

Programación Orientada a Objetos

De C a Java

Objetos como estado y comportamiento

Gerson Garrido

Semana 1

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. El curso: qué vamos a aprender y qué vamos a construir
2. De C a Java: bytecode, JVM, tipos y referencias
3. Las herramientas: `javac`, `java` y `jshell`
4. ¿Por qué objetos? Datos sueltos frente a estado + comportamiento
5. Una primera clase, `new` y la identidad de los objetos

> **Lectura**
>
> Eck, *Introduction to Programming Using Java* (v. 9): secs. 1.3, 1.5, 2.1–2.3, 2.6, 5.1 y 5.2.

SECCIÓN 01

El curso

01

Cuatro partes

Objetos y clases

clases, referencias, encapsulamiento

sem. 1–4

Herencia e interfaces

polimorfismo, dynamic dispatch

sem. 5–8

Programar con tipos

excepciones, genéricos, colecciones, streams

sem. 9–12

Diseño

SOLID, patrones, refactoring

sem. 13–16

Los unit tests con JUnit acompañan todo el curso desde la semana 3.

Cómo funciona el curso

- **Teoría** (4 h): dos sesiones por semana.
- **Laboratorio** (2 h): práctica guiada, con pruebas.
- **Proyecto**: Bodega, un solo sistema que crece todo el semestre.
- **Exámenes**: parcial en la semana 8 y final al cierre del curso.
- **Lectura**: el libro de Eck, gratis en línea, y los apuntes del curso.
- **Lenguaje**: Java 25, la versión LTS actual.

Todo está en el sitio del curso

Calendario, guías semanales, apuntes, diapositivas, código y enunciados.

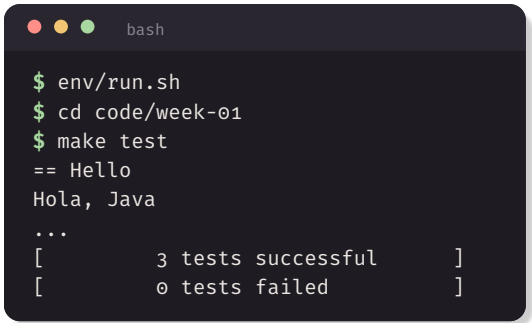
Bodega: lo que van a construir

	Entrega	Lo que agrega
P0	Calentamiento	Programas cortos; la clase `Product` como primera clase
P1	Inventario	Clases con invariantes, `ArrayList`, pruebas JUnit
P2	Catálogo	Jerarquía de productos, polimorfismo, interfaces
P3	Ventas	Excepciones, genéricos, `Map` y `Set`
P4	Reportes	Lambdas y streams; refactoring con SOLID
P5	Aplicación	Patrones, archivos y una interfaz de línea de comandos

Cada entrega debe pasar sus pruebas y las de las anteriores: el código de la semana 2 lo siguen usando en la semana 16.

El entorno: el mismo para todos

- Un contenedor **Docker** con el JDK 25, JUnit y `make`.
- Igual en Linux, Windows (WSL2) y macOS, incluido Apple Silicon.
- Editan con su editor de siempre; compilan y ejecutan en el contenedor.
- Se instala en el laboratorio de esta semana.



SECCIÓN 02

De C a Java

02



Java en una diapositiva

- Creado en Sun Microsystems por el equipo de **James Gosling**; publicado en **1995**.
- La idea: escribir una vez, ejecutar en cualquier parte.
- Hoy: Oracle y la comunidad OpenJDK. Una versión cada 6 meses; una **LTS** cada 2 años.
- **Java 25** (septiembre de 2025) es la LTS actual.



James Gosling, 2008. Peter Campbell, CC BY-SA 4.0, Wikimedia Commons.

Lo familiar y lo nuevo

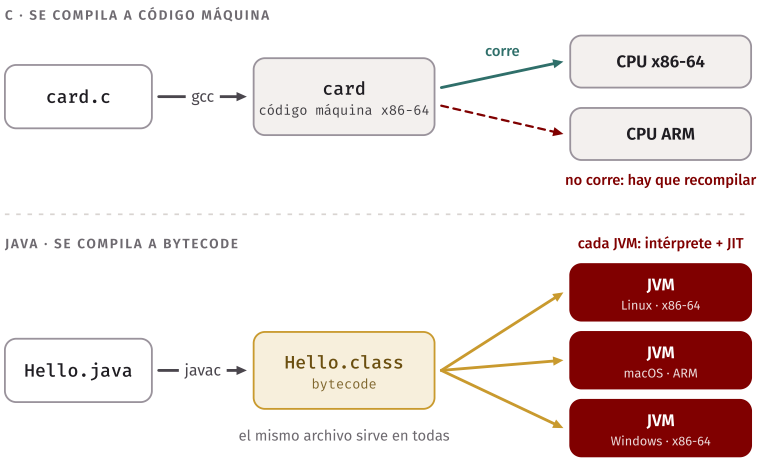
Igual que en C

Llaves y punto y coma. `if`, `while`, `for`, `switch`. Operadores. `int`, `long`, `double`, `char`. Funciones con parámetros y valor de retorno.

Lo que cambia por debajo

- Se compila a bytecode, no a código máquina.
- Tipos de tamaño fijo; `boolean` aparte.
- Referencias en lugar de punteros.
- Sin `free`: garbage collector.
- Todo el código vive en clases.

Compilar: C contra Java



¿Interpretar no es lento? El JIT

bash

```
$ java -version
openjdk version "25.0.5" 2026-10-20 LTS
OpenJDK Runtime Environment Temurin-25.0.5+8 (build 25.0.5+8-LTS)
OpenJDK 64-Bit Server VM Temurin-25.0.5+8 (build 25.0.5+8-LTS, mixed mode,
    sharing)
```

- La JVM (HotSpot) empieza interpretando el bytecode.
- Mide qué métodos se usan mucho y los compila a código máquina mientras el programa corre: compilador **JIT** *(just-in-time)*.
- *Mixed mode* = intérprete + JIT. Un programa que corre varios segundos pasa casi todo su tiempo en código máquina optimizado para *ese* procesador.

Mirar el bytecode: `javap -c`

bash · salida recortada

```
$ javap -c build/Hello.class
Compiled from "Hello.java"
final class Hello {
  void main();
    Code:
       0: ldc           #7     // String Hola, Java
       2: invokestatic  #9     // Method java/lang/IO.println:(Ljava/lang/
    Object;)V
       5: return
}
```

`ldc` carga la cadena; `invokestatic` llama a `IO.println`. Y aunque no escribimos `class`, hay una clase `Hello`: en Java todo vive en una clase.

Tres herramientas del JDK

Herramienta	Qué hace	En C
`javac`	Compila `.java` a `.class` (bytecode)	`gcc -c`
`java`	Arranca la JVM y ejecuta una clase (o un archivo fuente)	`./programa`
`jshell`	Evalúa código Java línea por línea	no hay
`javap`	Desensambla un `.class`	`objdump -d`

Dos maneras de ejecutar

Compilar y ejecutar, como en C:

```bash
$ javac -d build Hello.java
$ ls build
Hello.class
$ java -cp build Hello
Hola, Java
```

A `java` se le da el nombre de la clase, no del archivo. `-cp` *(class path)*: dónde buscar los `.class`.

Directo desde el archivo fuente:

```bash
$ java Hello.java
Hola, Java
```

La JVM compila en memoria y ejecuta; no deja `.class`. Cómodo para programas de un archivo.

Archivos fuente compactos (Java 25)

Hello.java · antes, el programa más corto

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hola, Java");
    }
}
```

Desde Java 25 *(JEP 512)*:

Hello.java

```java
void main() {
    IO.println("Hola, Java");
}
```

- El compilador envuelve todo en una clase implícita con el nombre del archivo.
- `IO.println` imprime; `IO.readln` lee una línea. Las clases comunes no necesitan `import`.
- Los usamos en las semanas 1 y 2. Las clases de verdad van en su propio archivo.

`jshell`: probar una idea sin escribir un programa

jshell

```
jshell> int x = 7 / 2
x ==> 3

jshell> double y = 7 / 2.0
y ==> 3.5

jshell> String s = "Lima"
s ==> "Lima"

jshell> int len = s.length()
len ==> 4
```

- Cada línea se evalúa al momento y muestra el valor que quedó.
- División entera, igual que en C: `7 / 2` es 3.
- `s.length()`: un `String` es un objeto, y se le piden cosas.
- Para salir: `/exit`.

Tipos primitivos: tamaño fijo

Tipo	Bits	Valores
`byte`	8	-128 a 127
`short`	16	$-32\,768$ a $32\,767$
`int`	32	unos $\pm 2{,}1 \times 10^9$
`long`	64	unos $\pm 9{,}2 \times 10^{18}$
`float`	32	IEEE 754 simple
`double`	64	IEEE 754 doble
`char`	16	una unidad UTF-16
`boolean`	–	`true`, `false`

- En C, el tamaño de `int` depende del compilador. En Java, siempre 32 bits.
- No hay `unsigned`.
- `boolean` no es un número: una condición tiene que ser `boolean`.
- `char` no es siempre un carácter: la ñ cabe en uno, muchos emojis necesitan dos.

Error común: una condición al estilo C

Check.java

```java
void main() {
    int n = 5;
    if (n) {
        IO.println("cinco");
    }
}
```

bash

```
$ javac -Xlint:all Check.java
Check.java:3: error: incompatible types:
    int cannot be converted to boolean
    if (n) {
        ^
1 error
```

Hay que decirlo: `if (n != 0)`

Así Java evita errores como `if (x = 0)`, que en C compila y casi nunca es lo que se quería.

Integer overflow

FromC.java

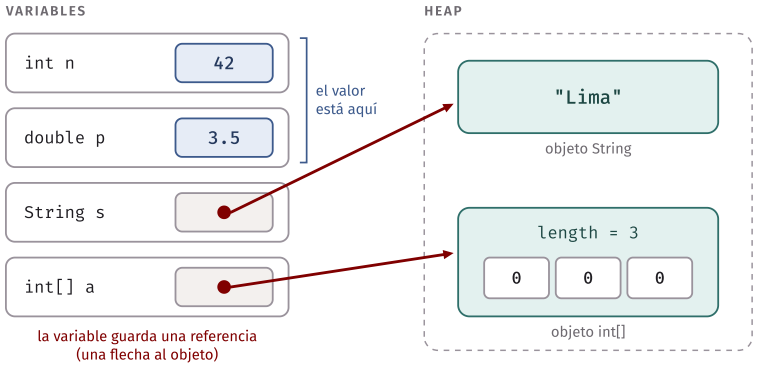
```java
int n = 2_147_483_647;
long big = n + 1L;
IO.println("n + 1 como int:  " + (n + 1));
IO.println("n + 1 como long: " + big);
```

bash

```
$ java -cp build FromC
n + 1 como int:  -2147483648
n + 1 como long: 2147483648
...
```

- El `int` da la vuelta (wraps around) y siempre al mismo valor: el tamaño es fijo. En C, el overflow de un `int` con signo es comportamiento indefinido.
- `1L` es un literal `long`; los _ dentro de un número solo ayudan a leerlo.

Primitivos y referencias



Una referencia se parece a un puntero, pero sin aritmética, sin convertirla en número y siempre a un objeto del tipo correcto. Más en la semana 2.

`String` y arreglos

- `String` es un objeto con métodos, no un `char*` con `'\0'`. No se modifica: `toUpperCase()` devuelve otro.
- Un arreglo es un objeto que conoce su longitud: `a.length`.
- Se crea con `new` y empieza en ceros, nunca con basura.
- Cada índice se verifica. En C, `a[3]` lee lo que haya en esa memoria.

jshell

```
jshell> int[] a = new int[3]
a ==> int[3] { 0, 0, 0 }

jshell> int n = a.length
n ==> 3

jshell> a[3] = 1
|  Exception java.lang.
     ArrayIndexOutOfBoundsException:
     Index 3 out of bounds for length 3
|        at (#7:1)
```

Sin punteros ni `free`

C

```c
int *p = malloc(100 * sizeof(int));
/* ... usar p ... */
free(p);   /* ¿y si se olvida? */
```

Sin `free`: memory leak. Dos `free`, o usar `p` después: cualquier cosa.

Java

```java
int[] p = new int[100];
// ... usar p ...
// no hay free
```

La JVM recupera la memoria de los objetos a los que ya no llega ninguna referencia.

Garbage collector

Liberar lo que nadie alcanza es seguro: sin `free` olvidados ni dobles, sin dangling pointers. Lo alcanzable no se libera (semana 2).

SECCIÓN 03

Objetos como estado + comportamiento

03



Una tarjeta de transporte, al estilo C

card.c

```c
struct card {
    char id[8];
    int balance;       /* saldo en céntimos */
};
void card_recharge(struct card *c, int amount) {
    c->balance += amount;
}
int card_pay(struct card *c, int fare) {
    if (c->balance < fare)
        return 0;      /* no alcanza el saldo */
    c->balance -= fare;
    return 1;
}
```

- Un `struct` con los **datos**.
- Funciones **aparte** que reciben un puntero al `struct`.
- Saldo en céntimos, entero: sin redondeos. S/ 10.00 son 1000.
- Funciona. Así se escriben muchos programas grandes en C.

En C, nada protege el saldo

transit_card.c

```c
int main(void) {
    struct card c = { "A-001", 0 };
    card_recharge(&c, 1000);
    card_pay(&c, 250);
    printf("%s: %d céntimos\n", c.id, c.balance);

    c.balance = -500;   /* nada lo impide */
    printf("%s: %d céntimos\n", c.id, c.balance);
    return 0;
}
```

bash

```
$ ./build/transit_card
A-001: 750 céntimos
A-001: -500 céntimos
```

- Ni `gcc -Wall -Wextra` ni el programa se quejan.
- La regla vive en `card_pay`, pero nada obliga a pasar por ahí.

¿Qué puede salir mal?

StructStyle.java · lo mismo en Java, al estilo C

```java
CardData card = new CardData();
card.id = "A-001";
recharge(card, 1000);  pay(card, 250);
IO.println(card.id + ": " + card.balance + " céntimos");
card.balance = -500;   // nada lo impide
IO.println(card.id + ": " + card.balance + " céntimos");
```

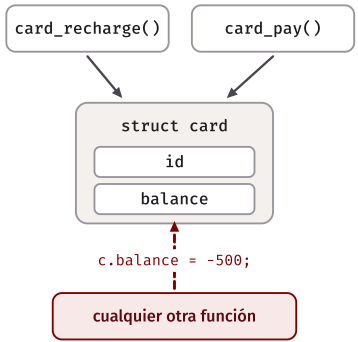
bash

```
$ java StructStyle.java
A-001: 750 céntimos
A-001: -500 céntimos
```

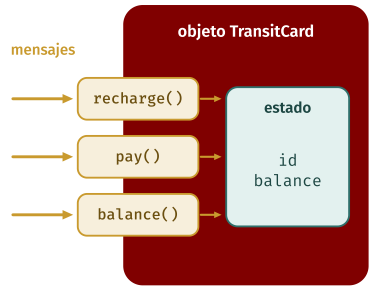
- Las reglas viven en `pay`, pero nada obliga a usarlo.
- ¿Cuál de las funciones que tocan `balance` fue?
- Nada une las funciones con los datos.

Datos sueltos frente a un objeto

AL ESTILO C



COMO OBJETO



los métodos son la única puerta al estado

```
alice.balance = -500;  no compila
```

Objeto = estado + comportamiento

Estado *(state)*

Lo que el objeto recuerda: código y saldo. En Java, campos *(fields)*.

Comportamiento *(behavior)*

Lo que sabe hacer: recargarse, cobrar, decir su saldo. En Java, métodos *(methods)*.

- El resto del programa no toca el estado: le envía un mensaje y el objeto ejecuta el método.
- En C: `card_pay(&c, 250)`. Con objetos, primero *a quién* y después *qué*: `alice.pay(250)`.
- Una **clase** describe un tipo de objeto: sus campos y sus métodos. Cada objeto creado con ella es una **instancia**.

Una primera clase: campos y constructor

TransitCard.java

```java
public class TransitCard {
    private final String id;
    private int balance;      // en cé
     ntimos

    public TransitCard(String id) {
        this.id = id;
        this.balance = 0;
    }

    // ... métodos ...
}
```

- **Campos**: el estado. Cada objeto tiene los suyos.
- `private`: solo el código de esta clase los toca. `final`: `id` no cambia. Más en la semana 3.
- **Constructor**: se llama como la clase y no tiene tipo de retorno. Se ejecuta una vez, al crear el objeto, y deja los campos con valores válidos.
- `this.id` es el campo; `id`, a secas, el parámetro.

Una primera clase: métodos

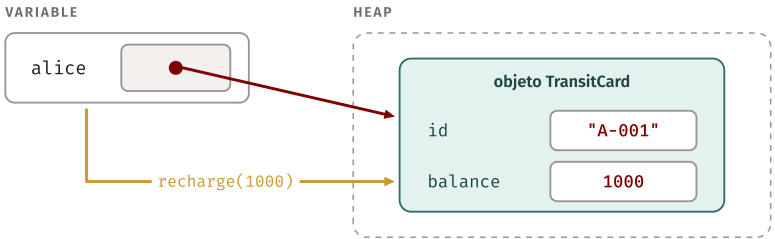
TransitCard.java · sin id()

```java
public void recharge(int amount) {
    balance += amount;
}
public boolean pay(int fare) {
    if (balance < fare) {
        return false;
    }
    balance -= fare;
    return true;
}
public int balance() {
    return balance;
}
```

- **Métodos**: el comportamiento.
- `balance` es el campo del objeto que recibió el mensaje: no hace falta un parámetro `struct card *c`.
- Las reglas de la tarjeta están en un solo lugar: la clase.

new, paso a paso

```
TransitCard alice = new TransitCard("A-001");
alice.recharge(1000);
```



4. un mensaje al objeto: el método cambia su estado

Mensajes y estado

CardDemo.java

```java
TransitCard alice = new TransitCard("A-001");
TransitCard bob = new TransitCard("A-002");

alice.recharge(1000);
boolean paid = alice.pay(250);
IO.println(alice.id() + ": pagó " + paid
        + ", saldo " + alice.balance());

paid = bob.pay(250);
IO.println(bob.id() + ": pagó " + paid
        + ", saldo " + bob.balance());
```

bash

```
$ java -cp build CardDemo
A-001: pagó true, saldo 750
A-002: pagó false, saldo 0
...
```

- El mismo mensaje, `pay(250)`, da resultados distintos: depende del estado de quien lo recibe.
- El método se escribe una vez; cada objeto tiene su propio estado, y lo recuerda.

Identidad

CardDemo.java

```java
TransitCard same = alice;
IO.println("alice == bob: " + (alice == bob));
IO.println("alice == same: " + (alice == same));
```

bash

```
$ java -cp build CardDemo
...
alice == bob: false
alice == same: true
```

- Cada `new` crea un objeto distinto, aunque sus campos valgan lo mismo.
- Con referencias, `==` pregunta: ¿las dos flechas llegan al mismo objeto?
- `same = alice` copia la flecha, no el objeto.
- La próxima semana: identidad frente a igualdad, y `null`.

La tarjeta en `jshell`

jshell · code/week-01

```
jshell> /open TransitCard.java
jshell> TransitCard c = new TransitCard("A
    -001")
c ==> TransitCard@31221be2
jshell> c.recharge(500)
jshell> int b = c.balance()
b ==> 500
jshell> c.balance = 99
|  Error:
|  balance has private access in TransitCard
...
```

- `@31221be2`: un número que cambia en cada ejecución.
- `c.balance()`, con paréntesis: el **método**.
- `c.balance`, sin paréntesis: el **campo**, `private`: no compila.
- La línea que rompía la tarjeta al estilo C ahora falla *antes* de ejecutar.

SECCIÓN 04

Laboratorio

04

`make` y `make test`

- `make`: compila a `build/` con `-Xlint:all -Werror`. Una advertencia detiene la compilación.
- `make test`: corre las demostraciones y las pruebas JUnit.
- `make clean`: borra `build/`.
- Escribir pruebas: semana 3. Hoy basta con ver que corren.

```bash
$ make test
mkdir -p build
javac -Xlint:all -Werror ...
== Hello
Hola, Java
...
[          3 tests found           ]
...
[          3 tests successful      ]
[          0 tests failed          ]
```

Salida recortada.

Resumen

- Java compila a **bytecode**; la **JVM** de cada plataforma lo ejecuta con intérprete y **JIT**.
- `javac` compila, `java` ejecuta, `jshell` evalúa línea por línea. Los archivos compactos de Java 25 empiezan en `void main()`.
- Primitivos de **tamaño fijo**; todo lo demás son **objetos** a los que se llega por **referencias**. Arreglos con `length` e índices verificados. Sin `free`.
- Datos sueltos + funciones aparte: nadie protege las reglas.
- Un **objeto** junta **estado** (campos) y **comportamiento** (métodos); se le envían **mensajes**. `new` reserva, construye y devuelve una referencia.
- Cada objeto tiene su propio estado y su propia **identidad**.

Para la próxima clase

- Lecturas de la semana 2: Eck, secs. 5.1 y 5.2 (relectura), 5.6.1 (`this`) y 4.3.3 (overloading).
- Laboratorio: instalar Docker y levantar el contenedor del curso (guía en el sitio).
- Correr `make test` en `code/week-01` y probar `jshell`.
- Leer el enunciado de **P0**: calentamiento y la clase `Product`.

Pregunta para pensar

Si dos variables apuntan a la misma tarjeta y una de ellas le envía `recharge(500)`, ¿qué saldo ve la otra? ¿Y si cada una hizo su propio `new`?