


UNIVERSIDAD NACIONAL DE INGENIERÍA

Programación Orientada a Objetos

# Herencia y la clase `Object`

## `extends`, `super`, `override` · `toString`, `equals`, `hashCode`

**Gerson Garrido**

Semana 5

# Hoy

1. Herencia: una clase que es un caso particular de otra
2. Constructores y `super(...)`: en qué orden se construye un objeto
3. Override frente a overloading; `protected` y `final`
4. `Object`: `toString`, `equals` y su contrato
5. `hashCode`: por qué va siempre con `equals`

> **Lectura**
>
> Eck (v. 9): 5.5.1–5.5.3, 5.5.6, 5.6.2, 5.6.3, 5.3.2; de 10.1.6 la parte de `equals` y de 10.3.3 la de `hashCode`.

SECCIÓN 01

# Herencia

01

# Una cuenta de ahorros

BankAccount.java · resumen

```java
public class BankAccount {
    private final String id;
    private long balance; // céntimos

    BankAccount(String id)
    void deposit(long cents)
    boolean withdraw(long cents)
    String id()
    long balance()
}
```

- Como en la semana 3: el saldo nunca es negativo.
- El banco quiere **cuentas de ahorros**: todo lo anterior, más un interés mensual y a lo más 3 retiros por mes.
- ¿Copiar y pegar `BankAccount.java`? Dos copias que se separan con el tiempo, y un método que recibe una `BankAccount` no acepta la otra.

# `extends`: escribir solo la diferencia

SavingsAccount.java

```java
public class SavingsAccount
        extends BankAccount {
    private final int ratePercent;
    private int withdrawals;

    public SavingsAccount(String id,
                          int ratePercent) {
        super(id);
        // ... validar ratePercent ...
        this.ratePercent = ratePercent;
    }
```

- `BankAccount`: **superclase** *(superclass)*.
- `SavingsAccount`: **subclase** *(subclass)*.
- La subclase hereda campos y métodos: no se repiten.
- `super(id)`: construir primero la parte `BankAccount`. Ya volvemos a esto.

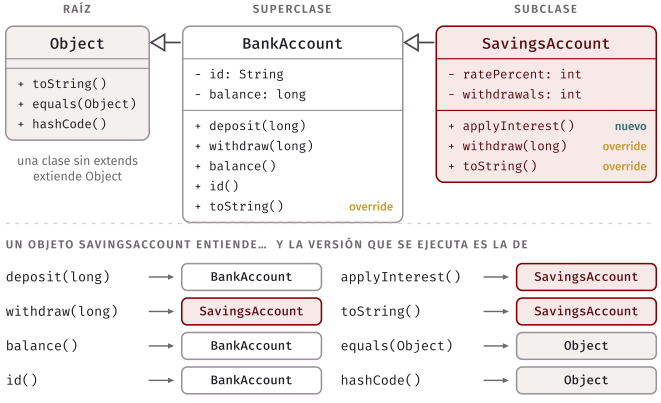
# Usar, agregar, hacer override

SavingsAccount.java

```java
@Override
public boolean withdraw(long cents) {
    if (withdrawals == MAX_WITHDRAWALS) {
        return false;
    }
    boolean ok = super.withdraw(cents);
    if (ok) {
        withdrawals++;
    }
    return ok;
}
```

- **Usar** lo heredado: `applyInterest()` llama a `balance()` y a `deposit(...)`, que también son suyos.
- **Agregar**: `applyInterest()`, `ratePercent`, `withdrawals`.
- **Hacer override**: otra versión de `withdraw`.
- `super.withdraw(cents)` reutiliza la de la superclase, que valida monto y saldo.

# La jerarquía en UML




Los campos id y balance también están dentro del objeto, pero son private: el código de SavingsAccount no los nombra; los usa a través de deposit, withdraw y balance().

# En acción

AccountDemo.java

```java
SavingsAccount savings =
        new SavingsAccount("A-001", 2);
savings.deposit(100_000); // heredado
savings.applyInterest();  // propio
IO.println(savings);      // override
for (int i = 1; i <= 4; i++) {
    IO.println("retiro " + i + ": "
        + savings.withdraw(1_000));
}
```

bash · líneas partidas

```
$ java -cp build AccountDemo
...
BankAccount[A-001, saldo 102000]
  (ahorros, 2 % mensual)
retiro 1: true
retiro 2: true
retiro 3: true
retiro 4: false
...
```

# Herencia = relación *es un*

**Suena bien: herencia**

Una cuenta de ahorros *es una* cuenta bancaria.
Un empleado por horas *es un* empleado.

**Suena mal: no es herencia**

Una playlist *es un* `ArrayList`.
Un auto *es un* motor.

- Si la frase suena falsa, la relación es *tiene un*: un campo.
- En el diagrama de clases: flecha con triángulo hueco hacia la superclase.
- Una clase sin `extends` extiende `Object`: toda clase desciende de `Object`.

# Los campos `private` se heredan… pero no se tocan

SavingsAccount.java · intento

```
1  balance += balance * 2 / 100;   // dentro de applyInterest()
```

bash · recortada

```
SavingsAccount.java:7: error: balance has private access in BankAccount
```

- El objeto tiene los cuatro campos, pero los de `BankAccount` siguen siendo suyos.
- Es el encapsulamiento funcionando: el invariante del saldo sigue a cargo de `BankAccount`. La subclase usa `deposit` y `withdraw`, como cualquier cliente.
- Los **constructores no se heredan**: cada clase declara los suyos.

SECCIÓN 02

# Constructores y `super`

02



# Primero, `super(...)`

SavingsAccount.java · sin super(id)

```java
public SavingsAccount(String id, int ratePercent) {
    this.ratePercent = ratePercent;
}
```

bash · recortada, líneas partidas

```
error: constructor BankAccount in class BankAccount
  cannot be applied to given types;
```

- Lo primero en un constructor es `super(...)`. Si falta, el compilador pone `super();`: aquí no compila.
- No agreguen un constructor vacío a `BankAccount` (cuentas sin código): escriban `super(id)`.

# Se llaman hacia arriba, terminan hacia abajo

ConstructorOrder.java

```java
class Vehicle {
    Vehicle() { IO.println(...); }
}
class Car extends Vehicle {
    Car() {   // super(); implícito
        IO.println(...);
    }
}
class SportsCar extends Car {
    SportsCar() {
        super();
        IO.println(...);
    }
}
```

`main` hace `new SportsCar()`:

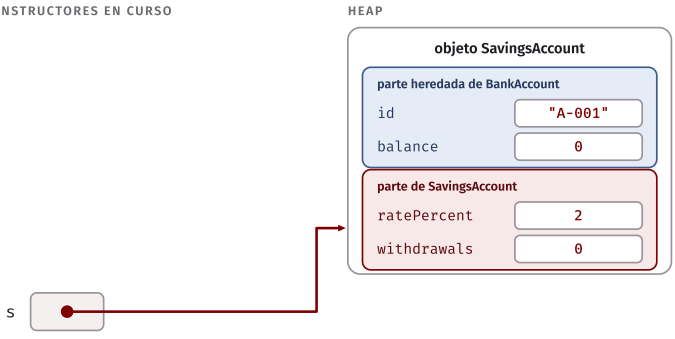
bash · recortada

```
Vehicle(): termina ...
Car(): termina ...
SportsCar(): termina ...
```

- Lo primero que se imprime es lo de `Vehicle`: cada parte se construye sobre una superclase ya lista.

# `new SavingsAccount("A-001", 2)`, paso a paso

```
SavingsAccount s = new SavingsAccount("A-001", 2);
```




5. new devuelve la referencia, que se guarda en s

# En el bytecode

bash · salida recortada

```
$ javap -c -cp build SavingsAccount
  public SavingsAccount(java.lang.String, int);
       0: aload_0
       1: aload_1
       2: invokespecial #1  // BankAccount."<init>":(Ljava/lang/String;)V
       5: iload_2
       6: ifge          23
```

- `super(id)` es `invokespecial` al constructor de `BankAccount` (`<init>`): la primera instrucción. Recién después se valida la tasa (`ifge`).
- En la semana 1, `Hello` llamaba a `Object."<init>"`: hasta la clase más simple llama al constructor de su superclase.

SECCIÓN 03

# Override

03



# `@Override` y `super.metodo()`

SavingsAccount.java

```java
@Override
public String toString() {
    return super.toString()
            + " (ahorros, " + ratePercent
            + " % mensual)";
}
```

- Override: la misma firma que el método heredado.
- `super.toString()`: la versión de `BankAccount`.
- `@Override`: el compilador verifica que de verdad hay un override.

**Sin `@Override`, un error de tipeo es un método nuevo**

`withdraw(int)` en lugar de `withdraw(long)`, o `tostring()`: compila, y se sigue ejecutando la versión heredada. Con `@Override`, no compila. Pónganla siempre.

# Override no es overloading

| | **Override** | **Overloading** |
|---|---|---|
| Firma | la misma que la heredada | mismo nombre, otros parámetros |
| Dónde | en una subclase | en la misma clase (o una subclase) |
| Efecto | reemplaza la versión heredada | agrega otro método; conviven |
| Quién elige | el objeto, al ejecutar | el compilador, por los tipos |

La última fila es el tema de la semana 6: dynamic dispatch.

# Un override no puede ser más cerrado

Shout.java

```java
public class Shout {
    @Override
    String toString() {
        return "HOLA";
    }
}
```

bash · líneas partidas

```
Shout.java:3: error: toString() in Shout
  cannot override toString() in Object
  attempting to assign weaker access
  privileges; was public
```

**¿Por qué?**

Cualquier código con una referencia a un `Object` puede llamar a `toString()`. Eso tiene que seguir valiendo si el objeto resulta ser un `Shout`.

# `protected`

| Modificador | la clase | el paquete | subclase en otro paquete | todos |
|---|---|---|---|---|
| `private` | sí | no | no | no |
| (ninguno) | sí | sí | no | no |
| `protected` | sí | sí | sí* | no |
| `public` | sí | sí | sí | sí |

- `protected long balance` dejaría a cualquier subclase romper el invariante: casi tan abierto como `public`.
- Es parte de la interfaz que se ofrece a quien extienda la clase. Úsenlo poco; mejor en métodos que en campos.

# * `protected` entre paquetes

savings/Savings.java · resumen

```java
import bank.Account; // protected balance
class Savings extends Account {
  void addInterest() {
    balance += balance / 100; // this
  }
  void copyFrom(Savings other) {
    balance = other.balance;  // Savings
  }
  void drain(Account other) {
    other.balance = 0;        // error
  }
}
```

- Desde otro paquete, la subclase usa el miembro protegido de su propio tipo: `this` u otro `Savings`.
- No a través de un `Account` cualquiera: podría ser de otra subclase. *balance has protected access in Account.*
- En el mismo paquete no hay restricción.

# `final`: hasta aquí llega la herencia

- Método `final`: no admite override. Clase `final`: no se extiende.
- `String` es `final`: nadie puede hacer un `String` que cambie o cuyo `length()` mienta.
- `Fraction` también; los `record`, siempre.

Name.java

```
1  public class Name extends
       String {
2  }
```

bash · recortada

```
Name.java:1: error: cannot
  inherit from final String
```

**Diseñar para la herencia cuesta**: decidir qué es `protected` y de qué se puede hacer override. Si una clase no se pensó para eso, `final`.

SECCIÓN 04

# La clase `Object`

04



# Lo que todo objeto entiende

| Método | Versión de `Object` | Override |
|---|---|---|
| `String toString()` | clase @ número hexadecimal | casi siempre |
| `boolean equals(Object o)` | `this == o`: identidad | en clases de valor |
| `int hashCode()` | depende del objeto | junto con `equals` |

# toString

jshell · ClockTime sin toString

```
jshell> ClockTime t =
   ...> new ClockTime(9, 30)
t ==> ClockTime@31221be2
```

Fraction.java

```java
@Override
public String toString() {
    return den == 1 ? Long.toString(num)
                    : num + "/" + den;
}
```

- `IO.println(x)` y `"..." + x` llaman a `x.toString()`.
- Heredado: la clase, `@` y el `hashCode()` en hexadecimal.
- Su override sirve para depurar y para leer las pruebas que fallan: `expected: <1/2> but was: <2/3>`.

# `equals`: igualdad de valor

Fraction.java

```java
@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (!(o instanceof Fraction f)) {
        return false;
    }
    return num == f.num
        && den == f.den;
}
```

- Heredado: `this == o`. Pero 1/2 y 2/4 deben ser iguales.
- Parámetro `Object`: la firma de `Object.equals`.
- `o instanceof Fraction f`: prueba el tipo y declara `f`. Con `null` da `false`. *(patrón: semana 8)*
- Comparar campos basta: el invariante deja la fracción reducida.

# Identidad e igualdad

FractionDemo.java

```java
Fraction a = new Fraction(1, 2);
Fraction b = new Fraction(2, 4);
IO.println("a == b: " + (a == b));
IO.println("a.equals(b): "
        + a.equals(b));
// ... contains y HashSet
```

bash

```
$ java -cp build FractionDemo
a = 1/2, b = 1/2
a == b: false
a.equals(b): true
hash: 33 y 33
list.contains(b): true
set: [1/2], tamaño 1
```

- Dos objetos, un valor.
- `contains` y `HashSet` usan `equals` (y `hashCode`).

# Error común: `equals(Fraction)`

EqualsOverloadDemo.java

```java
// sin @Override
public boolean equals(WrongFraction f) {
    return num == f.num && den == f.den;
}
// en main:
Object o = b;
a.equals(b)   // equals(WrongFraction)
a.equals(o)   // equals(Object)
```

bash

```
a.equals(b): true
a.equals(o): false
list.contains(b): false
```

- Es un overload: `equals(Object)` sigue siendo el de `Object`, y es el que llama `contains`.

**Con `@Override`, no compila**

```
error: method does not override or implement a method from a supertype
```

# El contrato de `equals`

Para referencias `x`, `y`, `z` distintas de `null`:

- **Reflexivo:** `x.equals(x)`.
- **Simétrico:** `x.equals(y)` si y solo si `y.equals(x)`.
- **Transitivo:** si `x.equals(y)` y `y.equals(z)`, entonces `x.equals(z)`.
- **Consistente:** sin cambios en los campos, siempre la misma respuesta.
- `x.equals(null)` es `false`, sin lanzar excepción.

> **¿Quién lo verifica?**
>
> El compilador, no. `ArrayList`, `HashSet` y `HashMap` confían en él. Lo verifican las pruebas.

# Una prueba por propiedad

FractionTest.java

```java
Fraction a = new Fraction(1, 2);
Fraction b = new Fraction(2, 4);
Fraction c = new Fraction(-3, -6);

@Test
void transitive() {
    assertTrue(a.equals(b));
    assertTrue(b.equals(c));
    assertTrue(a.equals(c));
}
```

FractionTest.java

```java
@Test
void notEqualToNull() {
    assertFalse(a.equals(null));
}
@Test
void notEqualToOtherType() {
    assertFalse(a.equals("1/2"));
}
```

- `c` con dos signos negativos: de paso, se prueba que el constructor normaliza.
- `"1/2"` se imprime igual, pero no es una fracción.

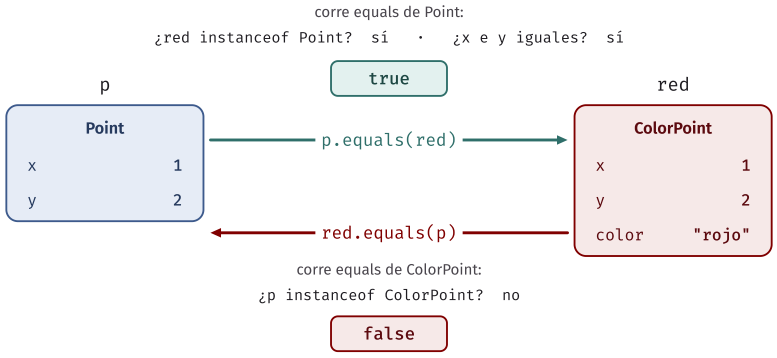
# Herencia y `equals`: `Point` y `ColorPoint`

Point.java y ColorPoint.java

```java
// en Point
return o instanceof Point p && x == p.x && y == p.y;
// en ColorPoint extends Point
return o instanceof ColorPoint cp && super.equals(cp) && color.equals(cp.color);
```

- Cada uno, por separado, parece razonable.
- ¿Qué da `new Point(1, 2).equals(new ColorPoint(1, 2, "rojo"))`? ¿Y al revés?

# La simetría se rompe




**Un ColorPoint es un Point, pero un Point no es un ColorPoint: la igualdad no es simétrica.**

# No hay arreglo perfecto con herencia

- **Ignorar el color si el otro es un `Point` común**: vuelve la simetría, pero rojo $=$ `p` $=$ azul y rojo $\neq$ azul: se pierde la transitividad.
- **Usar `getClass()` en lugar de `instanceof`**: el contrato se cumple, pero un `ColorPoint` nunca es igual a un `Point`. Se pierde la sustitución: un `HashSet<Point>` con $(1, 2)$ no contiene un `ColorPoint` en $(1, 2)$.

> **Una señal**
>
> Extender una clase de valor con un campo que participa en la igualdad no tiene solución buena. La semana 7: un `ColorPoint` que tiene un `Point` en vez de ser uno. Y por eso `Fraction` es `final`.

# `hashCode`: el compañero de `equals`

- `HashSet` y `HashMap` reparten los elementos en **buckets** según `hashCode()`.
- Al buscar, solo comparan con `equals` dentro de un bucket.

**El contrato**

- Objetos iguales $\Rightarrow$ mismo `hashCode`.
- Consistente mientras no cambien los campos.
- Distintos *pueden* coincidir; mejor que sea raro.

El `hashCode` de `Object` depende del objeto. Si una clase hace override de `equals` y no de `hashCode`, rompe la primera regla.

# Sin `hashCode`, el `HashSet` no encuentra

FRACTION · CON HASHCODE

```
set.add(1/2)          hash 33 → bucket 1
set.contains(2/4)     hash 33 → bucket 1 → equals                    true
```




CODE · EQUALS SIN HASHCODE

```
set.add(a)            hash 31221be2 → bucket 0
set.contains(b)       hash 4e50df2e → bucket 14, vacío               false
```




bucket = hash mezclado, módulo 16 · los hashes de Object son de ejemplo: dependen del objeto, no de sus campos

# El programa y la advertencia

bash

```
$ java -cp build HashDemo
a.equals(b): true
mismo hash: false
set.contains(a): true
set.contains(b): false
tamaño: 2
```

Un conjunto con dos elementos iguales.

bash · líneas partidas

```
warning: [overrides] Class Code
  overrides equals, but neither it nor
  any superclass overrides hashCode
```

- Con `-Werror`, detiene la compilación.
- `HashDemo` la calla con `@SuppressWarnings` porque es el tema del ejemplo. En código real: escribir `hashCode`.

# Escribir `hashCode`

Fraction.java

```java
@Override
public int hashCode() {
    return 31 * Long.hashCode(num)
             + Long.hashCode(den);
}
```

- Con los mismos campos que usa `equals`; ni uno más.
- Atajos: `Objects.hash(...)` y `Objects.equals(a, b)`.
- Los `record` traen los tres métodos hechos.

Para 1/2: $31 \cdot 1 + 2 = 33$.

**Error común: un campo que cambia**

Si cambia un campo que usa `hashCode` mientras el objeto está en un `HashSet`, queda en el bucket viejo y ya no se encuentra.

# ¿Valor o entidad?

**Valor**

Fracción, dinero, una hora, un punto. Se define por sus componentes: `equals` los compara todos. Suele ser inmutable; a menudo, un `record`.

**Entidad**

Una cuenta, una persona, un producto. Sigue siendo la misma aunque cambie su estado: identidad de `Object`, o comparar solo un código que no cambia.

- `BankAccount` no hace override de `equals`: dos cuentas con el mismo saldo no son la misma.
- Para P1: ¿dos `Product` con el mismo código son el mismo producto?

SECCIÓN 05

# Laboratorio

05

# Laboratorio: jerarquías y el contrato de `equals`

1. Extender `BankAccount` (p. ej. `CheckingAccount` con comisión mensual): `super(...)`, un `@Override` y una llamada a `super`.
2. `toString`, `equals` y `hashCode` para `ClockTime`, con una prueba por propiedad del contrato (también `null` y otro tipo).
3. `Point`/`ColorPoint`: encontrar con una prueba la violación de simetría o de transitividad.
4. Mostrar con una prueba que un objeto con `equals` y sin `hashCode` no se encuentra en un `HashSet`.

Código inicial en `code/week-05`; `FractionTest` sirve de modelo.

# make test

bash · salida recortada

```
$ make test
...
== HashDemo
a.equals(b): true
mismo hash: false
...
[         5 containers successful ]
[        18 tests found           ]
[        18 tests successful      ]
[         0 tests failed          ]
```

- `SavingsAccountTest`: lo heredado, lo nuevo y los overrides.
- Una prueba verifica algo que no está en `SavingsAccount`: el código en blanco lo rechaza `BankAccount`, vía `super(id)`.
- `FractionTest`: el contrato de `equals` y `hashCode`.

# Resumen

- **Herencia** = relación *es un*. `extends`: usar, agregar y **hacer override**.
- Los campos `private` heredados están en el objeto, pero la subclase usa los métodos públicos.
- Los constructores no se heredan; primero `super(...)`. Terminan desde `Object` hacia abajo.
- Override = misma firma, con `@Override`; overloading = otros parámetros. `protected` con moderación; `final` cierra.
- Toda clase desciende de `Object`. `toString` siempre; `equals(Object)` en clases de valor, cumpliendo su **contrato**.
- `Point`/`ColorPoint`: herencia y `equals` no se llevan bien.
- Iguales $\Rightarrow$ mismo `hashCode`, con los mismos campos que `equals`.

# Para la próxima clase

- Semana 6: Eck 5.5.4 (*Polymorphism*) y relectura de 5.5.3; luego 5.5.5 (*Abstract Classes*).
- **P1** se entrega la semana 6: revisen si `Product` necesita `equals`, `hashCode` y `toString`.
- Ejercicios de los apuntes de la semana 5.

**Pregunta para pensar**

`BankAccount acc = new SavingsAccount("A-001", 2);` ¿Compila? Si compila, ¿qué `withdraw` se ejecuta con `acc.withdraw(100)`: el de `BankAccount` o el de `SavingsAccount`?