Tema 3 – Herencia en Java – Parte 2

Programación Orientada a Objetos

Grado en Ingeniería Informática

DIS
Departamento de Informática y Sistemas

UNIVERSIDAD DE MURCIA

Contenido

- Restringir la herencia.
- Clase `Object`.
- Autoboxing.
- Igualdad de objetos.
- Representación textual de objetos.
- Copia de objetos.

Restringir la herencia

- ¿Es seguro permitir que los subtipos redefinan cualquier método?
- La redefinición incorrecta del método **`situar`**`(Punto)` de la `Burbuja` podría comprometer la consistencia y seguridad de la aplicación.

- En Java se puede aplicar el **modificador final** a un método para indicar que no puede ser redefinido.
- Asimismo, el modificador final es **aplicable a una clase** indicando que no se puede heredar de ella.

Restringir la herencia

```
public class Burbuja {
   …
   public final void situar(Punto posicion) {
      region.desplazar(posicion);
   }
   …
}
```

- El método `situar` es el mismo para todas las burbujas, **no se puede redefinir**.

Restringir la herencia

```
public final class String … {
  …
}
```

- No se puede crear una subclase de la clase `String`.
- Un `Record` es implícitamente `final`, de manera que no se puede extender explícitamente un registro.

Revisión redefinición de métodos

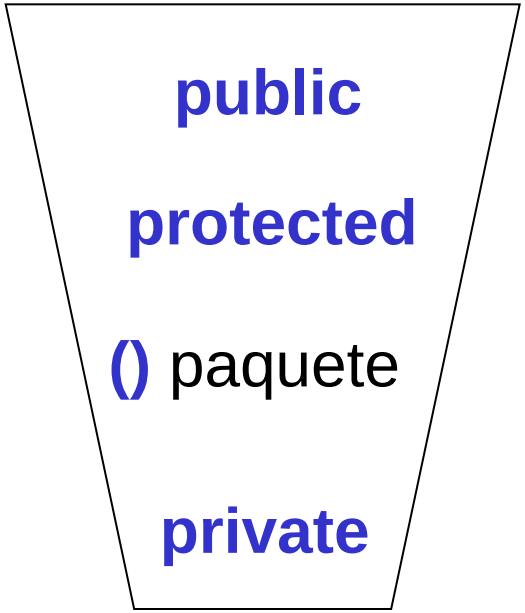
- Al redefinir un método podemos cambiar su implementación (refinamiento, reemplazo).

- También se puede **cambiar el nivel de visibilidad** de la declaración para **incrementarlo**.

- Es posible **cambiar el tipo de retorno** a otro más específico (descendiente) → **Regla covariante**.
  - En Burbuja `>> Burbuja copia() {…}`
  - En BurbujaLimitada `>> BurbujaLimitada copia() {…}`

Revisión redefinición de métodos

- En Java, los **niveles de visibilidad son incrementales**



public → todo el código

protected → clase + paquete + subclases

(*nada*) → clase + paquete

private → clase

¿Podemos cambiar la visibilidad de un método privado para hacerlo público?

Clase `Object`

- El lenguaje Java define la clase **Object** como raíz de la jerarquía de todas las clases del lenguaje.

- **Todas las clases heredan** directa o indirectamente **de la clase Object**:
  - Si una clase no hereda de ninguna otra, el compilador añade `extends Object` a su declaración.

- ➔ Una variable de tipo **`Object`** puede referenciar a cualquier tipo del lenguaje:
  - Objetos creados a partir de clases.
  - Tipos primitivos gracias al *autoboxing*.

Clase `Object` y Autoboxing

- El *autoboxing* es un mecanismo automático encargado de **convertir tipos primitivos en objetos y viceversa**.

- De este modo, una referencia de tipo `Object` puede referenciar a cualquier tipo de datos.

- Conversión de tipos primitivos a objetos:
  - Java construye *objetos envoltorio* donde almacenar los valores primitivos.
  - Para cada tipo primitivo hay una clase envoltorio: `Integer` para `int`, `Double` para `double`, etc.

Clase `Object` y Autoboxing

- La conversión entre el tipo primitivo y el tipo envoltorio es automática.

```
Integer valor = 3;
int entero = valor;
```

- En caso de asignar un valor primitivo a `Object` también se produce autoboxing. Sin embargo, en este caso hay que hacer un casting al tipo envoltorio para recuperar el valor.

```
Object obj = 3;
int entero = (Integer) obj;
```

Métodos clase `Object`

- En Java, la clase **`Object`** incluye las **características comunes a todos los objetos**:
  - **`public boolean equals`**`(Object obj)`

    ➔ Igualdad de objetos
  - **`protected`** `Object` **`clone`**`()`

    ➔ Ofrece una copia *superficial* del objeto.
  - **`public`** `String` **`toString`**`()`

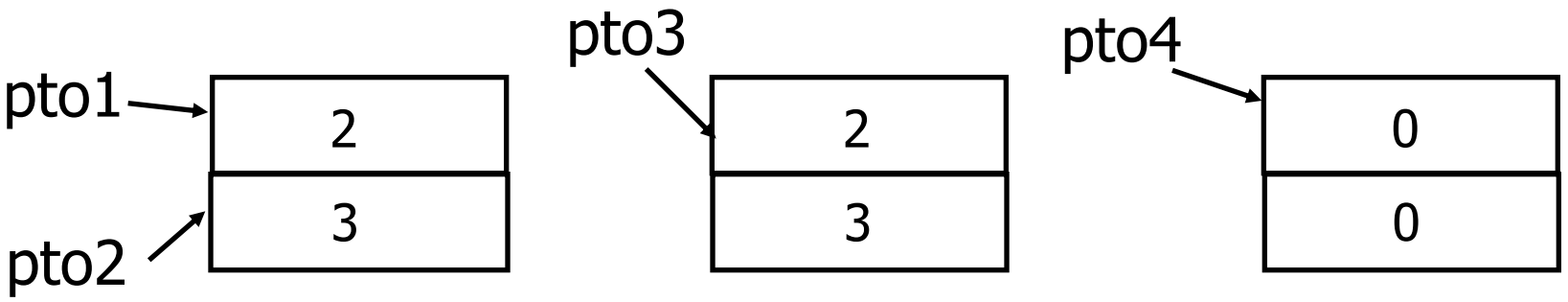
    ➔ Representación textual de un objeto
  - **`public final`** `Class` **`getClass`**`()`

    ➔ Clase a partir de la que ha sido instanciado un objeto.
  - **`public int hashCode`**`()`

    ➔ Código *hash* utilizado en las colecciones.

Igualdad de objetos

- El operador == se utiliza para consultar la **identidad de referencias** ➔ dos referencias son idénticas si contienen el mismo *oid*.



- Identidad entre referencias:
  - `pto1 == pto2; // True`
  - `pto1 == pto3; // False`
- El **método equals** permite implementar la igualdad de objetos.

Método `equals`

- La **implementación en la clase `Object`** consiste en **comprobar la identidad** del objeto receptor y el parámetro.

```
public boolean equals(Object obj) {
    return this == obj;
}
```

  - Por tanto: `pto1.equals(pto3); // False`
- Es necesario **redefinir el método equals** en las clases donde necesitemos la operación de igualdad.
- Sin embargo, hay que elegir la **semántica** de igualdad más adecuada para la clase.

Tipos de igualdad

- **Tipos de igualdad**:
  - **Superficial**: los campos primitivos de los dos objetos son iguales y las referencias a objetos *idénticas* (comparten los mismos objetos, *aliasing*).
  - **Profunda**: los campos primitivos son iguales, NO comparten los objetos, aunque las referencias son iguales (**equals**).
  - **Adaptada** a las necesidades de la aplicación.

Método `equals` en `Punto`

- Ejemplo de **igualdad superficial**

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;

    Punto otro = (Punto) obj;

    return this.x == otro.x && this.y == otro.y;
}
```

Explicación: Método `equals` en `Punto`

- 1. Caso trivial: dos objetos con el mismo oid siempre serán iguales.
- 2. Caso trivial: el parámetro es la referencia nula.
- 3. Comprueba que el **tipo de los objetos** sea el mismo (método `getClass`).
  - Ejemplo: si suponemos que `Punto3D` hereda de `Punto`, no sería correcto considerar la igualdad entre objetos de esas clases.
- 4. Realiza el casting del parámetro (**siempre de tipo `Object`**) para poder comparar sus atributos
  - La consulta del tipo (`getClass`) garantiza el casting correcto.
- 5. Comparación de los atributos
  - Nota: un objeto tiene acceso a los atributos privados de otro de su misma clase.

Método `equals` en `Circulo`

- Ejemplo de **igualdad profunda**

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Circulo otro = (Circulo) obj;

    return this.centro.equals(otro.centro)
           && this.radio == otro.radio;
}
```

Método `equals` en `Burbuja`

- Ejemplo de **igualdad adaptada**:
  - De acuerdo a la *semántica* del problema, NO se tienen en cuenta las propiedades *velocidad actual* y *explotada*.

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Burbuja otra = (Burbuja) obj;

    return this.region.equals(otra.region)
        && this.velocidadLimite == otra.velocidadLimite;
}
```

Método `equals` en herencia

- El método `equals()` heredado en una subclase es correcto si:
  - No tiene nuevos atributos.
  - Los nuevos atributos no se tienen en cuenta en la igualdad.
- Si fuera necesario redefinir, debe reutilizarse la versión heredada:

```
public boolean equals(Object obj) {

  if (super.equals(obj) == false) return false;

   BurbujaLimitada otro = (BurbujaLimitada)obj;

   return …; // compara atributos propios
}
```

Método `hashCode`

- Devuelve un número que representa el código de dispersión (código *hash*) de un objeto.

- Es utilizado para almacenar los objetos en colecciones cuya implementación se basa en una **tabla de dispersión** (`HashMap` y `HashSet`, se estudian en tema 4).

- **Importante**: la implementación del `hashCode()` tiene que ser **consistente** con la implementación del `equals()`
  - **Si** `o1.equals(o2) es true` **entonces** `o1.hashCode() == o2.hashCode()`
  - Lo contrario no tiene por qué ser cierto.

Método `hashCode`

- La implementación en la clase `Object` del método `hashCode()` deriva el código de dispersión de la dirección de memoria del objeto.

- **Importante**: si en una clase se redefine el método `equals()` también hay que redefinir el método `hashCode()` para que las implementaciones sean consistentes.

- La implementación de los métodos `equals()`/`hashCode()` es **consistente** si está basada en los mismos atributos.

Método `hashCode`

- En la redefinición del método `hashCode()` se calcula el código de dispersión aplicando un algoritmo que trata de reducir la probabilidad de que objetos distintos tengan el mismo código (*colisión*).

- El **algoritmo** propuesto combina el código *hash* de cada uno de los atributos.
  - Si es primitivo de tipo entero, se utiliza su valor.
  - Para el resto de atributos de tipo primitivo se utilizan las clases envolventes (por ejemplo `Double` para `double`), para calcular el código *hash*.
  - Para atributos de tipo objeto (incluyendo arrays y enumerados) se utiliza el método `hashCode()`.

Método `hashCode` en `Circulo`

- Ejemplo de redefinición en la clase `Circulo`.
  - **Importante**: se utilizan los mismos atributos que en la igualdad.

```
@Override
public int hashCode() {
    int primo = 31;
    int result = 1;
    result = primo * result + centro.hashCode();
    result = primo * result + radio;
    return result;
}
```

Método `hashCode` en `Cuenta`

- Ejemplo de redefinición en la clase `Cuenta`:
  - El atributo `saldo` es un número real, `estado` es un enumerado y `ultimasOperaciones` es un array.
  - La clase `java.utils.Arrays` ofrece un método que calcula el código de dispersión de un array.

```
@Override
public int hashCode() {
 int primo = 31;
 int result = 1;
 …
 result = primo * result + new Double(saldo).hashCode();
 result = primo * result + ((estado == null) ? 0 :
                            estado.hashCode());
 result = primo * result + Arrays.hashCode(ultimasOperaciones);
 return result;
}
```

Método `hashCode`

- La implementación del algoritmo es tedioso y propenso a errores.
- La clase `Objects` ofrece métodos de utilidad (`static`) para trabajar con objetos, tolerantes con el valor `null`.
- Para implementar el método `hashCode` se puede delegar en el método `Objects.hash`.

Método `hashCode` en `Circulo`–v2

- Ejemplo de redefinición en la clase `Circulo`.

```
@Override
public int hashCode() {

   return Objects.hash(centro, radio);

}
```

Método `hashCode` en herencia

- El método `hashCode()` se redefine en una subclase si se ha redefinido el método `equals()`
  - ➔ **consistencia** de las implementaciones.
- La redefinición debe reutilizar la versión heredada.

```
public int hashCode() {
  int primo = 31;
  int result = super.hashCode();

  // añadir atributos utilizados en equals() redefinido
  result = primo * result + Objects.hash(…);

  return result;
}
```

Método `equals` en `Circulo`-v2

- Los métodos de la clase `Objects` también se pueden utilizar para implementar la igualdad.

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Circulo otro = (Circulo) obj;

    return Objects.equals(this.centro,otro.centro)
           && this.radio == otro.radio;
}
```

Método `toString`

- El método devuelve una cadena de texto que “representa” al objeto.

- La implementación de este método en la clase `Object` devuelve el nombre de la clase del objeto seguida del símbolo ‘`@`’ y un número hexadecimal correspondiente al código hash del objeto.

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

Método `toString` en `Punto`

- En la redefinición del método **`toString()`** se aplica un patrón de implementación.
- La llamada **`getClass().getName()`** retorna el nombre de la clase de la instancia actual.
- Es recomendable utilizar siempre esta llamada (en vez de poner “Punto”, por ejemplo) para que la implementación sea **heredable**.

```
@Override
public String toString() {
    return getClass().getName()
            + " [x=" + x + ", y=" + y + "]";
}
```

Método `toString` en `Circulo`

- El resultado del método es una cadena resultado de concatenar el nombre de la clase del objeto con sus propiedades.
- En la clase `Circulo`, el atributo `centro` es un objeto (`Punto`).
- Al concatenarlo a la cadena, se aplica automáticamente el método `toString()`.

```
@Override
public String toString() {
    return getClass().getName()
            + " [centro=" + centro
            + ", radio=" + radio
            + ", perímetro=" + getPerimetro() + "]";
}
```

Método `toString` en `Cuenta`

- Para mostrar la representación `toString()` de un array es necesario utilizar un método de la clase `java.util.Arrays`.

```
@Override
public String toString() {
  return getClass().getName() +
         "[codigo = " + codigo +
         ", titular = "+ titular +
         ", saldo = "+ saldo +
         ", ultimasOperaciones = " +
               Arrays.toString(ultimasOperaciones) +
         "]";
}
```

Método `toString` en herencia

- El método `toString()` debe ser redefinido en una subclase si añade nuevas propiedades.
- En caso de redefinición, reutilizar la versión heredada:

```
  public String toString() {
   // añade a la cadena atributos propios
   return super.toString() + "[ … ]";
}
```

- La llamada `getClass().getName()` siempre mostrará correctamente el nombre de la clase:
  - `getClass()` retorna el tipo de la instancia actual.

Métodos de `Object` en `Record`

- Cuando se declarar un `Record` se generan automáticamente los métodos `equals`, `hashCode` y `toString`.

- `hashCode y equals`: dos instancias de un registro son iguales si son del mismo tipo y son iguales (`equals`) todos los valores de sus componentes.

- `toString`: incluye la representación textual de todos sus componentes, precedido por el nombre del registro.

Copia de objetos

- La **asignación de referencias** (=) copia el *oid* del objeto y no la estructura de datos.

- Para obtener una copia de un objeto hay **programar un método que construya una copia**.

- El método **`clone`** declarado en la clase **`Object`** (es heredado) implementa una copia campo a campo de los objetos (***copia superficial***).

- El programador de una clase que ofrezca la copia debe hacer visible el método (está declarado `protected`) y adaptar la copia superficial si fuera necesario.

Tipos de copia

- **Tipos** de copia:
  - **Copia superficial**: los campos de la copia son exactamente iguales a los del objeto receptor, y por tanto, con los atributos de tipo objeto se produce *aliasing*.
  - **Copia profunda**: los campos primitivos de la copia son iguales y las referencias a objetos son copias (no hay *aliasing*).
  - **Adaptada** a las necesidades de la aplicación.

Redefinición del método `clone`

- Signatura del método en la clase `Object`:

```
protected Object clone()
                  throws CloneNotSupportedException
```

- Para hacerlo visible hay que cambiar la visibilidad a **`public`**.
- Para adaptarlo a la clase que se está copiando hay que aplicar la ***regla covariante*** y cambiar el tipo de retorno.
- Para que no ocurra la excepción, la clase que se quiere copiar tiene que indicar que se permite la copia (**`implements`** `Cloneable`).
- Ocultamos la excepción, que sabemos que no va a ocurrir si la clase es *cloneable*.

Copia de `Punto`

- Clase `Punto`:
  - Solo tiene atributos de tipo primitivo.
  - Una copia de un punto debe tener los mismos valores.
  - Por tanto, sería una **copia superficial**.

- En las siguientes diapositivas se redefine el método `clone` haciendo uso de un método de soporte (privado) encargado de hacer la invocación a la copia superficial (método `clone` heredado).
- Se recomienda programar el método de soporte para aislar los detalles técnicos del uso de la copia superficial.

Copia de `Punto`

```
public class Punto implements Cloneable {
    …
   @Override
   public Punto clone() {

      Punto copia = copiaSuperficial();

     // Aquí pondríamos las adaptaciones
     // En el caso del punto, no es necesario
     // Solo queremos la copia superficial

      return copia;
   }
}
```

Método de soporte para la copia

- El método de soporte se encarga de realizar convenientemente la llamada a la versión heredada de clone.
- Cabe destacar que se maneja la excepción que sabemos que no va a ocurrir si la clase es *cloneable*.

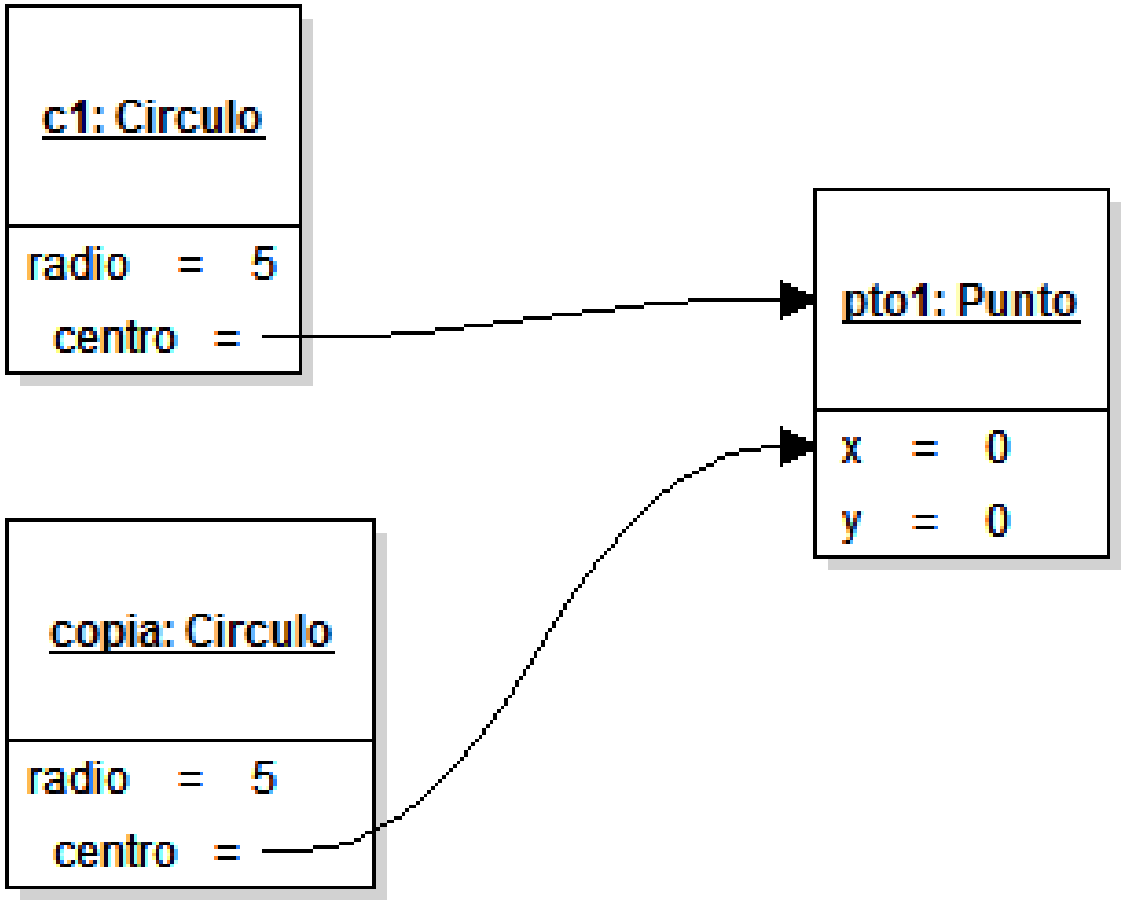
```
private Punto copiaSuperficial() {
    try {
        Punto copiaSuperficial = (Punto) super.clone();
        return copiaSuperficial;
    }
    catch (CloneNotSupportedException e) {
        // No sucede si en la cabecera de la clase
        // indicamos "implements Cloneable"
        System.err.println("La clase no es cloneable");
    }
    return null; // no se ha podido obtener la copia
}
```

Copia de `Circulo`

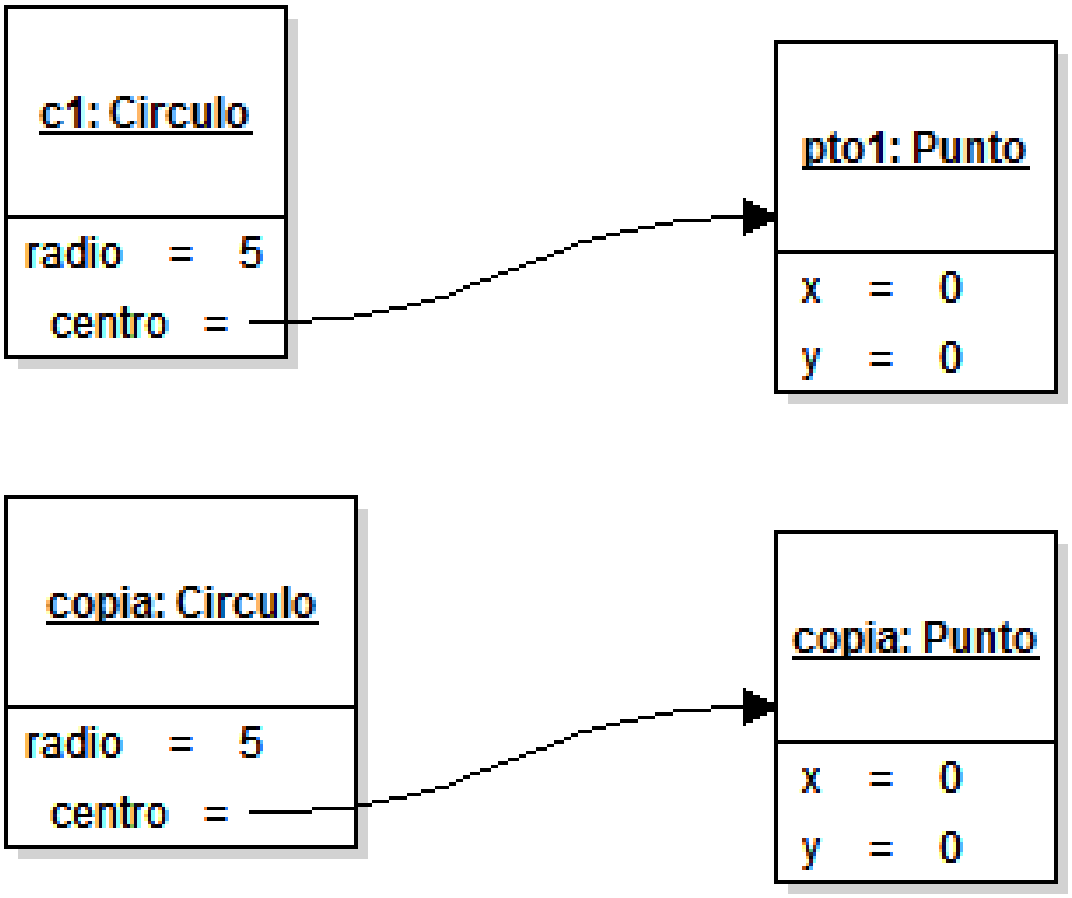
- Clase `Circulo`:
  - Una copia de un círculo tendrá el mismo radio y el mismo centro (`Punto`).
  - Sin embargo, no deben compartir el mismo objeto centro. Tendría que ser una **copia del centro**.
  - Así pues, deberíamos implementar una **copia profunda**.

- En las siguientes diapositivas se contrasta la copia superficial del círculo (incorrecta) y la copia profunda.

Copia superficial de `Circulo`



Copia profunda de `Circulo`



Copia de `Circulo`

- Implementamos una **copia profunda** en la clase `Circulo`:
  - Evitamos el *aliasing* del centro obteniendo una copia:

```
@Override
public Circulo clone() {

    Circulo copia = copiaSuperficial();

    // Adaptamos la copia superficial

    // Corregimos el aliasing incorrecto

    copia.centro = this.centro.clone();

    return copia;
}
```

Método de soporte para la copia

- Aplicamos el mismo patrón de código que en `Punto`:

```
private Circulo copiaSuperficial() {
    try {
        Circulo copiaSuperficial = (Circulo) super.clone();
        return copiaSuperficial;
    }
    catch (CloneNotSupportedException e) {
        System.err.println("La clase no es cloneable");
    }
    return null; // no se ha podido obtener la copia
}
```

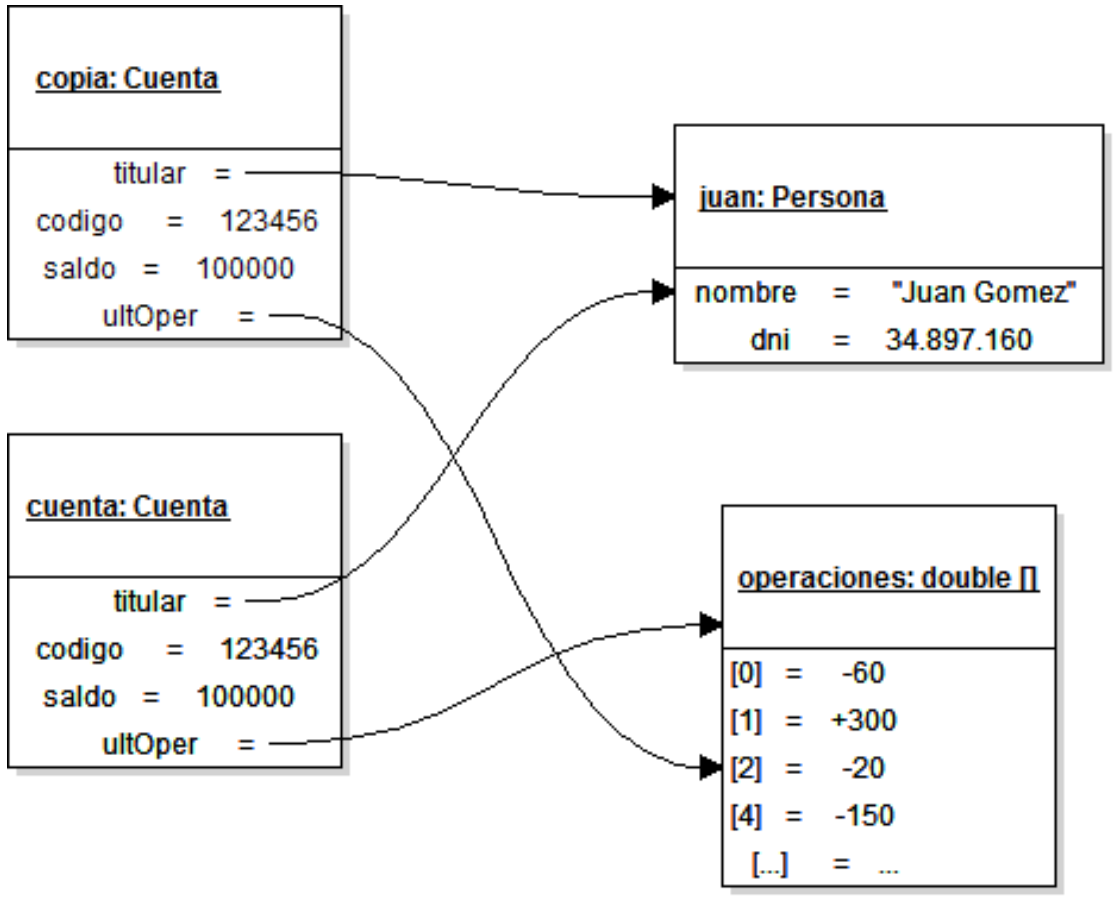
Copia de `Cuenta`

- Clase `Cuenta`:
  - Una copia de una cuenta debería compartir el objeto que representa al titular.
  - Además, debería evitar el aliasing del array de últimas operaciones.
  - La nueva cuenta debería tener un número de cuenta distinto.
  - En definitiva, las copias superficial y profunda no son adecuadas. Debemos programar una **copia adaptada**.

- En las siguientes diapositivas se contrasta la copia superficial (incorrecta), la copia profunda (incorrecta) y la copia adaptada (correcta).

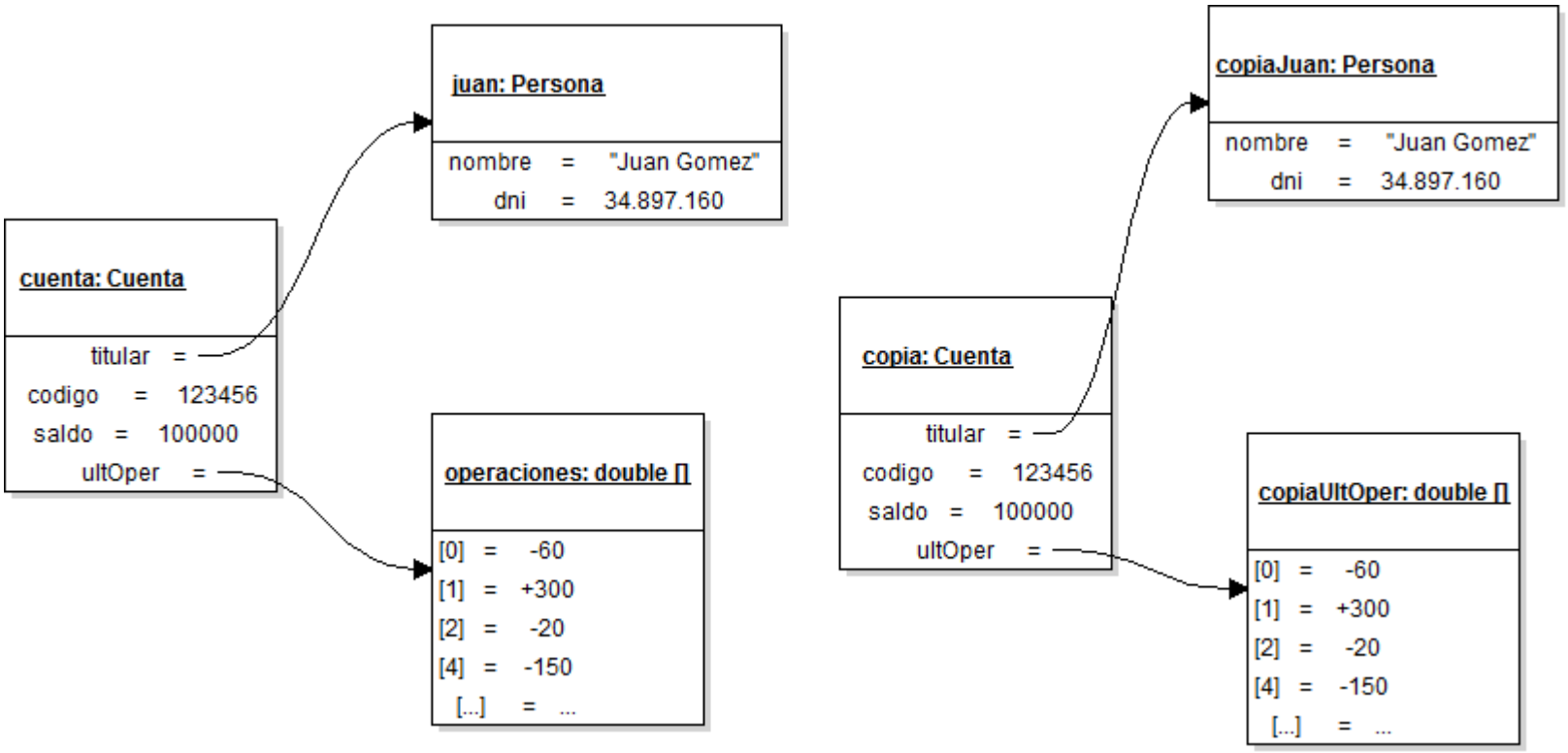
Copia superficial de `Cuenta`

- **Copia superficial**:
  - *Aliasing* **incorrecto** al compartir las últimas operaciones.
  - No deberían tener el mismo código



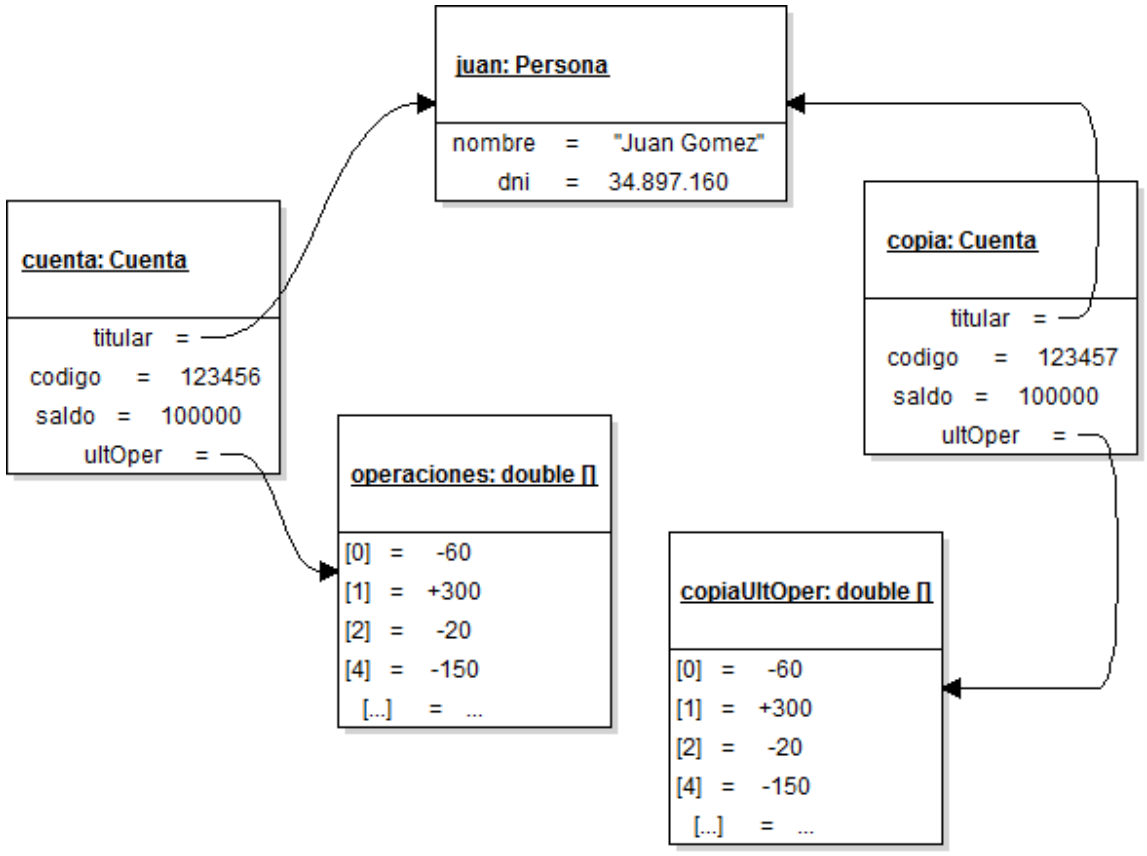
Copia profunda de `Cuenta`

- **Copia profunda**:
  - No tiene sentido duplicar el objeto persona y que tengan el mismo código.



Copia correcta de `Cuenta`

- **Copia adaptada**: cumple los requisitos de la aplicación



Método `clone` en `Cuenta`

- Aunque la copia de la clase `Cuenta` es **adaptada**, **siempre partimos de la copia superficial**.
- Para copiar el array utilizamos el método `copyOf` de la clase `java.util.Arrays`.

```
public class Cuenta implements Cloneable{
  …
  @Override
  public Cuenta clone(){
    Cuenta copia = copiaSuperficial();
    // Adaptaciones
    copia.codigo = ++ultimoCodigo;
    copia.ultimasOperaciones =
       Arrays.copyOf(ultimasOperaciones, ultimasOperaciones.length);
    return copia;
  }
}
```

Copia de objetos en herencia

- Redefinir el método `clone` a partir de la copia superficial que ofrece `Object` hace que la implementación sea **heredable**
  - La llamada al método `clone()` de `Object` siempre retorna **un objeto igual a la instancia actual**.
  - Una subclase sólo necesitaría redefinir el método si los nuevos atributos que aporta no son copiados correctamente con la copia superficial (por ejemplo, *aliasing* incorrecto).

Copia de objetos en herencia

- En **herencia**, aunque el método `clone` heredado sea correcto, conviene redefinirlo aplicando la **regla covariante** y llamar a la versión del padre:

```
public class BurbujaLimitada extends Burbuja {
   …
   public BurbujaLimitada clone(){

     return (BurbujaLimitada)super.clone();
   }
}
```

- **Nota**: el casting siempre será correcto, ya que la copia superficial siempre garantiza que el objeto que retorna es igual que la instancia actual, y por tanto, tiene el mismo tipo.

Método `getClass()` vs operador `instanceof`

- El método **`getClass()`** retorna el tipo dinámico de la referencia (**variable**).
- En general **no es recomendado** su uso. En su lugar es preferible el operador **`instanceof`** para consultar la compatibilidad de tipos.
- **Ejemplo**: la consulta con **`getClass()`** deja fuera las burbujas crecientes, que también son burbujas débiles (**incorrecto**).

```
public void simulador(Burbuja burbuja) {
    ...
    if (burbuja.getClass() == BurbujaDebil.class) {
      ...
}
```