

Introducción a la Programación Orientada a Objetos en Java

Contenido

- Conceptos básicos de la P. O. O.
 - Clases y objetos
 - Métodos y mensajes
 - Creación de objetos
 - Uso de objetos. Variables y ámbito
 - Clases Especiales
 - Clases Enumeradas
 - Registros. Tuplas Nombradas
 - Clases anidadas

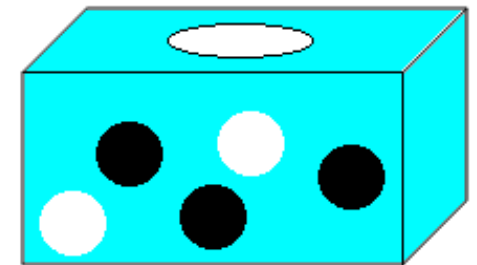
¿Qué es POO?

- Estilo de programación que trata de representar un modelo de la realidad basado en los datos a manipular.
 - Las abstracciones de datos se modelan con **objetos**.
 - Diseño enfocado al cliente. Los objetos se refieren a datos que el cliente entiende porque forman parte de la especificación del problema.

Reutilización de código

Ejemplo: Urna

- Queremos manipular urnas capaces de contener bolas blancas y negras.
 - Utilizando la abstracción, una urna se puede representar por un objeto que contiene dos enteros (que llamaremos **su estado**)
 - nBlancas: int. Número de bolas blancas
 - nNegras: int. Número de bolas negras



Urna (nBlancas:34, nNegras: 16)

Comportamiento de urna

- Una vez fijado el estado de la urna, el cliente especifica cómo operar con la urna.
- Por ejemplo
 - Queremos saber el total de bolas que tiene la urna.
 - Queremos poder introducir una bola del color que queramos (blanco o negro).
 - Queremos sacar aleatoriamente una bola de la urna
 - Queremos saber si la urna está vacía.
 - Estas operaciones definen lo que llamaremos **el comportamiento** de la urna.

Ejemplo. Jarra

- Queremos manipular jarras que tienen una capacidad y un contenido (siempre en litros):
 - capacidad: int. Lo que cabe en la jarra
 - contenido: int. Lo que actualmente tiene la jarra
- Una jarra la representamos como un objeto con dos enteros, uno para la capacidad y otro para el contenido (**su estado**)

Jarra(capacidad:7, contenido:3)



Jarra. Comportamiento

- Queremos poder llenar la jarra desde una fuente hasta completarla.
- Queremos poder volcar la jarra en un sumidero hasta vaciarla.
- Queremos poder volcar una jarra sobre otra hasta que la segunda se llene o la primera se vacíe.
- Estas operaciones definen el **comportamiento** de las jarras.

Urnas y Jarras. Conceptos

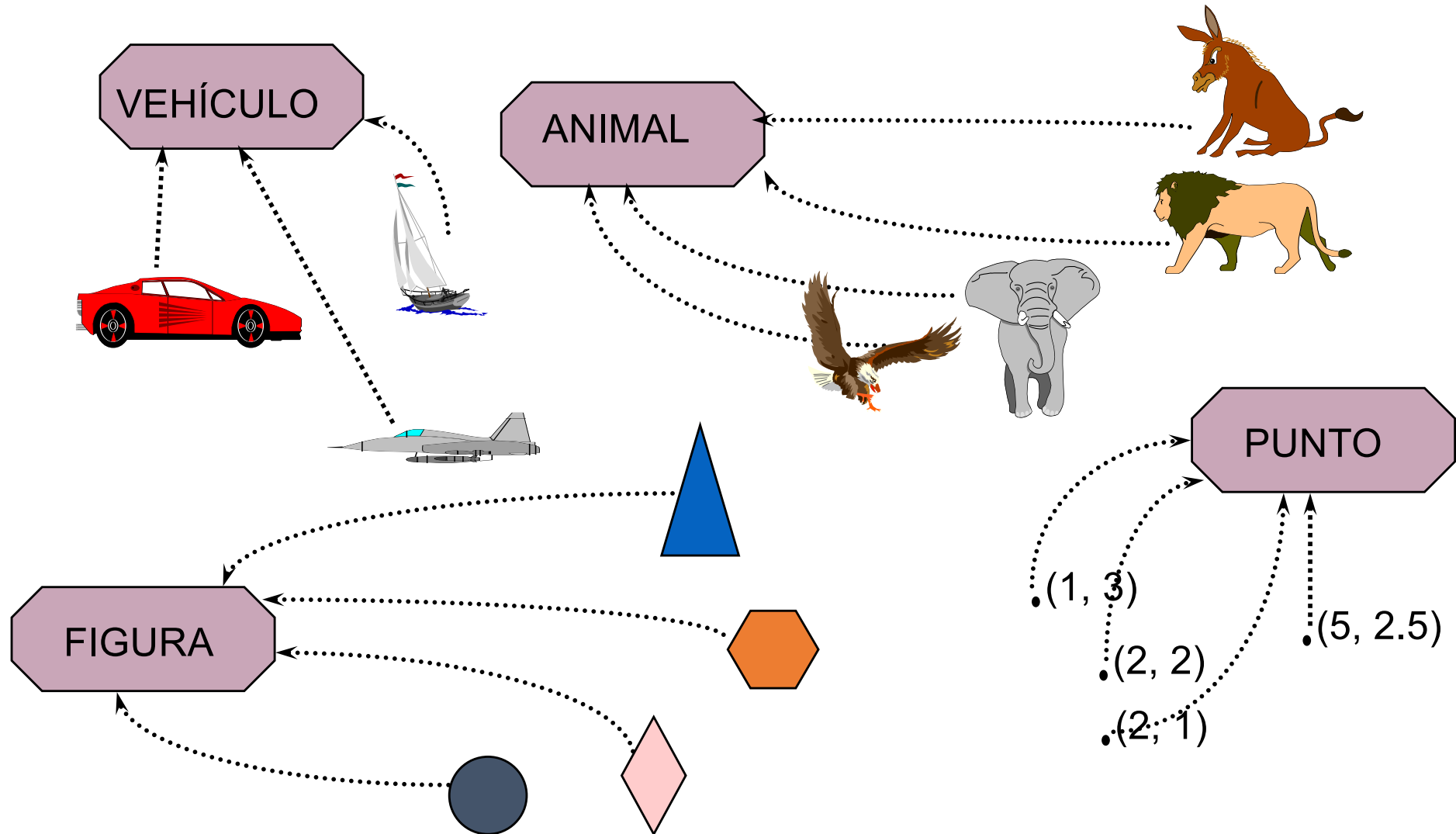
- Cada urna y cada jarra tiene **su propio estado**.
- Podemos disponer de diferentes urnas cada una con su estado.
Urna(3,5) Urna(8,2) Urna(32,17) Urna(9,0)
- Podemos disponer de diferentes jarras cada una con su estado.
Jarra(7, 5) Jarra(8, 8) Jarra(9,0) Jarra(2,1)
- Todas las urnas tienen **el mismo comportamiento**
- Todas las jarras tienen **el mismo comportamiento**.
- Es posible crear una abstracción mayor:
 - Que defina la forma del estado.
 - Que defina el comportamiento.

Paso de Mensajes

- Cuando actuamos sobre un objeto a través de su comportamiento, podemos:
 - Consultar el estado del objeto
 - Modificar el estado del objeto
- Ejemplo: si tenemos la siguiente urna `Urna(3,5)`
 - y le preguntamos por el total de bolas nos devolverá 8.
 - y si le introducimos una bola negra, su estado cambiará a `Urna(3,6)`
- Ejemplo Si tenemos las jarras `Jarra(7, 5)` `Jarra(5,3)`
 - y le preguntamos si están vacías, ambas nos responderán con `false`.
 - y si volcamos la segunda sobre la primera, el estado de ambas jarras se modifica pasando a ser `Jarra(7, 7)` `Jarra(5,1)`
- La manera de actuar sobre un objeto a través de su comportamiento es por medio de **paso de mensajes**.
 - A una urna **se le envía un mensaje** pidiéndole el número total de bolas.
 - A una jarra **se le envía un mensaje** pidiéndole que se rellene con el contenido de otra jarra.

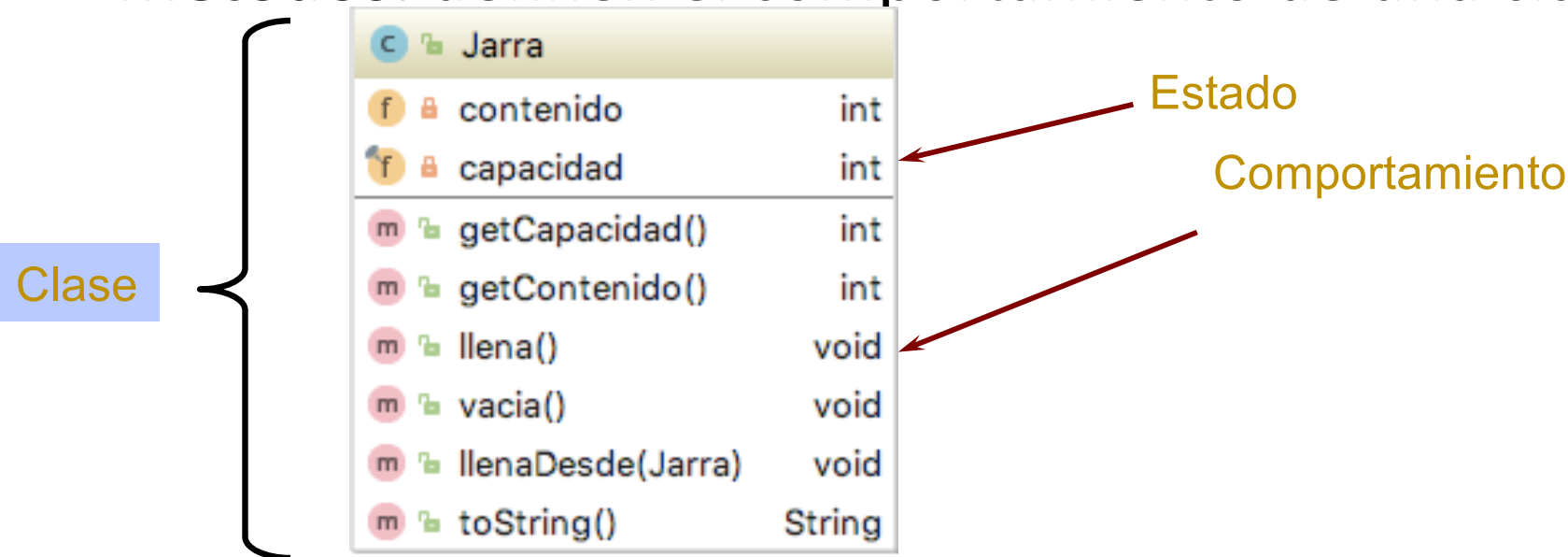
Clases y Objetos

- **CLASE = MÓDULO + TIPO**
 - Criterio de estructuración del código
 - **Forma del estado** + **Comportamiento**
 - Entidad estática (en general)
 - Ej: La clase Urna o la clase Jarra
- **OBJETO = **Instancia** de una CLASE**
 - Objeto (Clase) = Valor (Tipo)
 - Entidad dinámica
 - Cada objeto tiene **su propio estado**
 - Objetos de una clase comparten su comportamiento
 - Urna(nBlancas: 3, nNegras: 5)**
 - Jarra(capacidad: 7, contenido: 2)**

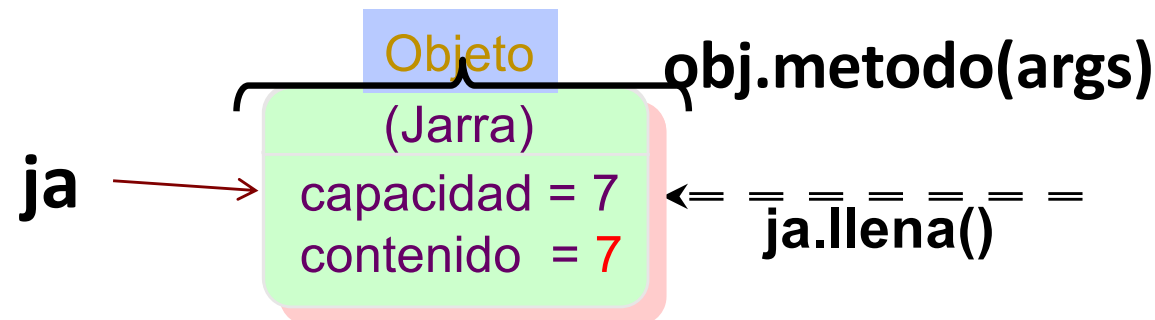


Representación de una clase. Estado, Métodos y mensajes

- **Métodos:** definen el comportamiento de una clase

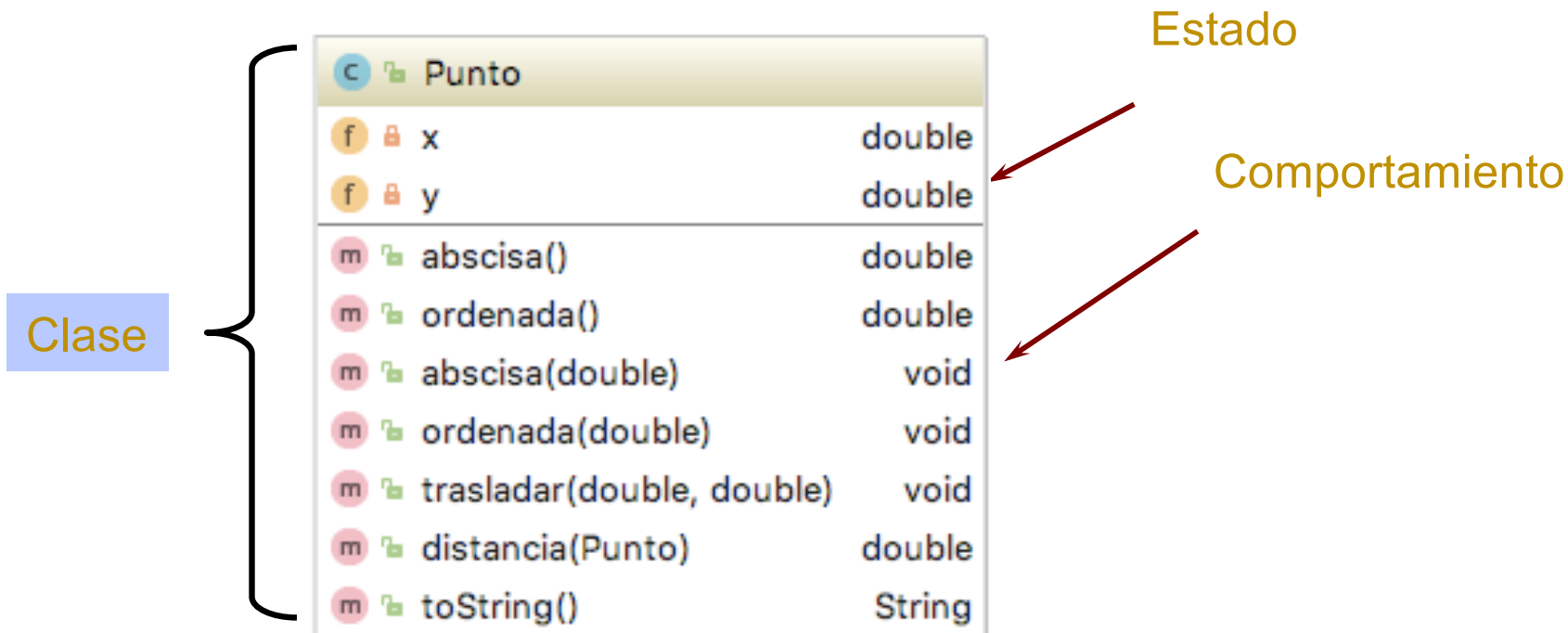


- Invocación de **métodos**: Paso de **mensajes**

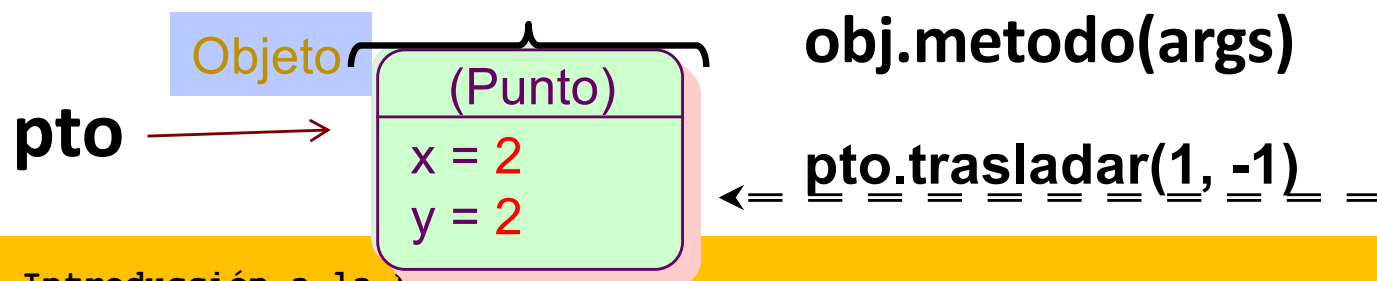


Representación de una clase. Estado, Métodos y mensajes

- **Métodos:** definen el comportamiento de una clase



- Invocación de **métodos**: Paso de **mensajes**



Paso de mensajes

obj.metodo(args)

- Un mensaje involucra a
 - Un receptor del mensaje (el objeto que lo recibe **obj**)
 - Un selector. El comportamiento aludido (**metodo**)
 - Argumentos. Que puede requerir el selector (**args**)
- Los **mensajes** que se envían a un determinado objeto deben tener un selector que “**corresponda**” con algún **método** que la clase tiene definido (con el comportamiento).
- Esta correspondencia se debe reflejar en la **signatura** del método: **nombre**, sus **argumentos** y sus **tipos**.

Clases

- Estructuras que encapsulan *datos y métodos*

“Punto.java”

```
public class Punto {  
    private double x, y;  
    public Punto() { x = y = 0; }  
    public Punto(double a, double b) { x = a; y = b; }  
    public double abscisa() { return x; }  
    public double ordenada() { return y; }  
    public void abscisa(double a) { x = a; }  
    public void ordenada(double b) { y = b; }  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public double distancia(Punto pto) {  
        return Math.sqrt(Math.pow(x - pto.x, 2) +  
            Math.pow(y - pto.y, 2));  
    }  
}
```

VARIABLES DE ESTADO
CONSTRUCTORES

MÉTODOS

Envío de mensajes

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

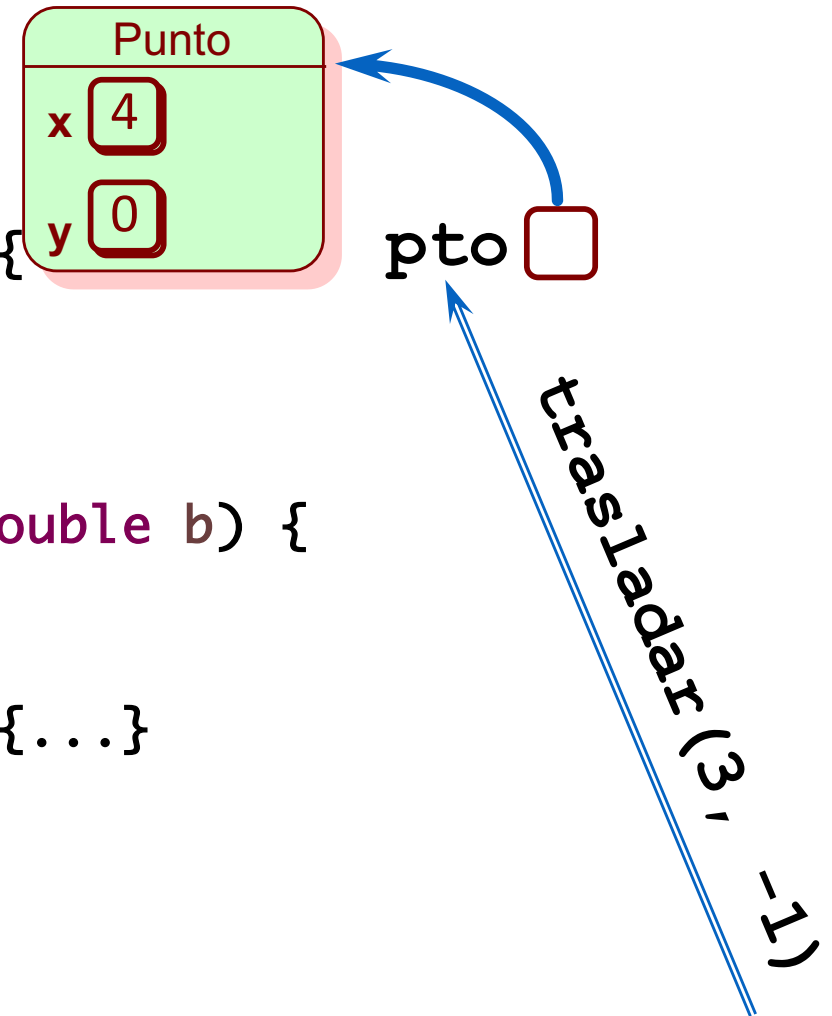
    public Punto(double a, double b) {
        x = a; y = b;
    }

    ...

    public void trasladar(double a, double b) {
        x += a; y += b;
    }

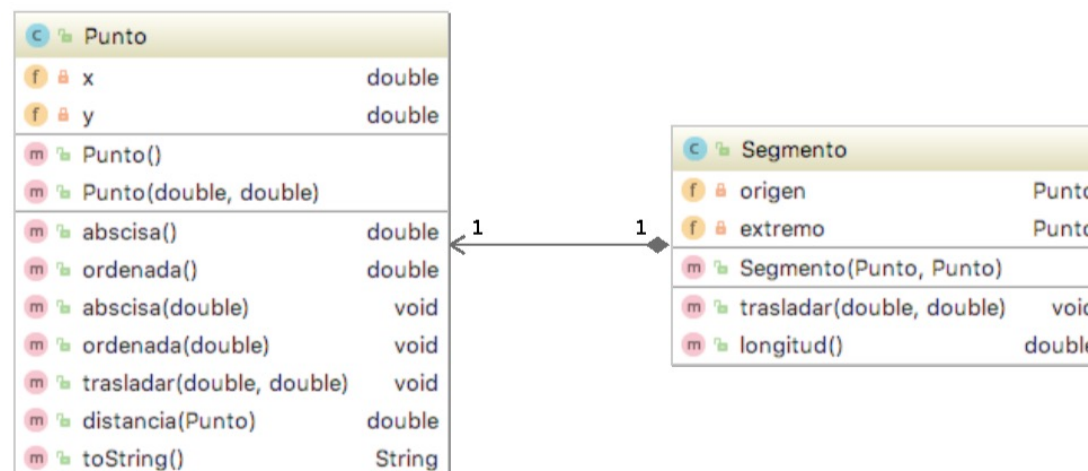
    public double distancia(Punto p) {...}
}
```

```
Punto pto = new Punto(1, 1);
pto.trasladar(3, -1);
```



Composición

- Mecanismo que permite la creación de nuevos objetos a partir de otros ya implementados.
- Responde a una relación del tipo “*tiene*” o “*está compuesto por*”.
- Así, por ejemplo, un segmento **está compuesto por** dos puntos (origen y extremo)
 - También podemos decir que los puntos origen y extremo “**forman parte del**” segmento, o que el segmento “**tiene**” dos puntos.



Composición

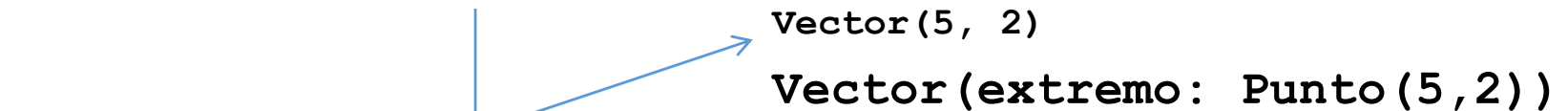
```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new Punto(x1, y1);  
        extremo = new Punto(x2, y2);  
    }  
  
    ... // Otros métodos  
  
    public double longitud() {  
        return origen.distancia(extremo);  
    }  
}
```

Para calcular la longitud de un segmento se utiliza el método distancia de la clase Punto

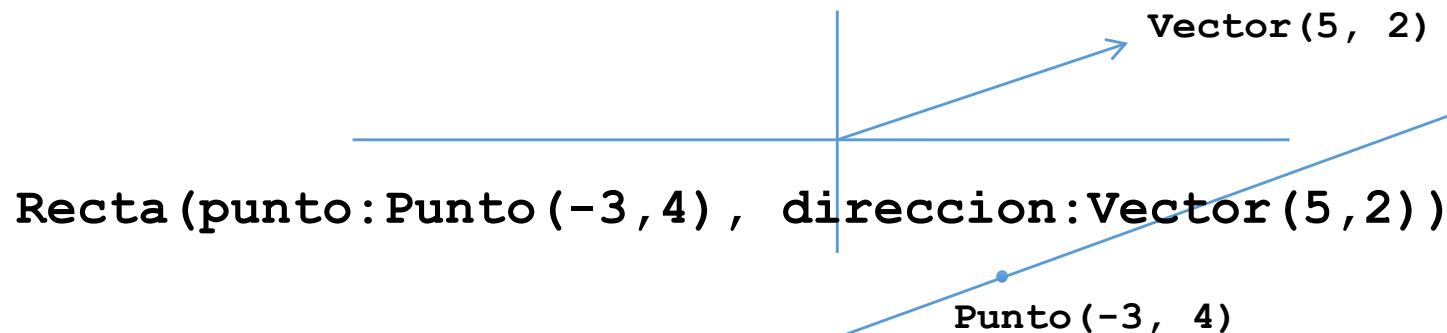
Otros ejemplos de composición

- Otros ejemplos:

- Un vector puede implementarse con un punto que representa el extremo del vector cuando su origen está en el origen de coordenadas.
 - El vector **tiene** un punto.



- Una recta podría implementarse con un vector y un punto. El punto es uno cualquiera de la recta y el vector proporciona la dirección de la misma.
 - La recta **"tiene"** o **"está definida por"** un punto y un vector



Más ejemplos de composición

- La composición de una clase dependerá del uso que se haga con sus objetos.
 - Una persona **tiene** una edad (que se representa con un entero) y un nombre (que se representa con un String).
 - Una persona **tiene** un nombre (que se representa con un String) y un DNI (que se representa con otro String)
 - Una persona **tiene** edad, DNI, **curso**, **asignaturas**, etc.
- Un coche **tiene** un modelo (que se representa por un String) y un precio (que se representa con un float)
- Un coche **está compuesto por** un **motor**, 4 **puertas**, 4 **ruedas**, **chasis**, etc.

Programa en Java

- Conjunto de clases

- diseñadas para colaborar en una tarea,
- con una clase distinguida que contiene un método de clase:

```
public static void main(String[] args)
```

que desencadena la ejecución del programa.

- Las demás clases pueden estar definidas *ad hoc* o pertenecer a una biblioteca de clases.

Ficheros de clases en Java

- **Cada clase declarada como pública** debe estar en un fichero con extensión **.java** y con nombre el de la clase.

`Urna.java` `Jarra.java` `Punto.java`

- Cada fichero **.java** puede contener varias clases **pero sólo una podrá ser pública** (la que da nombre al fichero).
- Cada fichero **.java** debe precompilarse generando un fichero **.class** (en *bytecodes*) por cada clase contenida en él.

```
javac Urna.java
```

- Un programa se ejecuta pasando el fichero **.class** de la clase distinguida al intérprete (máquina virtual de Java)

```
java Urna
```

Ejecución de un programa

```
public class HolaMundo {  
    public static void main(String[] args) {  
        System.out.println("Hola Mundo");  
    }  
}
```

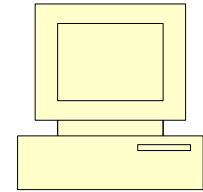
HolaMundo.java

javac

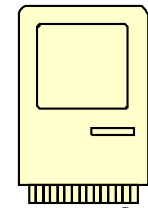
Bytecodes

HolaMundo.class

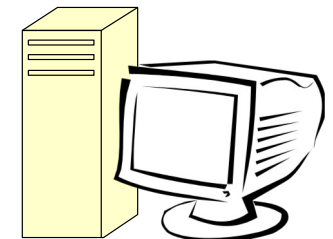
java



Windows



MacOS



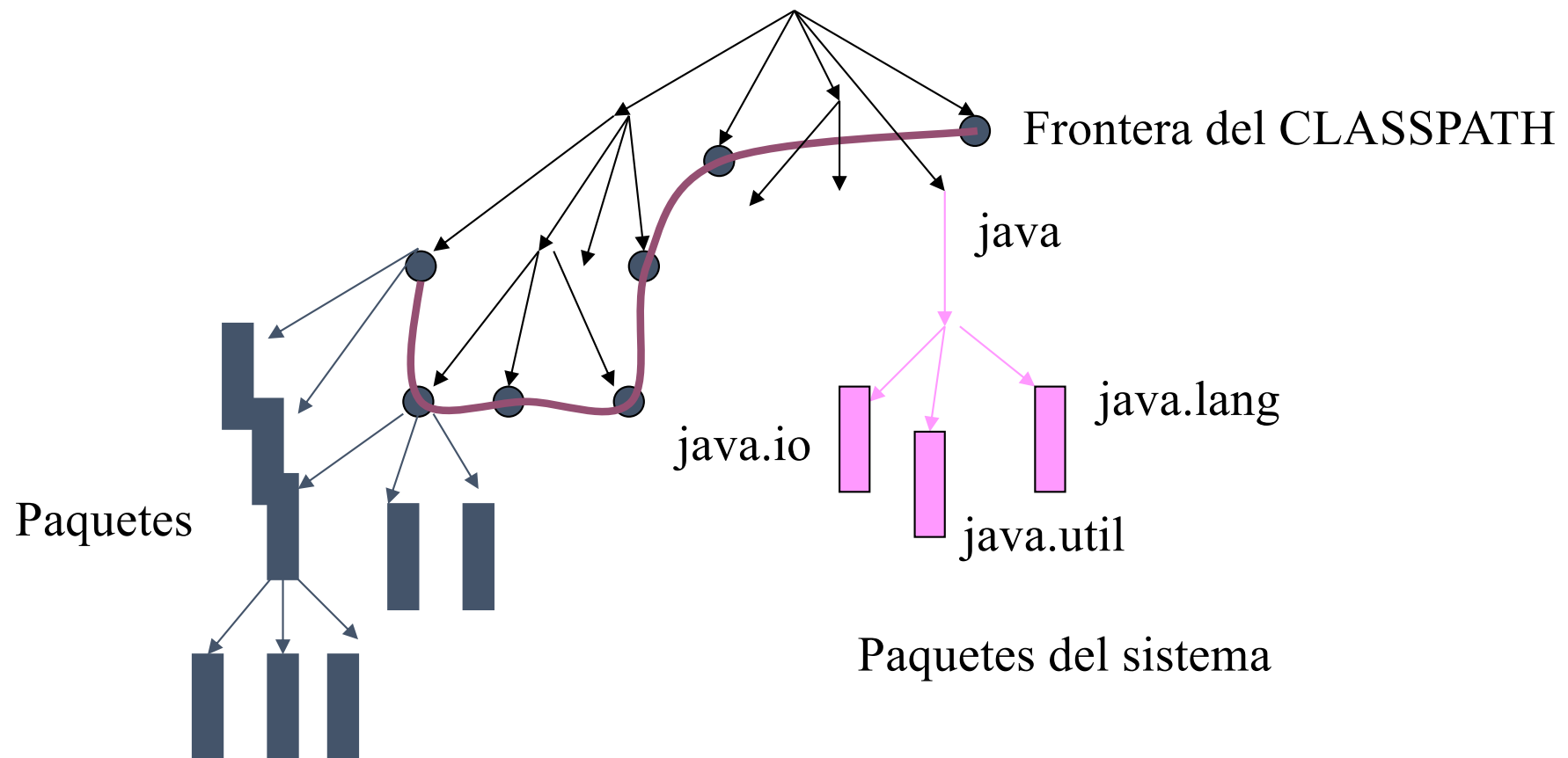
Linux

```
$ ls  
HolaMundo.java  
$ javac HolaMundo.java  
$ ls  
HolaMundo.java  
HolaMundo.class  
$ java HolaMundo  
Hola Mundo  
$
```

Paquetes

- Las clases se organizan en ***paquetes* (package)**: **mecanismos lógicos para agrupar clases relacionadas.**
- Todas las clases de un paquete deben estar localizadas en un **mismo subdirectorio.**
- Los paquetes del sistema están situados en subdirectorios específicos:
 - .../java
 - .../javax
- La variable **CLASSPATH** contiene una lista con todos los caminos de búsqueda de los paquetes donde hay clases que se necesitan.

Estructura de paquetes Java



Ejemplo: `org.uma.mbd.mdJarras.jarras`

Paquetes básicos del sistema

- `java.lang`: para funciones del lenguaje
- `java.util`: para utilidades adicionales
- `java.io`: para entrada y salida
- `java.nio`: para entrada y salida
- `java.text`: para formato especializado
- `java.awt`: para diseño gráfico e interaz de usuario
- `java.awt.event`: para gestionar eventos
- `javax.swing`: nuevo diseño de GUI
- `java.net`: para comunicaciones
- ...

Acceso a las bibliotecas de Java

- El nombre de cada paquete debe coincidir con el camino que va desde algún directorio del **CLASSPATH** (o desde **/java** o **/javax**) al subdirectorio correspondiente al paquete.
- A las clases incluidas en **java.lang** se puede acceder simplemente por sus nombres, p.e.: **System** o **Math**.
- Las clases de un paquete (salvo las de **java.lang**) sólo se pueden acceder por sus nombres cualificados o a través de una cláusula **import**;

Ejemplo cualificando

- Programa para calcular el valor medio de un millón de números generados aleatoriamente, usando las clases
 - **Random** del paquete **java.util**
 - **System** del paquete **java.lang**

```
public class TestAleatorio {  
    public static void main(String[] args) {  
        java.util.Random rnd = new java.util.Random();  
        double sum = 0.0;  
        for (int i = 0; i < 1000000; i++)  
            sum += rnd.nextDouble();  
        System.out.println("media = " + sum / 1000000.0);  
    }  
}
```

Ejemplo con import

- Programa para calcular el valor medio de un millón de números generados aleatoriamente, usando las clases
 - `Random` del paquete `java.util`
 - `System` del paquete `java.lang`

```
import java.util.Random;
public class TestAleatorio {
    public static void main(String[] args) {
        Random rnd = new Random();
        double sum = 0.0;
        for (int i = 0; i < 1000000; i++)
            sum += rnd.nextDouble();
        System.out.println("media = " + sum / 1000000.0);
    }
}
```

Ejemplo con import static

```
import static java.lang.Math.*;
```

```
public class Punto {  
    private double x, y;  
    public Punto() { x = y = 0; }  
    public Punto(double a, double b) {  
        x = a; y = b;  
    }  
    public double abscisa() { return x; }  
    public double ordenada() { return y; }  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public void abscisa(double a) { x = a; }  
    public void ordenada(double b) { y = b; }  
    public double distancia(Punto pto) {  
        return sqrt(pow(x - pto.x, 2) + pow(y - pto.y, 2));  
    }  
}
```

Se accede sin
cualificar a todos los
métodos static de
Math

Creación de objetos

- Cuando un objeto se crea:
 - Se le reserva espacio de memoria
 - Se les asigna valor inicial a sus variables de estado

- Para crearlo se debe utilizar un constructor:

`new <constructor>(<lista args>)`

- **new** devuelve una referencia al objeto que crea.

- Puede asignarse a una variable

`pto = new Punto(3, 4);`

- Puede usarse en una expresión

`pto.distancia(new Punto(2, 3));`

Constructores de objetos

- Una clase puede definir varios constructores
 - Con **distinto número de argumentos** o
 - Con **argumentos de distintos tipos**.

```
public Punto() {  
    x = 0;  
    y = 0;  
}  
public Punto(double a, double b) {  
    x = a;  
    y = b;  
}
```

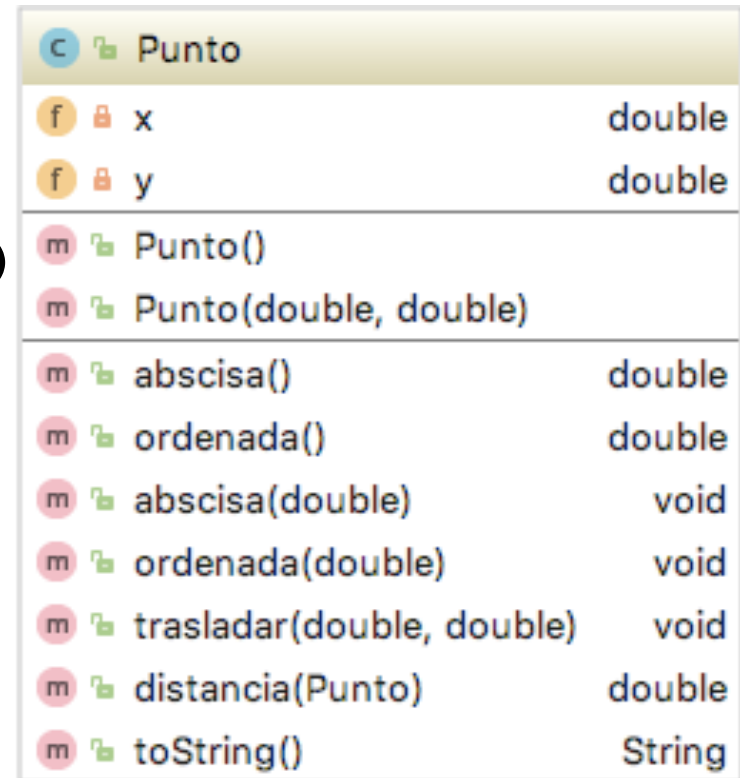
- Si no hay ningún constructor entonces el sistema proporciona uno **“por defecto”**
 - Si hay constructores, el “constructor por defecto” no se crea.

Constructores de objetos

- Un constructor puede invocar a otro de la misma clase:

```
public Punto() {  
    this(0,0);  
}
```

```
public Punto(double a, double b)  
    x = a;  
    y = b;  
}
```



Punto	
f	x double
f	y double
m Punto()	
m Punto(double, double)	
m	abscisa() double
m	ordenada() double
m	abscisa(double) void
m	ordenada(double) void
m	trasladar(double, double) void
m	distancia(Punto) double
m	toString() String

Variables de instancia y de clase

- Las **variables de instancia** (atributos o estado)
 - Cada **instancia (objeto)** tiene sus propias **variables de instancia (estado)**.
 - Se acceden etiquetándolas con el **nombre o la referencia de la instancia**.
 - Un punto tiene dos variables de instancia: x, y
 - Una urna tiene dos variables de instancia: nBlancas y nNegras
 - Una jarra tiene dos variables de instancia: capacidad y contenido
 - Un segmento tiene dos variables de instancia: origen y extremo
 - Un vector tiene una variable de instancia: extremo
 - Una recta tiene dos variables de instancia: punto y direccion
- Las **variables de clase**
 - Son **comunes a todos los objetos** de la clase.
 - Se declaran como **static**.
 - Se acceden/invocan etiquetando sus nombres **con el nombre de la clase** cuando son visibles (y también con **el nombre de alguna instancia** aunque está **desaconsejado**).

Métodos de instancia

- Los métodos de instancia **se invocan** mediante **mensajes** contruidos precediendo el **nombre del método** con **el nombre o la referencia de la instancia**.
- Un **método de instancia** **tiene acceso** a las **variables de instancia propias** (del receptor). Puede usarse **this** para cualificarlas.
- También puede acceder a las **variables de instancia** de **cualquier objeto de la misma clase**. Se cualifican con el nombre.

```
public double distancia(Punto pto) {  
    return Math.sqrt(Math.pow(x - pto.x, 2) +  
        Math.pow(y - pto.y, 2));  
}
```

```
Punto pto = new Punto(3,5);  
pto.distancia(new Punto(1,4));
```

Otro objeto de
la misma clase

Propias del receptor
Puede usarse **this.x**

Propiedades

- Como norma general, en Java

- Si se quiere dar acceso a una variable de instancia se implementa un método **getter**, es decir un método cuyo nombre comienza con **get** y le sigue el nombre de la variable. Si la variable es boolean puede cambiarse **get** por **is**.
- Si se quiere dar acceso de escritura a una variable de instancia se define un **setter**, es decir, un método que comienza con **set** y le sigue el nombre de la variable.

Jarra		
U	contenido	int
U	capacidad	int
m	Jarra(int)	
U	getCapacidad()	int
U	getContenido()	int
m	llena()	void
m	vacía()	void
m	llenaDesde(Jarra)	void
m	toString()	String

Jarra		
m	Jarra(int)	
m	llena()	void
m	vacía()	void
m	llenaDesde(Jarra)	void
m	toString()	String
p	capacidad	int
p	contenido	int

Métodos de clase

- Los métodos de clase
 - Sólo tienen acceso a las variables de clase.
 - Se declaran como **static**.
 - Se acceden/invocan etiquetando sus nombres con el nombre de la clase cuando son visibles (y también con el nombre de alguna instancia aunque está desaconsejado).

```
public double distancia(Punto pto) {  
    return Math.sqrt(Math.pow(x - pto.x, 2) +  
                        Math.pow(y - pto.y, 2));  
}
```

Métodos de clase (static)
de la clase Math

Métodos y variables de clase

```
public class Vuelo {  
    static private int sigVuelo = 1;    // De clase  
    private String localizadorVuelo;    // De instancia
```

```
    static private int nuevoIdentificador() {  
        // De clase  
        int vuelo = sigVuelo;  
        sigVuelo++;  
        return vuelo;  
    }
```

```
    public Vuelo(String lin) {  
        localizadorVuelo = lin + "_" + Vuelo.nuevoIdentificador();  
    }
```

```
    ...
```

```
}
```

Vuelo		
f	sigVuelo	int
f	localizadorVuelo	String
Vuelo(String)		
m	nuevoIdentificador()	int
m	toString()	String

```
Vuelo v1 = new Vuelo("Iberia");    // Iberia_1  
Vuelo v2 = new Vuelo("Lufhtansa"); // Lufhtansa_2  
Vuelo v3 = new Vuelo("Iberia");    // Iberia_3
```

Métodos factoría

```
public class VueloF {  
    static private int sigVuelo = 1;    // De clase  
    private String localizadorVuelo;    // De instancia  
    static private int nuevoIdentificador() {  
        // De clase  
        int vuelo = sigVuelo;  
        sigVuelo++;  
        return vuelo;  
    }  
    private VueloF(String lin) {  
        localizadorVuelo = lin + "_" + VueloF.nuevoIdentificador();  
    }  
    static public VueloF creaVuelo(String lin) { // Método factoría (de clase)  
        return new VueloF(lin);  
    }  
    ...  
}
```









VueloF		
f	sigVuelo	int
f	localizadorVuelo	String
Methods		
m	VueloF(String)	
m	nuevoIdentificador()	int
m	creaVuelo(String)	VueloF
m	toString()	String

```
VueloF v1 = VueloF.creaVuelo("Iberia");    // Iberia_1  
VueloF v2 = VueloF.creaVuelo("Lufhtansa"); // Lufhtansa_2  
VueloF v3 = VueloF.creaVuelo("Iberia");    // Iberia_3
```

Control de la visibilidad

Existen cuatro niveles de visibilidad. Los tres primeros son:

- **private** – visibilidad dentro de la propia clase
- **public** – visibilidad desde cualquier paquete
- Por omisión – visibilidad dentro del propio paquete (package)

			Mismo paquete	Otro paquete	
			Otra Clase	Otra Clase	
		-	private	NO	NO
		+	public	SÍ	SÍ
		~		SÍ	NO

La vida de los objetos

- Los objetos **son siempre** **instancias** de alguna clase.
- Se deben **crear** por medio de un constructor.
- Durante la ejecución de un programa
 - Se crean objetos
 - Interactúan entre ellos por medio del envío de mensajes.
 - Se eliminan los objetos no necesarios
 - La eliminación es automática.

Variables que referencian a objetos

- Las variables se declaran de una clase (o interfaz)

```
Punto pto;
```

Declaración

`pto`

- Es una referencia a un objeto, **NO** es un objeto
- **No puede recibir aún mensajes.**

- Una variable puede referenciar a un objeto (Asignación)

```
pto = new Punto(3, 4);
```

- **Ya puede recibir mensajes.**

Asignación

- Estos dos pasos se pueden realizar simultáneamente:

```
Punto pto = new Punto(3, 4);
```

inicialización

Uso de objetos

- A las variables de estado de un objeto se accede de la forma `pto.x`, siempre que sea visible:

- A las de instancia dentro de un método con `this.x`

```
public double distancia(Punto pto) {  
    return Math.sqrt(Math.pow(this.x - pto.x, 2) +  
        Math.pow(this.y - pto.y, 2));  
}
```

“El acceso directo a las variables de estado de un objeto por parte de otro de otra clase no es aconsejable”.

- Si no hay conflicto de nombres, `this` puede suprimirse.

```
public double distancia(Punto pto) {  
    return Math.sqrt(Math.pow(x - pto.x, 2) +  
        Math.pow(y - pto.y, 2));  
}
```

Se supone que un objeto debe proteger su estado.

Uso de objetos

- Invocación de los métodos
 - Los métodos cuando son visibles, se invocan mediante la forma: *objeto.mensaje(argumentos)*

```
Punto pto = new Punto(3, 4);  
pto.trasladar(2, 2);
```

```
double d = pto.distancia(new Punto(1,2));
```

Asignación o Copia

- Asignación.
 - Comparte objeto.
- Copia
 - Definir un constructor de copia
 - Como argumento tiene otro objeto de la misma clase
 - Incluir un método de copia en la clase correspondiente.
 - Utilizar el método **public protected clone()**
 - Copia superficial

Uso de objetos (asignación)

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(Punto pto1, Punto pto2) {  
        origen = pto1;  
        extremo = pto2;  
    }  
    ... // Otros métodos  
    public double longitud() {  
        return origen.distancia(extremo);  
    }  
}
```

Uso de objetos (duplicación)

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(Punto pto1, Punto pto2) {  
        origen = new Punto(pto1.ordenada(), pto1.abscisa());  
        extremo = new Punto(pto2.ordenada(), pto2.abscisa());  
    }  
    ... // Otros métodos  
    public double longitud() {  
        return origen.distancia(extremo);  
    }  
}
```

- Solo se utiliza en caso de no querer que los objetos componentes sean accesibles desde otros objetos.
- En este caso hay que asegurarse de que ninguno de los métodos de acceso van a proporcionar referencias a estos componentes.

Identidad e igualdad de objetos

- (Identidad). Dos referencias que comparten al **mismo objeto** son idénticas. Se usa `==`

```
String s1 = "hola a todos";  
Punto pto2 = pto1;  
System.out.println(pto1 == pto2) // true
```

- (Igualdad). Dos referencias que señalan a **dos objetos distintos** que contienen los mismos valores **pueden ser** iguales. Se usa `equals`.

`equals` es programable (se verá en el tema 4)

```
String s1 = "hola a todos";  
String s2 = new String("hola a todos");  
System.out.println(s1 == s2) // false  
System.out.println(s1.equals(s2)) // true
```

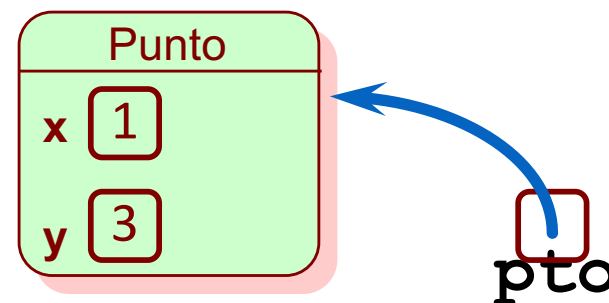

Eliminación de objetos

- La eliminación de objetos
 - Automática cuando el objeto es inalcanzable
 - Porque se pierden todas las referencias a él.
 - Se puede provocar la pérdida de una referencia a un objeto
`pto = null;`
- Se puede solicitar la eliminación automática
 - invocando el método de clase **gc ()** de la clase **System**.
 - Es una sugerencia. Puede completar la acción si lo cree conveniente.

Tipos básicos versus clases

- Variables de tipos básicos
 - Almacenan el **valor**
- Variables de objetos
 - Almacenan la **referencia al objeto**
- Esto tiene consecuencias en la manipulación de referencias y valores.

```
Punto pto = new Punto(1, 3);
```



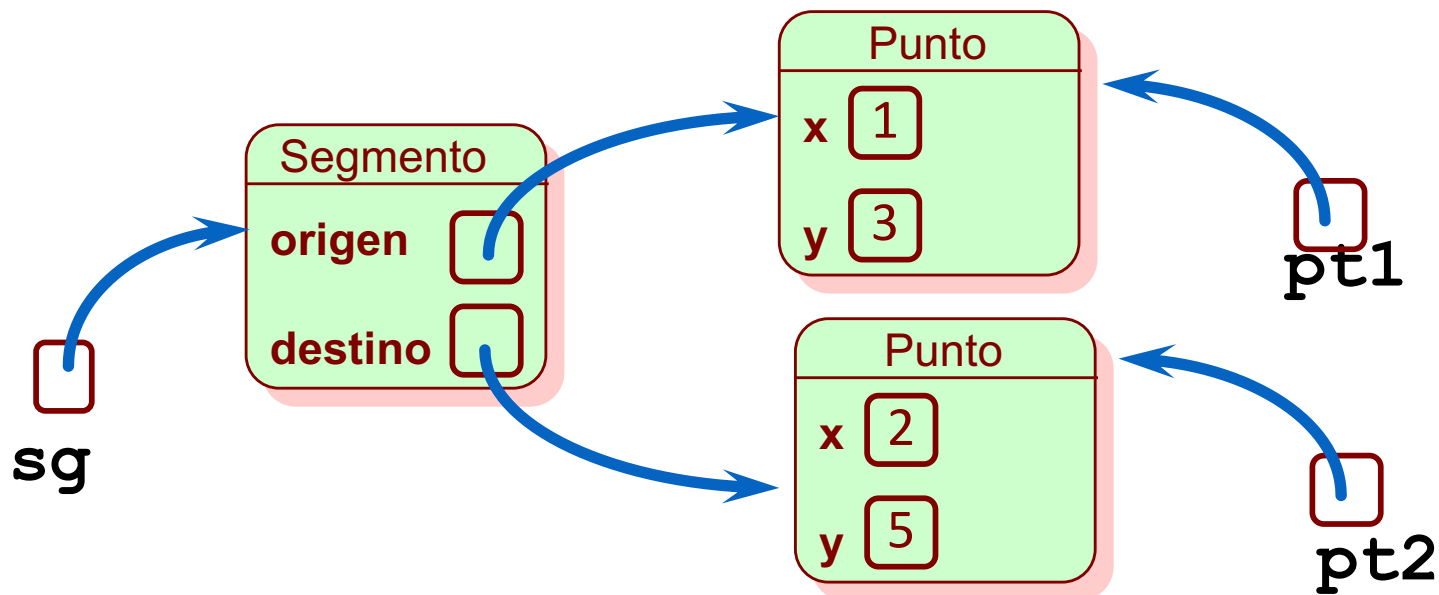
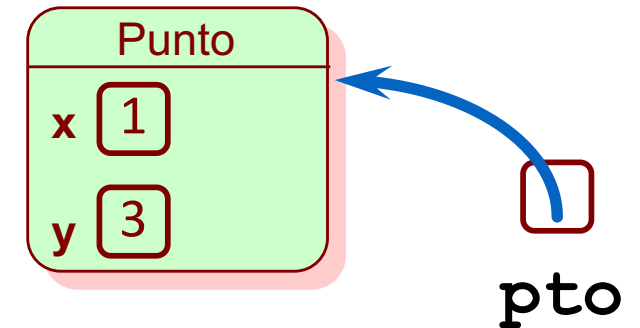
Tipos básicos y clases

```
Punto pto = new Punto(1, 3);
```

```
Punto pt1 = new Punto(1, 3);
```

```
Punto pt2 = new Punto(2, 5);
```

```
Segmento sg = new Segmento(pt1, pt2);
```



Conversiones implícitas: contextos

- La conversión implícita se produce en los siguientes contextos:
 - **Asignaciones** (el tipo de la expresión se promociona al tipo de la variable de destino)
 - **Invocaciones de métodos** (los tipos de los parámetros actuales se promocionan a los tipos de los parámetros formales)
 - **Evaluación de expresiones aritméticas** (los tipos de los operandos se promocionan al del operando con el tipo más general y, como mínimo se promocionan a `int`)
 - **Concatenación de cadenas** (los valores de los argumentos se convierten en cadenas)

Ámbito de una variable

- Un identificador debe ser único dentro de su ámbito.
- El **ámbito** de una variable es la zona de código donde se puede usar su identificador sin cualificar.
- El ámbito determina cuándo se crea y cuándo se destruye espacio de memoria para la variable.
- Las variables, según su ámbito, se clasifican en las siguientes categorías:
 - Variable de clase o de instancia
 - Variable local
 - Parámetro de método
 - Parámetro de gestor de excepciones

Ámbitos

class MiClase { ...

Variables de clase

Parámetros y

~~Variables de instancia~~

Variables locales

Parámetros
del catch

static public void método(...) {

{...
}

catch (...) {

}

}

}

Inicialización de variables

- Las **variables de clase** se inicializan automáticamente al **cargar la clase** en la máquina virtual.
- Las **variables de instancia** se inicializan automáticamente **cada vez que se crea una instancia**.
- Las **variables locales** no se inicializan de forma automática y el compilador produce un error si no se hace manualmente.
- Valores de inicialización automática:
`false '     ' 0 +0.0F +0.0D null`

Control de excepciones (I)

- Mecanismo de ayuda para la comunicación y el manejo de errores
- Cuando se produce un error en un método:
 1. Se genera un objeto de la clase **Exception** con información sobre el error,
 2. Se **interrumpe** el flujo normal de ejecución, y
 3. El entorno de ejecución **trata de encontrar** un tratamiento para dicho objeto excepción.
 1. dentro del propio método o
 2. en uno anterior en la pila de activaciones.

Control de excepciones (II)

Existe una sentencia en tres partes relacionada con el control de excepciones:

- **try**
delimita un bloque de instrucciones donde se puede producir una excepción,
- **catch**
identifica un bloque de código asociado a un bloque **try** donde se trata un tipo particular de excepción,
- **finally**
identifica un bloque de código que se ejecutará después de un bloque **try** con independencia de que se produzcan o no excepciones.

Control de excepciones (III)

El aspecto normal de un segmento de código con control de excepciones sería el siguiente:

```
try {  
    <sentencia/s>  
} catch (<tipoexcepción> <identif>) {  
    <sentencia/s>  
}  
...  
} catch (<tipoexcepción> <identif>) {  
    <sentencia/s>  
} finally {  
    <sentencia/s>  
}
```

Bloque vigilado

Manejador

Manejador

Siempre se ejecuta

Provocar una excepción

- Es posible provocar una excepción por medio de **throw**.
- Por ejemplo:
 - Si al crear una jarra nos proporcionan una capacidad negativa podemos lanzar una excepción:

```
public Jarra(int capacidadInicial) {  
    if (capacidadInicial < 0)  
        throw new IllegalArgumentException("capacidad negativa");  
    capacidad = capacidadInicial;  
    contenido = 0;  
}
```

¿qué ocurre si volcamos una jarra sobre sí misma?

Clases Especiales

- Evitan tener que escribir mucho código en determinadas situaciones que aparecen frecuentemente.
- **Enumeraciones**
 - Permite crear clases con un número **determinado** de instancias.
 - Se declaran con `enum` en lugar de con `class`.
- **Tuplas nombradas (registros)**
 - Permite crear clases **inmutables** para albergar varios datos.
 - Muy útil cuando un método debe devolver más de un valor.
 - Se declaran con `record` en lugar de con `class`.

Java 16

Enumeraciones

- Permite crear clases con un **número determinado de instancias**.
 - No se pueden crear más instancias de la clase
- Se declaran con `enum` en lugar de con `class`
- Ejemplo
 - Queremos crear la clase Moneda que solo tenga dos instancias, una que representa a la cara y otra que representa a la cruz.
 - Veamos cómo hacerlo con `class` y con `enum` para ver la ventaja de utilizar `enum`.

La clase Moneda con `class`

```
public class Moneda {  
    public final static Moneda Cara;  
    public final static Moneda Cruz;  
    static {  
        Cara = new Moneda();  
        Cruz = new Moneda();  
    }  
    private Moneda() {};  
    @Override  
    public String toString() {  
        return (this == Cara) ? "Cara" : "Cruz";  
    }  
}
```

Dos variables de
clase para guardar
las dos instancias
únicas

Bloque static:
Se ejecuta al
cargar la clase

```
public class EjemploSinEnum {  
    public static void main(String [] args) {  
        Moneda c = Moneda.Cara;  
        Moneda r = Moneda.Cruz;  
        System.out.println(c)           // Cara  
    }  
}
```



La clase Moneda con enum

```
public enum Moneda {Cara, Cruz};

public class EjemploConEnum {
    public static void main(String [] args) {
        Moneda c = Moneda.Cara;
        Moneda r = Moneda.Cruz;
    }
}
```



- Al hacerlo así, no solo se crea el método toString sino que se crean algunos métodos de utilidad más como valueOf, values, ordinal, etc.
 - valueOf(String) devuelve un enumerado conocido el String que lo representa.
 - static values() devuelve un array con todos los enumerados.
 - int ordinal() devuelve la posición del enumerado en la descripción.

Otro ejemplo de enumerado

```
enum Semana {Lun, Mar, Mie, Jue, Vie, Sab, Dom};
```

```
public class EjemploEnum {  
    public static void main(String[] args) {  
        Semana s = Semana.Lun;  
        Semana t = Semana.valueOf("Mie");
```

```
        for (Semana se : Semana.values())  
            System.out.print(se + " ");  
        System.out.println();  
        int n = t.ordinal();  
        System.out.print(n);
```

```
                Lun Mar Mie Jue Vie Sab Dom
```

```
                2
```

```
        }
```

```
    }
```

Tuplas nombradas. Record

- Permite crear **clases inmutables** para albergar varios datos.
- Muy útil cuando un método debe devolver más de un valor.
- Se declaran con `record` en lugar de con `class`
- Ejemplo
 - Queremos crear la clase `Par` que mantenga dos datos, un `String` y un `double`.
 - Veamos cómo hacerlo con `class` y con `record` para ver las ventajas de usar `record`.

La clase Par con class

```
public final class Par {  
    final private String primero;  
    final private double segundo;  
    public Par(String p, double d) {  
        primero = p;  
        segundo = d;  
    }  
    public String primero() {  
        return primero;  
    }  
    public double segundo() {  
        return segundo;  
    }  
    @Override  
    public String toString() {  
        return "Par[primero="+primero+", segundo="+segundo+"]";  
    }  
}
```

```
public class EjemploSinRecord {  
    public static void main(String [] args) {  
        Par p = new Par("Hola", 34.2);  
        String s = p.primero();  
        System.out.println(p);  
    }  
}
```



La clase Par con record

```
public record Par(String primero, double segundo){};
```

```
public class EjemploConRecord {  
    public static void main(String [] args) {  
        Par p = new Par("Hola", 34.2);  
        String s = p.primer();  
        System.out.println(p);  
    }  
}
```



- Al hacerlo así, no solo se crean los métodos primero, segundo y toString sino que se crean algunos otros métodos de utilidad.

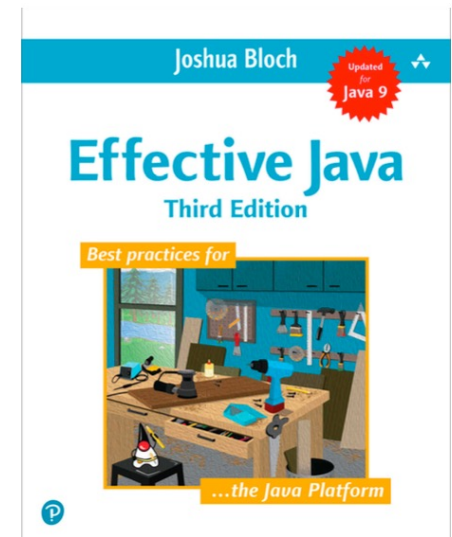
Recomendación 17 de 90 del libro Effective Java

“Classes should be made immutable unless there is a very good reason to make them mutable... If a class cannot be made immutable, you should still limit its mutability as much as possible.”

— *Joshua Bloch*

“Las clases deben ser inmutables a menos que haya una razón muy buena para hacerlos mutable ... Si una clase no puede ser de hecho inmutable, se debería limitar su mutabilidad tanto como sea posible.”

— *Joshua Bloch*



Clases anidadas

- Se definen dentro del cuerpo de otra clase.
- Aunque se pueden distinguir diversos tipos de clases anidadas (internas, locales, anónimas), dependiendo del ámbito en el que se declaren, solo consideraremos las denominadas:
 - **Clases internas estáticas**
 - **Clases anónimas (las veremos más adelante)**
- Una clase interna estática es la que se define como un atributo más de la clase, y en la que se utiliza el calificador **static**.
 - Para acceder a ellas debe calificarse con el nombre de la clase externa (si es visible). Salvo esto, es una clase como cualquier otra.

Clases internas estáticas

```
public class Urna {  
  
    static public enum ColorBola {Blanca, Negra};  
  
    private int nBlancas, nNegras;  
  
    public Urna(int nB, int nN) {  
        nBlancas = nB;  
        nNegras  = nN;  
    }  
  
    public ColorBola sacaBola() {  
        ColorBola bolaSacada = null;  
        if (...) {  
            bolaSacada = ColorBola.Blanca;  
            nBlancas--;  
        } else {  
            bolaSacada = ColorBola.Negra;  
            nNegras--;  
        }  
        return bolaSacada;  
    }  
    ...  
}
```

- Un ejemplo de clase interna estática con datos enumerados.

```
Urna.ColorBola cb = Urna.ColorBola.Negra;
```

```
...
```