

Clase Object

🧑 Clase Persona

● hashCode()

● toString()

● equals()

Sobrescritura personalizada de equals()

🛑 Seguridad ante todo

Sobrescritura personalizada mejorada de equals()

★ Sobrescritura rápida de equals()

● compareTo()

Sobrescritura personalizada de compareTo()

★ Sobrescritura rápida de compareTo()

Clase Object

En Java, **todos los objetos** heredan de la clase `Object`. Esto significa que cualquier clase que crees en Java, ya sea directamente o a través de herencia, tiene algunos métodos básicos que provienen de `Object`. Entre estos métodos, se encuentran algunos que son muy importantes para personalizar el comportamiento de tus clases, como `toString()`, `equals()` y `compareTo()`.

¿Por qué es esto importante? Al sobrescribir estos métodos, puedes hacer que tus clases sean más útiles, ya que el comportamiento por defecto de la clase `Object` puede no ser el que necesitas. A continuación, veremos estos métodos y cómo se pueden sobrescribir para ajustarlos a nuestras necesidades.

Important

Solo tendrás que preocuparte de sobrescribir estos métodos en tus propias clases. Los tipos primitivos y la clase `String` ya tienen sobrescritos los métodos `toString()`, `equals()` y `compareTo()`, por lo que puedes usarlos directamente para comparaciones y conversiones sin necesidad de redefinirlos.

Clase Persona

En todos los ejemplos usaremos una clase simple llamada `Persona`. Siempre se pondrá todo el código necesario para los ejemplos concretos, pero para que te hagas una idea de todo el código de la clase, aquí la tienes completa:

```
1 public class Persona {
2     private String nombre;
3     private int edad;
4
5     public Persona(String nombre, int edad) {
6         this.nombre = nombre;
7         this.edad = edad;
8     }
9
10    public String getNombre() {
11        return nombre;
12    }
13
14    public void setNombre(String nombre) {
15        this.nombre = nombre;
16    }
17
18    public int getEdad() {
19        return edad;
20    }
21
22    public void setEdad(int edad) {
23        this.edad = edad;
24    }
25 }
```

● hashCode()

El método `hashCode()` es un método de la clase `Object` que devuelve un valor entero, conocido como el **código hash** del objeto. Sirve para identificar de forma unívoca al objeto dentro de la memoria. No hay dos objetos distintos que tengan el mismo hashcode.

```
1 Persona persona = new Persona("Juan", 25);
2 System.out.println(persona.hashCode()); //Imprime: 1791741888
```

Si imprimimos un objeto sin llamar a ningún método, Java llama de forma implícita al método `toString()` que tienen los objetos:

```
1 Persona persona = new Persona("Juan", 25);
2 System.out.println(persona);           // Imprime: Persona@6acbcfc0
3 System.out.println(persona.toString()); // Imprime: Persona@6acbcfc0
```

Esto indica que el método `toString()` por defecto de la clase `Object` se ha llamado, que devuelve el nombre de la clase (`Persona`) seguido del símbolo `@` y el hashcode del objeto en formato hexadecimal (`6acbcfc0`).

Esto no proporciona información útil sobre los atributos del objeto y es por eso que se recomienda sobrescribir el método `toString()` para obtener una representación más clara y significativa.

💡 Tip

Aunque el método `hashCode()` no suele usarse mucho, es conveniente saber que existe para entender mejor los conceptos de POO y de identidad de los objetos.

● toString()

El método `toString()` se utiliza para obtener una **representación en texto** de un objeto. Por defecto, este método devuelve el nombre de la clase y el código hash del objeto. Sin embargo, en la mayoría de los casos, querrás que este método devuelva una descripción más clara y útil de tu objeto.

Ejemplo de sobrescritura de `toString()`:

```
1 @Override
2 public String toString() {
3     return "Me llamo " + this.nombre + ", y tengo " + this.edad + " años.";
4 }
```

Sobrescribir `toString()` te permite personalizar el texto que representa tu objeto cuando lo imprimes o lo conviertes a una cadena de texto.

📌 Important

La anotación `@Override` se usa en Java para **indicar explícitamente que un método está sobrescribiendo un método de una clase padre o una interfaz**. Aunque no es obligatoria, tiene varias ventajas importantes:

1. **Verificación del compilador:** Al usar `@Override`, el compilador verifica que realmente estás sobrescribiendo un método existente. Si cometes un error en el nombre, la firma del método o el tipo de retorno, el compilador te dará un error. Esto ayuda a evitar errores comunes.
2. **Mejora la legibilidad:** Al incluir `@Override`, dejas claro que el método tiene su origen en una clase o interfaz superior, lo que facilita la lectura del código y su comprensión.
3. **Mantenimiento del código:** Si en el futuro alguien modifica la clase base o interfaz y cambia el método que creías sobrescribir, la anotación `@Override` hará que el compilador detecte el error, evitando comportamientos inesperados.

Aunque tienes libertad para personalizar el método `toString()` como quieras, es recomendable seguir un **formato estándar** que haga más fácil la lectura y comprensión de los objetos en diferentes contextos. La forma más común y recomendada es representar el objeto así:

```
1 @Override
2 public String toString() {
3     return "Persona{nombre='" + nombre + "', edad=" + edad + '}';
4 }
```

```
1 Persona{nombre='Max Power', edad=25}
```

Este formato muestra el nombre de la clase seguido de `{}` y una lista de sus atributos clave con sus valores correspondientes separados por comas. Este enfoque es útil porque es claro, fácil de entender y facilita la depuración y el análisis del comportamiento de los objetos en el código, manteniendo un estilo coherente y profesional.

● `equals()`

El método `equals()` se usa para **comparar si dos objetos son iguales**. Por defecto, este método solo compara si dos objetos **son el mismo** en memoria (es decir, si son la misma instancia). Sin embargo, en muchos casos, querrás comparar objetos basándote en sus atributos.

Nosotros hacemos las comprobaciones que necesitemos y **finalmente tenemos que retornar `true` si hemos considerado que son iguales o `false` en caso contrario**.

Sobrescritura personalizada de `equals()`

```
1 @Override
2 public boolean equals(Object obj) {
3     Persona persona = (Persona) obj; // Hacemos casting para poder acceder a
    sus atributos
4     // La siguiente expresión dará true o false
5     return this.nombre.equals(persona.getNombre()) && this.edad ==
    persona.getEdad();
6 }
```

Al sobrescribir `equals()`, puedes definir **qué significa que dos objetos sean iguales**, comparando los atributos que tu quieras. Aquí en el ejemplo anterior hemos comparado que un objeto tenga el mismo nombre y edad que otro. Si se da ese caso retorna un `true`, o `false` en caso contrario.

Important

Tu decides qué comprobaciones quieres realizar para determinar que dos objetos son iguales. Por ejemplo, si tuviésemos el atributo `nif`, podríamos determinar que si sus `nif` son iguales, los objetos son iguales. A pesar de que tengan distinto `nombre` y/o `edad`.

Tenemos que hacer *casting* ya que el método debe recibir un objeto ya que así está definido en la clase `Object`. Gracias al polimorfismo podemos pasar un como parámetro una clase más concreta como `Persona` y después hacerle casting para poder acceder a sus métodos.

Seguridad ante todo

Pero hay un pequeño problema de seguridad. Como el método `equals()` puede recibir un `Object`, quién use ese método podría enviarnos cualquier objeto de cualquier clase. El problema viene al hacerle el casting. ¿Que pasaría intentamos hacer un **cast** de un objeto a una clase que no es compatible? **El programa lanzará una excepción de tipo `ClassCastException`** y terminará su ejecución.

Sobrescritura personalizada mejorada de `equals()`

```
1 @Override
2 public boolean equals(Object obj) {
3     if (this == obj) return true;
4     if (obj == null || this.getClass() != obj.getClass()) return false;
5     Persona persona = (Persona) obj;
6     return this.nombre.equals(persona.getNombre()) && this.edad ==
    persona.getEdad();
7 }
```

- Primero, **comparamos si los objetos a comparar son el mismo**. Si son el mismo, es obvio que son iguales, por lo que retornamos `true` y terminamos la ejecución.
- Segundo, **si el objeto recibido es `null`** retornamos un `false`.
- Tercero, **si los objetos a comparar son de distintas clases**, retornamos `false`.
- Y ya realizadas las comprobaciones, procedemos a hacer casting sin riesgos y a hacer las comprobaciones que queramos.

💡 Tip

🤖 Las sentencias `return` al principio simplifican el código ya que evitan tener que seguir haciendo comprobaciones cuando ya hemos determinado un resultado. Aún así, aquí tienes otra opción que hace lo mismo con un único `return`.

```
1  @Override
2  public boolean equals(Object obj) {
3      // Variable auxiliar donde guardar el resultado
4      boolean esIgual = false; // Por defecto son distintos
5
6      //Si son el mismo
7      if (this == obj) {
8          esIgual = true;
9      } else if (obj != null && this.getClass() == obj.getClass()) {
10         //Ya sabemos que no es nulo y son de la misma clase, por lo que
         hacemos el casting sin miedo
11         Persona persona = (Persona) obj;
12         //Si tienen el mismo nombre y edad, es que son iguales
13         if (this.nombre.equals(persona.getNombre()) && this.edad ==
         persona.getEdad()) {
14             esIgual = true;
15         }
16     }
17
18     return esIgual; // Solo tenemos un único return
19 }
```

★ Sobrescritura rápida de equals()

Una forma rápida de sobrescribir el método `equals()` en una clase es utilizando el método `toString()` de los objetos para comparar sus representaciones en cadena. Por ejemplo, si tenemos una clase `Persona` con atributos como `nombre` y `edad`, podríamos sobrescribir `equals()` de la siguiente manera:

```
1  @Override
2  public boolean equals(Object otro) {
3      return this.toString().equals(otro.toString());
4  }
```

Realmente lo que estamos comparando es que ambos `toString()` sean iguales. ES una forma simple y eficiente de comprobar que los objetos “sean iguales”. Si algún atributo es distinto cambiará la salida del `toString()` ya no coincidirá y dirá que son distintos.

Si además queremos hacer las [comprobaciones de seguridad](#) explicadas antes, sería:

```

1  @Override
2  public boolean equals(Object obj) {
3      if (this == obj) return true;
4      if (obj == null || getClass() != obj.getClass()) return false;
5      Persona persona = (Persona) obj;
6      return this.toString().equals(persona.toString());
7  }

```

💡 Tip

El usar el `equals()` de la clase `String` en nuestro propio `equals()` nos ahorra tener que ir comparando atributo por atributo para determinar si los objetos son iguales.

● compareTo()

La sobrescritura del método `compareTo()` es una de las claves para permitir que los objetos de una clase puedan **ordenarse** de forma natural, y se hace cuando implementamos la interfaz `Comparable`.

Si intentas comparar dos objetos de una clase cualquiera utilizando **operadores de igualdad** como `!=` o `==`, te encontrarás con que no provocará ningún error, pero lo que se está comparando en realidad es la **referencia** en memoria (hashcode) y no el contenido de los objetos (sus atributos).

⚠ Warning

Los operadores `>`, `<`, `>=`, `<=` **si provocarán un error de compilación** si usan en expresiones con objetos porque los operadores relacionales están definidos solo para tipos primitivos numéricos o caracteres.

Sabemos comparar atributos de un objeto, pero ¿podemos comparar dos objetos directamente para saber cual es "mayor" o "menor"? Esto es indispensable para poder ordenar una lista de objetos.

El método `compareTo()` se utiliza para **comparar dos objetos** de la misma clase y determinar su orden. Devuelve un número entero que indica si el objeto actual (el que llama al método) es "menor", "igual" o "mayor" que el objeto que se pasa como parámetro. El resultado se interpreta de la siguiente manera:

- **Un número negativo:** significa que el objeto actual (`this`) es "menor" que el objeto comparado.
- **Cero (0):** significa que ambos objetos son "iguales" en cuanto al criterio de orden.
- **Un número positivo:** significa que el objeto actual (`this`) es "mayor" que el objeto comparado.

Sobrescritura personalizada de `compareTo()`

Imagina que tenemos una clase `Persona` que queremos ordenar por la edad de las personas:

```

1  public class Persona implements Comparable<Persona> {
2      private String nombre;

```

```

3     private int edad;
4
5     // Constructores
6     ...
7
8     @Override
9     public int compareTo(Persona otraPersona) {
10        // Comparando por edad
11        if (this.edad < otraPersona.edad) {
12            return -1; // El objeto actual es menor
13        } else if (this.edad > otraPersona.edad) {
14            return 1; // El objeto actual es mayor
15        } else {
16            return 0; // Ambos son iguales en edad
17        }
18    }
19
20    // Getters para nombre y edad
21    ...
22 }

```

Explicación paso a paso:

1. **Implements Comparable:** Implementamos la interfaz `Comparable<Persona>`, que nos obliga a sobrescribir el método `compareTo()`.
2. **Comparación:** Dentro de `compareTo()`, comparamos la edad del objeto actual (`this.edad`) con la edad del objeto que estamos pasando como parámetro (`otraPersona.edad`).
3. **Resultado:**
 - Si el objeto actual es más joven (`this.edad < otraPersona.edad`), devolvemos -1, indicando que es "menor".
 - Si es más viejo (`this.edad > otraPersona.edad`), devolvemos 1, indicando que es "mayor".
 - Si tienen la misma edad (`this.edad == otraPersona.edad`), devolvemos 0, indicando que son "iguales" en este aspecto.

Este enfoque permite que los objetos de la clase `Persona` se ordenen automáticamente, por ejemplo, en un `ArrayList` usando `Collections.sort()`.

💡 Tip

📄 El método `compareTo()` define cómo se comparan dos objetos de la misma clase, devolviendo un valor que permite saber cuál debe ir antes o después. Sobrescribir este método es clave para poder ordenar los objetos de una forma lógica, ya sea por un atributo numérico, alfabético, o cualquier otro criterio.

💡 Tip

🤖 Si vamos a comparar únicamente números, es más fácil hacer lo siguiente:

```

1  @Override
2  public int compareTo(Persona otraPersona) {
3      return this.edad - otraPersona.edad;
4  }

```


Seguimos cumpliendo las reglas. La cantidad devuelta da exactamente igual, solo importa si es negativo, 0 ó positivo.

★ Sobrescritura rápida de `compareTo()`

Para implementar el método `compareTo()` de forma rápida en una clase, puedes aprovechar el método `toString()` para comparar las representaciones en cadena de los objetos.

Por ejemplo, si defines el método `toString()` en tu clase de manera que los atributos se concatenen en un orden específico (como nombre, edad), puedes utilizar `this.toString().compareTo(otraPersona.toString())` dentro de `compareTo()`. Esto permitirá que los objetos se ordenen según el criterio establecido en el `toString()`.

```
1  @Override
2  public int compareTo(Persona otraPersona) {
3      return this.toString().compareTo(otraPersona.toString());
4  }
5
6  @Override
7  public String toString() {
8      return "Persona{nombre='" + nombre + "', edad=" + edad + '}';
9      // Asegúrate de que el orden de los atributos sea el deseado
10 }
11
```

📌 Note

No es necesario comprobar las clases ni hacer casting porque no accedemos a los atributos de la clase y usamos el `toString()`, que lo tiene seguro porque lo tienen todos los objetos.

⚠ Caution

Si el objeto recibido es `null`, al hacer la llamada al `toString()` si que provocaremos una excepción de tipo `NullPointerException`. A veces está bien que dejemos que salten las excepciones porque si no, no nos daremos cuenta del error y será más difícil detectarlo.

📌 Important



Es crucial que los atributos en el `toString()` estén organizados de manera coherente para que la comparación sea lógica y cumpla con el objetivo de ordenación deseado. Si mostramos primero el nombre y después la edad, como las cadenas se comparan de forma lexicográfica, en primer lugar se ordenarán por nombre y en caso de igualdad, por edad.

⚠ Warning

Al comparar dos cadenas usando el método `compareTo()`, la comparación se realiza de forma **lexicográfica**, similar a cómo se ordenan las palabras en un diccionario. Esto significa que se compara carácter por carácter según su valor Unicode, y no se toman en cuenta los valores numéricos en su contexto.

Por ejemplo, al comparar las cadenas `"10"` y `"2"`, el resultado será que `"10"` es menor que `"2"` porque el primer carácter (`'1'`) de `"10"` tiene un valor Unicode menor que el carácter `'2'`. Esto puede llevar a resultados inesperados si se espera que se ordenen como números. Por lo tanto, al utilizar `toString()` para la comparación, es importante asegurarse de que los atributos sean ordenados adecuadamente en formato de cadena si se desea un orden numérico.

Tip

 Y con este truco, hemos usado los 5 tipos de avisos .
