

Programación Orientada a Objetos

Clases, objetos y referencias

Constructores, heap, alias, `null` e identidad

Gerson Garrido

Semana 2

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Una clase completa: campos, métodos, `this` y constructores
2. Overloading y `this(...)`
3. Stack y heap: qué guarda una variable de tipo clase
4. Alias y paso de parámetros
5. `null`, el `NullPointerException` y el garbage collector
6. Identidad (`==`) frente a igualdad (`equals`)
7. Laboratorio: depurar con `jdb`

> **Lectura**
>
> Eck, *Introduction to Programming Using Java* (v. 9): secs. 5.1 y 5.2, 5.6.1 (`this`), 4.3.3 (overloading), 2.3.3 (`equals` en `String`).

Lo que quedó pendiente la semana pasada

- `TransitCard alice;` no crea ningún objeto. ¿Qué pasa si se usa?
- `alice == same` era `true` porque `same = alice` copió la flecha. ¿Qué hay exactamente en esa variable?
- Dos tarjetas con el mismo código y saldo: ¿son la misma tarjeta? ¿Son iguales?

Esta semana

Primero, una clase con todas sus piezas. Después, la memoria: dónde viven las variables y los objetos.

SECCIÓN 01

Una clase completa

01

El ejemplo: un termómetro de máximas y mínimas

Thermometer.java

```java
public class Thermometer {
    private String place;
    private double current;   // grados C
    private double min;
    private double max;
    private int readings;

    // constructores ...
    // record(double celsius) ...
    // place(), current(), min(), max(),
    // readings(), range() ...
}
```

- Recuerda la temperatura actual, la mínima y la máxima.
- La **clase** es el molde; cada `new` crea una **instancia** *(instance)* con su propio estado.
- El código de los métodos se escribe una vez; los campos, cada objeto tiene los suyos.

Campos: valores por defecto

Tipo del campo	Empieza en
`int`, `long`, ...	`0`
`double`, `float`	`0.0`
`boolean`	`false`
`char`	el carácter 0
referencias	`null`

- `new` pone todos los campos en su valor por defecto antes del constructor.
- Un campo nunca tiene basura, como sí puede tenerla una variable en C.

Variables locales: sin valor por defecto

Uninit.java

```java
void main() {
    int count;
    IO.println(count + 1);
}
```

- Sin valor hasta que se le asigna uno; el compilador lo verifica.
- En C tendría basura. En Java, ni compila.

bash

```
$ javac -Xlint:all Uninit.java
Uninit.java:3: error: variable count might not have been initialized
    IO.println(count + 1);
               ^
1 error
```

Métodos de instancia: el receptor y `this`

Thermometer.java

```java
public void record(double celsius) {
    current = celsius;
    if (celsius < min) {
        min = celsius;
    }
    if (celsius > max) {
        max = celsius;
    }
    readings++;
}
```

- En `lima.record(22.0)`, el objeto de `lima` es el **receptor** *(receiver)*.
- Java le pasa al método una referencia escondida al receptor: `this`.
- `current` significa `this.current`. En C: `c->current`, con el `struct card *c` que escribíamos a mano.

Consultas y modificadores

Consulta *(query)*

Devuelve algo sobre el estado y no lo cambia.

`current()`, `max()`, `range()`

Modificador *(command)*

Cambia el estado; casi siempre es `void`.

`record(double celsius)`

Thermometer.java

```java
public double range() {
    return max - min;       // no hay campo range: se calcula
}
```

Nadie espera que `max()` cambie el termómetro: separarlos ayuda a razonar. Y quien llama a `range()` no sabe si el valor estaba guardado o se calculó.

Constructores

Thermometer.java

```java
public Thermometer(String place, double initial) {
    this.place = place;
    this.current = initial;
    this.min = initial;
    this.max = initial;
    this.readings = 1;
}
```

- Mismo nombre que la clase, sin tipo de retorno.
- Corre una vez por objeto, justo después de que `new` reserva la memoria.
- Su trabajo: dejar los campos en un estado con sentido. La primera lectura es la actual, la mínima y la máxima.

El constructor por defecto desaparece

- Sin ningún constructor, el compilador agrega uno sin parámetros: el **constructor por defecto**.
- Apenas escribimos uno propio, ya no está.
- Si la clase solo tuviera `Thermometer(double initial)`:

bash · líneas largas partidas

```
$ javac -Xlint:all -d out Thermometer.java Main.java
Main.java:2: error: constructor Thermometer in class Thermometer
    cannot be applied to given types;
    Thermometer t = new Thermometer();
                    ^
  required: double
  found:    no arguments
  ...
```

Overloading y `this(...)`

Thermometer.java

```java
public Thermometer(String place, double initial) {
    ... // toda la inicialización
}
public Thermometer(double initial) {
    this("sin lugar", initial);
}
public Thermometer() {
    this(0.0);
}
```

- **Overloading**: mismo nombre, parámetros distintos.
- El compilador elige por los argumentos; el tipo de retorno no cuenta.
- `this(...)` llama a otro constructor: primera instrucción.
- La inicialización vive en un solo lugar.

Usar la clase

ThermometerDemo.java · recortado

```java
Thermometer lima = new Thermometer("Lima", 18.5);
lima.record(22.0);  lima.record(16.0);
Thermometer lab = new Thermometer(21.0);    // overloading: otro constructor
Thermometer fresh = new Thermometer();      // y otro más
lab.record(35.0);
```

bash

```
$ java -cp build ThermometerDemo
Lima: actual 16.0, mín 16.0, máx 22.0, 3 lecturas
sin lugar: 21.0
sin lugar: 0.0
rango de lab: 14.0, rango de lima: 6.0
```

Error común: un constructor con `void`

Thermometer.java

```java
public class Thermometer {
    public void Thermometer(double t) {
        current = t;
    }
}
```

- Con `void` es un método, no un constructor.
- Queda el constructor por defecto. *¿no arguments?* Busquen un `void`.

bash · líneas largas partidas

```
Main.java:2: error: constructor Thermometer in class Thermometer
    cannot be applied to given types;
  required: no arguments
  found:    double
```

Error común: shadowing, el parámetro tapa al campo

ShadowBug.java

```java
class BuggyThermometer {
    private double current;

    BuggyThermometer(double current) {
        current = current;
    }
    ...
}
```

bash

```
$ java -cp build ShadowBug
esperaba 25.0 y obtuve 0.0
```

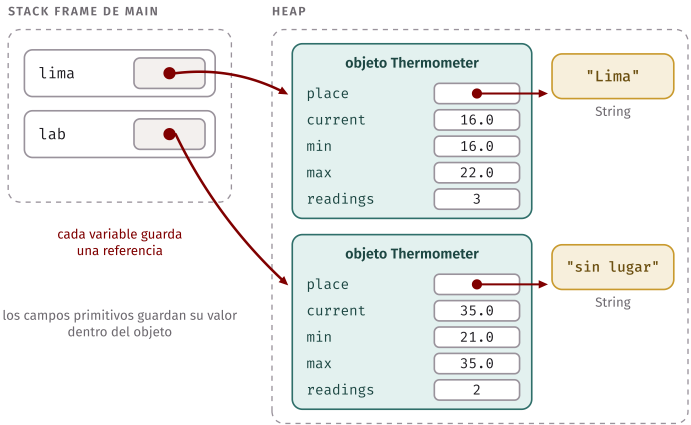
- Dentro del constructor, `current` es el parámetro: se asigna a sí mismo.
- El campo se queda en su valor por defecto, `0.0`. Compila sin advertencias, incluso con `-Xlint:all`.
- La corrección: `this.current = current;`

SECCIÓN 02

Referencias, heap y null

02

Stack y heap


STACK FRAME DE MAIN
HEAP
lima
lab
objeto Thermometer
place
current
16.0
min
16.0
max
22.0
readings
3
"Lima"
String
objeto Thermometer
place
current
35.0
min
21.0
max
35.0
readings
2
"sin lugar"
String
cada variable guarda una referencia
los campos primitivos guardan su valor dentro del objeto

Lo que dice la figura

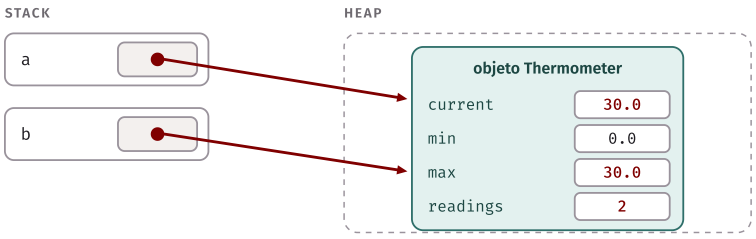
- `lima` y `lab` guardan solo una flecha, sin importar cuántos campos tenga el objeto.
- Los campos primitivos guardan su valor dentro del objeto.
- `place` es un `String`: también es una referencia, a otro objeto.
- El stack frame desaparece cuando el método termina. El objeto dura mientras alguien pueda llegar a él.

Como un puntero a un `struct` reservado con `malloc`...

...pero sin aritmética, sin convertirlo en número, siempre al tipo correcto y sin `free`. Copiar una referencia es barato y nunca copia el objeto.

Alias

```
Thermometer a = new Thermometer();
Thermometer b = a;
b.record(30);
```



el mensaje enviado por b cambia el objeto; a.max() también da 30.0

Alias en código

Aliasing.java

```java
Thermometer a = new Thermometer("Lima", 18.0);
Thermometer b = a;              // alias: dos variables, un solo objeto
b.record(30.0);
IO.println("a.max() = " + a.max() + "   (a == b: " + (a == b) + ")");
```

bash

```
a.max() = 30.0   (a == b: true)
```

- Nadie le envió un mensaje a `a`, y su máxima cambió.
- En un programa grande, con referencias guardadas en campos y listas, un alias olvidado hace que un objeto “cambie solo”.

Antes de modificar un objeto, pregúntense quién más tiene una referencia a él.

Paso de parámetros: siempre por valor

Aliasing.java

```java
void heatUp(Thermometer t) {
    t.record(t.current() + 5);
}
void replace(Thermometer t) {
    t = new Thermometer("Cusco", 4.0);
    t.record(-2.0);
}
```

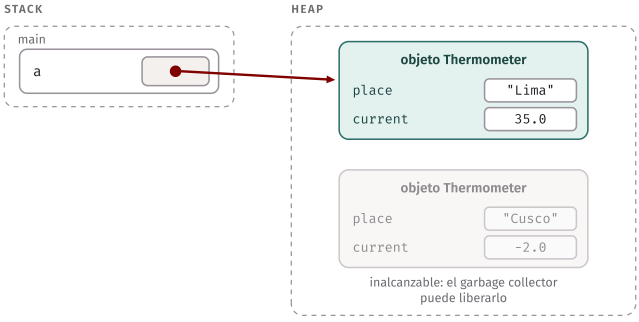
- El parámetro recibe una copia: de la referencia.
- `heatUp` modifica el mismo objeto: se ve afuera.
- `replace` cambia su copia: `a` sigue en Lima.

bash · después de heatUp(a) y de replace(a)

```
después de heatUp:  Lima, actual 35.0
después de replace: Lima, actual 35.0
```

Paso por valor, paso a paso

```
heatUp(a);    // dentro: t.record(t.current() + 5);
replace(a);   // dentro: t = new Thermometer("Cusco", 4.0); t.record(-2.0);
```



al volver, el stack frame desaparece: a sigue en Lima, con 35.0

“Java pasa los objetos por referencia”

No

Lo que se pasa, por valor, es la referencia.

- Si el objeto se pasara por referencia, después de `replace(a)` la variable `a` apuntaría al termómetro de Cusco.
- Java no tiene el `&` de C++: ningún método puede cambiar a qué objeto apunta una variable de quien lo llamó.
- Sí puede cambiar el estado del objeto, si la clase tiene modificadores.

`null`: una referencia que no apunta a nada

NullDemo.java

```java
Thermometer ok = new Thermometer("Lima", 18.0);
IO.println(ok.place() + " tiene " + ok.place().length() + " letras");

Thermometer broken = new Thermometer(null, 18.0);    // nada lo impide (todavía)
IO.println("lugar: " + broken.place());
IO.println(broken.place() + " tiene " + broken.place().length() + " letras");
```

- `null` es el valor por defecto de los campos de tipo referencia.
- Se puede comparar (`t == null`), asignar, pasar e imprimir.
- Lo que no se puede: enviarle un mensaje. No hay objeto.

Leer un `NullPointerException`

bash · líneas largas partidas

```
$ java -cp build NullDemo
Lima tiene 4 letras
lugar: null
Exception in thread "main" java.lang.NullPointerException:
  Cannot invoke "String.length()" because the return value
  of "Thermometer.place()" is null
        at NullDemo.main(NullDemo.java:8)
```

- `at NullDemo.main(NullDemo.java:8)`: método y línea.
- *Cannot invoke* `"String.length()"`: el mensaje que se intentó enviar.
- *because the return value of* `"Thermometer.place()"` *is null*: qué expresión era `null` *(JEP 358, desde Java 15)*. Con `javac -g`, nombra también las variables locales.
- Imprimir `null` con `+` no falla: escribe `null`.

El error real está antes

- El programa falla en la línea 8. El dato malo entró en la línea 6: `new Thermometer(null, 18.0)`.
- Un termómetro sin lugar no tiene sentido, y la clase lo aceptó en silencio.
- Cuanto más tarde salta el error, más lejos está de su causa.

La semana 3

El constructor valida sus argumentos y rechaza `null` en el momento de crear el objeto: un objeto inválido nunca llega a existir.

El garbage collector

- Un objeto al que no llega ninguna referencia es **inalcanzable**: el termómetro de Cusco después de `replace`.
- El garbage collector recupera su memoria cuando le conviene. No hay `free` ni destructores.
- Se vuelve inalcanzable al perder la última referencia: otra asignación, `null`, o el fin del método que tenía la variable.

Lo que no libera

Un objeto todavía alcanzable, aunque ya no sirva: una lista que guarda referencias viejas. Así son los memory leaks en Java.

Identidad frente a igualdad

Identidad *(identity)*

¿Las dos flechas llegan al mismo objeto?

`a == b`

Igualdad *(equality)*

¿Los dos objetos representan lo mismo?

`a.equals(b)`, si la clase lo define

- `String` define `equals`: compara los caracteres.
- Si la clase no lo define, hereda uno que hace lo mismo que `==`.

== y `equals` con textos

Equality.java

```java
String s1 = "Lima";
String s2 = "Lima";
String part = "Li";
String s3 = part + "ma";

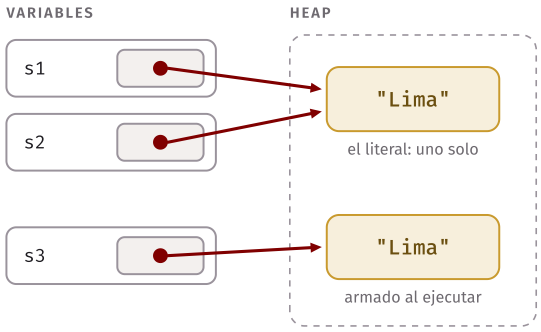
IO.println("s1 == s2:       " + (s1 == s2));
IO.println("s1 == s3:       " + (s1 == s3));
IO.println("s1.equals(s3): " + s1.equals(s3));
```

bash

```
s1 == s2:       true
s1 == s3:       false
s1.equals(s3): true
```

- Java guarda una sola copia de cada literal: `s1` y `s2` son el mismo objeto.
- `s3` se arma al ejecutar: otro objeto con el mismo texto.

Dos flechas, un objeto; dos objetos, el mismo texto



RESULTADO

`s1 == s2` → `true`
misma flecha

`s1 == s3` → `false`
flechas distintas

`s1.equals(s3)` → `true`
mismo texto

Error común: comparar textos con ==

ReadCompare.java

```java
String city = IO.readln("Ciudad: ");
IO.println("city == \"Lima\":      " + (city == "Lima"));
IO.println("city.equals(\"Lima\"): " + city.equals("Lima"));
```

bash

```
$ echo Lima | java -cp build ReadCompare
Ciudad: city == "Lima":      false
city.equals("Lima"): true
```

Con literales a veces “funciona”; con un texto leído, no. Para comparar textos, siempre `equals`.

Y con nuestras clases

Equality.java

```java
Thermometer t1 = new Thermometer("Lima", 18.0);
Thermometer t2 = new Thermometer("Lima", 18.0);
IO.println("t1 == t2:       " + (t1 == t2));
IO.println("t1.equals(t2): " + t1.equals(t2));
```

bash

```
t1 == t2:       false
t1.equals(t2): false
```

- `Thermometer` no define `equals`: el heredado compara identidad.
- ¿Cuándo son “iguales” dos termómetros? ¿Mismo lugar? ¿Mismas lecturas? Es una decisión de diseño.
- Escribir un `equals` correcto: semana 5.

Alias en `jshell`

jshell · code/week-02

```
jshell> /open Thermometer.java
jshell> Thermometer a = new Thermometer()
a ==> Thermometer@31221be2
jshell> Thermometer b = a
b ==> Thermometer@31221be2
jshell> b.record(30)
jshell> double m = a.max()
m ==> 30.0
```

- `@31221be2` no es una dirección, pero dice si es el mismo objeto.
- `a` y `b`: el mismo número, un solo objeto.
- `record` es `void`: `jshell` no muestra nada. Pero `a.max()` ya da 30.0.

SECCIÓN 03

Laboratorio

03

Un debugger

- Detiene el programa en un **breakpoint**, muestra variables y campos, avanza instrucción por instrucción.
- El JDK trae `jdb`, de línea de comandos: funciona dentro del contenedor.
- Los editores tienen uno gráfico. Las ideas son las mismas.
- Hay que compilar con `javac -g`: el `Makefile` de la semana ya lo hace.

`stop at C:n`	breakpoint en la línea `n`	`next`	ejecuta la línea
`run`	arranca	`step`	entra al método
`where`	call stack	`cont`	sigue
`print x`	valor de `x`	`dump x`	campos del objeto

`jdb`: detenerse en `record`

bash · salida recortada

```
$ jdb -classpath build ThermometerDemo
> stop at Thermometer:33
> run
Breakpoint hit: "thread=main",
  Thermometer.record(), line=33 bci=0
main[1] where
  [1] Thermometer.record (Thermometer.java:33)
  [2] ThermometerDemo.main (ThermometerDemo.java:4)
main[1] print celsius
 celsius = 22.0
```

jdb

```
main[1] dump this
 this = {
    place: "Lima"
    current: 18.5
    min: 18.5
    max: 18.5
    readings: 1
}
```

`where`: llamado desde la línea 4. `print`: el parámetro. `dump this`: el receptor.

`jdb`: avanzar y volver a mirar

jdb

```
main[1] next
Step completed: ... line=34 bci=5
main[1] next
Step completed: ... line=37 bci=19
main[1] next
Step completed: ... line=38 bci=28
```

jdb

```
main[1] dump this
 this = {
    place: "Lima"
    current: 22.0
    min: 18.5
    max: 18.5
    readings: 1
}
```

- De la 34 a la 37: 22.0 no es menor que 18.5, el primer `if` no entra.
- En la 38, `current` ya es 22.0 y `max` todavía 18.5.
- `clear Thermometer:33` quita el breakpoint; `cont` deja terminar el programa.

Laboratorio: primeras clases y depuración

1. Una clase pequeña con overloading de constructores (uno delega con `this(...)`), consultas y modificadores. Probarla desde un archivo compacto.
2. Dibujar a mano stack y heap después de cada línea de un programa con alias. Comprobarlo en `jshell`.
3. Provocar un `NullPointerException` y un error de alias. Leer el stack trace y encontrar la línea responsable.
4. Depurar con `jdb` (o el editor): un breakpoint dentro de un método, `next`, `dump this`.

`make test` en `code/week-02`: demostraciones y pruebas. `NullDemo` termina con excepción a propósito.

Resumen

- Campos con **valor por defecto**; variables locales sin él, y el compilador lo verifica.
- Métodos sobre un **receptor** (`this`). **Consultas** y **modificadores**.
- **Constructores**: sin tipo de retorno; el por defecto desaparece; **overloading** y `this(...)`.
- Variables en el **stack**, objetos en el **heap**; una variable de tipo clase guarda una **referencia**.
- `=` copia la referencia: **alias**. Los argumentos se pasan **por valor**.
- `null` no apunta a nada; el `NullPointerException` dice qué expresión y qué línea.
- El **garbage collector** libera lo inalcanzable.
- `==`: **identidad**. `equals`: **igualdad**, si la clase lo define. Textos con `equals`.

Para la próxima clase

- Lecturas de la semana 3: Eck, 5.1.3 (getters y setters), 4.7.1 y 8.2.2–8.2.3 (precondiciones e invariantes), 4.3.7 (`throw`).
- **P0** se entrega la semana 3: la clase `Product` usa justo lo de hoy (constructores, `this`, consultas y modificadores).
- Correr `make test` en `code/week-02` y probar `jdb` con otra línea.

Pregunta para pensar

Con los métodos públicos de `Thermometer`, ¿se puede crear un termómetro sin lugar? ¿Y uno cuya mínima sea mayor que su máxima? ¿Quién debería impedirlo?