : String
+ File(String nf)
+ createNewFile()
+ delete() : boolean
+ exists() : boolean
+ getName() : String
+ isFile() : boolean
+ ...

Creación del archivo que el nombre de *name*

Verifica si el archivo existe físicamente

```
File miArchivo = new File("c:\toto.txt");
if (miArchivo.exists()) {
    miArchivo.delete();
} else {
    miArchivo.createNewFile();
}
```

Las colecciones

➤ Por el momento hemos estudiado la tabla para estructurar los datos

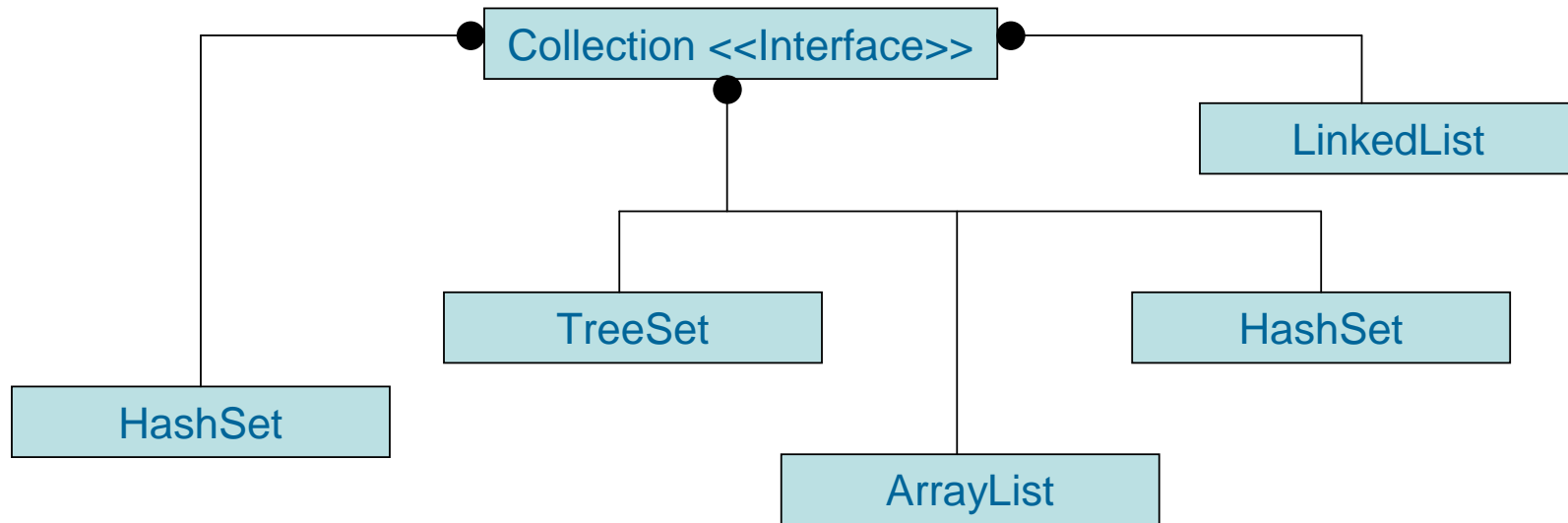
- ❑ Tamaño estático
- ❑ Lente para la búsqueda de elementos particulares
- ❑ Imposibilidad de utilizar un pattern de desplazamiento en los elementos

➤ Java propone desde la versión 2 de clases permitiendo de manipular las principales estructuras de datos

- ❑ Las tablas dinámicas implementadas por *ArrayList* y *Vector*
- ❑ Las listas implementadas por *LinkedList*
- ❑ Los conjuntos implementados por *HashSet* y *TreeSet*

Las colecciones

- Estas clases implementan indirectamente una misma interface Collection que completan de funcionalidades propias



- Desde la versión 5 de Java, posibilidad de utilizar los genéricos para caracterizar el contenido de las Collection
 - ❑ Antes : Carro miCarro = (Carro)myList.get(2)
 - ❑ Ahora : Carro miCarro = myList.get(2)



Ahora no problema de conversión explícita

Las colecciones

➤ El interfase **Collection** permite

- ❑ *Los generics y referencias* : posibilidad de almacenar elementos de tipo cualquier, por poco que se trata de objetos. Un nuevo elemento introducido en una colección Java es una referencia al objeto y no una copia
- ❑ *Los iterator* : permiten recorrer uno a uno los distintos elementos de una colección
- ❑ Eficacia de las operaciones sobre colecciones
- ❑ Operaciones comunes a todas las colecciones : las colecciones que vamos a estudiar implementan todas al mínimo el interfase **Collection**, de manera que disponen de funcionalidades comunes

Las colecciones : los generics Java

- Con la versión 6 de Java posibilidad de explotar los generics en las colecciones y por otros aspectos del lenguaje también

- Se añadió una sintaxis particular de manera a tener en cuenta los generics
 - `< ? >` : indique que es necesario precisar el tipo de la clase
 - `< ? , ? >` : indique que es necesario precisar dos tipos

- Con los generics, va a ser posible fijar en la construcción de la colección el tipo del contenido almacenado en las colecciones

- Ventajas
 - Todos los métodos asesores y modificadores que manipulan los elementos de una collection son *firmados* según el tipo definido a la construcción de la collection
 - Verificación de los tipos durante el desarrollo (antes problema de `CastClassException`)

Las colecciones : Iterator

➤ Los iterator permiten de recorrer los elementos de una collection Sin conocimiento precisa del tipo de la collection : polimorfismo

➤ Existe dos familias de iterator :

□ *monodireccionales*

El curso de la colección se hace de un inicio hacia un fin; se pasa una única vez sobre cada uno de los elementos

□ *bidireccionales*

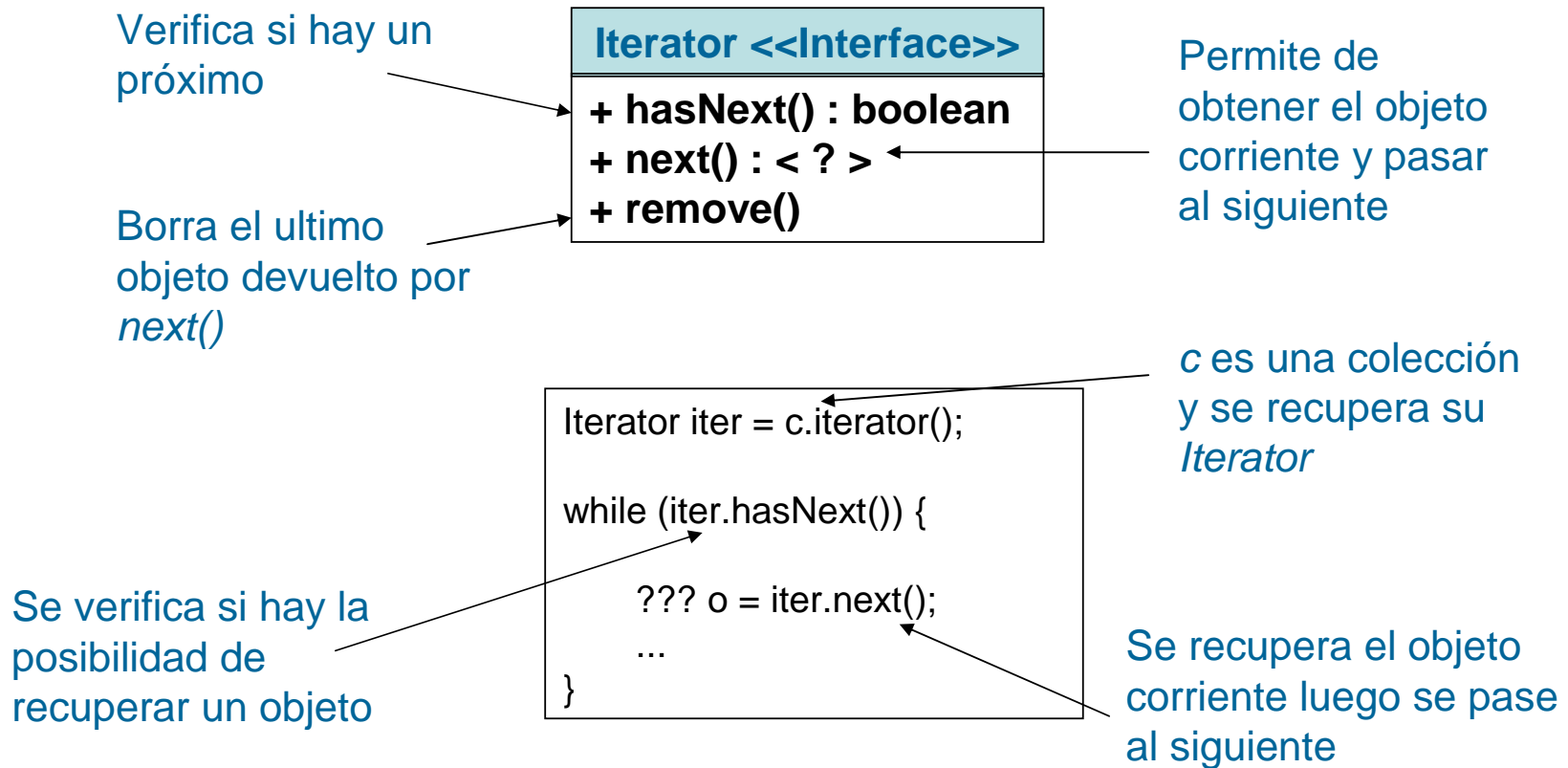
El curso de la colección puede hacerse en las dos direcciones; se puede avanzar y retroceder a su manera en la collection

i La noción de Iterator es parte del conjunto de los Design Patterns

Las colecciones : Iterator

➤ Iterator monodireccionales : interface *Iterator*

- Por defecto, todas las colecciones tienen un atributo de tipo *Iterator*



Las colecciones : Iterator

➤ Iterator bidireccionales : interfase *ListIterator*

- ❑ Conciernen las listas y tablas dinámicas
- ❑ Permite de añadir o borrar objetos

Iterator <<Interface>>

ListIterator <<Interface>>

- + previous() : < ? >
- + hasPrevious() : boolean
- + add(< ? >) ←
- + set(< ? >) ←
- + ...

Verifica si hay un precedente

Se verifica si hay la posibilidad de recuperar un objeto anteriormente

Recupera el objeto anteriormente luego se pasa al precedente

Añade o modifica a la posición corriente un elemento de la colección

```
Iterator iter = c.listIterator();  
while (iter.hasPrevious()) {  
    ??? o = iter.previous();  
    ...  
}
```

c es una colección y se recupera su *ListIterator*
Inicializa en inicio de lista

Las colecciones : LinkedList

➤ Este clase permite de manipular listas dichas « doblemente encadenadas ».

A cada elemento de colección, se asocia implícitamente dos informaciones que son las referencias al elemento precedente y siguiente



Nada mas después de estos elementos, se hace una vuelta detrás

```
LinkedList<String> l1 = new LinkedList<String>();  
ListIterator iter = l1.listIterator();  
  
iter.add("Buenos dias");  
iter.add("Hola");  
  
while(iter.hasPrevious()) {  
    String o = iter.previous();  
    System.out.println(o);  
}
```

Añade de elementos a través del iterator
La utilización de la *LinkedList* es transparente

Las colecciones : LinkedList

- Posibilidad de utilizar las colecciones (en este caso LinkedList es un ejemplo) sin los iterator pero menos potente !!!

```
LinkedList<String> l1 = new LinkedList<String>();  
l1.add("Buenos dias");  
l1.add("Hola");  
for (int i = 0; i < l1.size(); i++) {  
    String o = l1.get(i);  
    System.out.println(o);  
}
```

Utilización
del
método
add de la
clase
LinkedList

La utilización
de la
LinkedList
No es
transparente.
Conocimiento
obligatorio de
esto métodos



**No modificar la collection (*add*
de *LinkedList*) mientras que
se utiliza el iterator (*next()*)**

Las colecciones : ArrayList

- La clase *ArrayList* es una encapsulación de la tabla con la posibilidad de volverlo dinámico en tamaño
- Posibilidad de utilizar *ListIterator* pero se prefiere su utilización a un elemento de rango dado

```
ArrayList<Object> myArrayList = new ArrayList<Object>();

myArrayList.add("Hola");
myArrayList.add(34);

for (int i = 0; i < myArrayList.size(); i++) {
    Object myObject = myArrayList.get(i);
    if (myObject instanceof String) {
        System.out.println("Cadena:" + ((String)myObject));
    }

    if (my_object instanceof Integer) {
        System.out.println("Integer:" + ((Integer)myObject));
    }
}
```

i Preferir la utilización de la clase *ArrayList* al lugar de la clase *Vector*

Las colecciones : HashSet

- La clase *HashSet* permite de administrar los conjuntos. Dos elementos no pueden ser idénticos
- Es necesario prever dos cosas en sus clases :
 - La redefinición del método *hashCode()* que esta utilizada para organizar los elementos de un conjunto (calculo de la hash table de un objeto)
 - La redefinición del método *equals(Object)* que compara objetos de misma clase para conocer la pertenencia de un elemento al conjunto

```
public class PruebaHashSet {
    public static void main(String[] argv) {
        Punto p1 = new Punto(1,3), p2 = new Punto(2,2);
        Punto p3 = new Punto(4,5), p4 = new Punto(1,8);
        Punto p[] = {p1, p2, p1, p3, p4, p3};

        HashSet<Punto> conj = new HashSet<Punto>();
        for (int i = 0; i<p.length; i++) {
            System.out.println("El Punto ") ; p[i].visualizar();
            boolean añade = conj.add(p[i]);
            if (añade) System.out.println(" se ha añadido");
            else System.out.println("ya esta presente");
            System.out.print("Conjunto = "); visualizar(conj);
        }
    }
}
```

```
public static void visualizar(HashSet conj) {
    Iterator iter = conj.iterator();
    while(iter.hasNext()) {
        Punto p = iter.next();
        p.visualizar();
    }
    System.out.println();
}
```


Las colecciones : HashSet

```
public class Punto {
    private int x,y;

    Punto(int x, int y) {
        this.x = x; this.y = y;
    }
    public int hashCode() {
        return x+y;
    }
    public boolean equals(Object pp) {
        Punto p = (Punto)pp;
        return ((this.x == p.x) &
            (this.y == p.y));
    }
    public void visualizar() {
        System.out.print("[ " + x + " " + y + " ] ");
    }
}
```

Redefinición de los métodos *hashCode()* y *equals(Object)*

Output - CursoJavaSE (run-single)

```
run-single:
El Punto [1 3] se ha añadido
Conjunto = [1 3]
El Punto [2 2] se ha añadido
Conjunto = [2 2] [1 3]
El Punto [1 3] ya esta presente
Conjunto = [2 2] [1 3]
El Punto [4 5] se ha añadido
Conjunto = [2 2] [1 3] [4 5]
El Punto [1 8] se ha añadido
Conjunto = [2 2] [1 3] [1 8] [4 5]
El Punto [4 5] ya esta presente
Conjunto = [2 2] [1 3] [1 8] [4 5]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Conclusión

➤ Durante este formación

- Se familiarizaron con la programación a objetos utilizando Java
- Aprendieron a escribir aplicaciones Java autónomos
- Descubrieron las clases Java más importantes
- Abordaron clases de API

➤ Lo que queda a hacer

- Durante este formación, no hicimos más que estudiar la superficie del lenguaje Java
- ... pero tienen mas a hacer estudiando la parte conexión a base de datos (JDBC) y el aspecto WEB de Java (JSP, Servlet, Tomcat...) !

Gracias por su atención ! Merci pour votre attention !