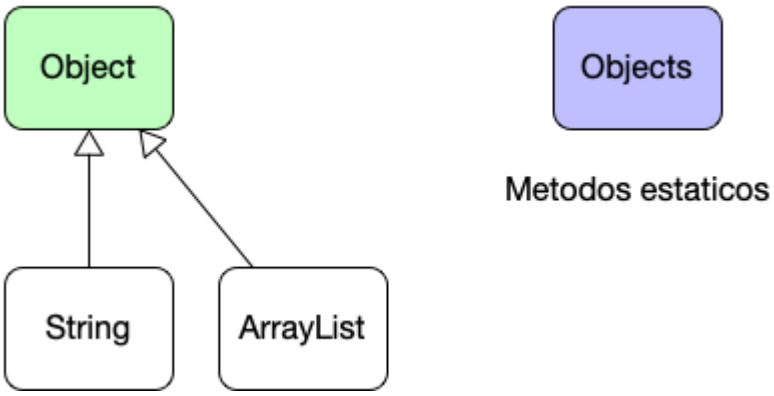
Tabla de Contenidos

- Validaciones más limpias con requireNonNull
- Comparaciones seguras con equals
- hashCode y toString sin complicaciones
- Trabajando con Streams y isNull
- Conclusión

Cuando trabajamos con Java, uno de los errores más comunes y temidos es el famoso NullPointerException. Seguro que en más de una ocasión te has encontrado con este problema en plena ejecución y has perdido tiempo tratando de localizar la causa. Para ayudarnos a escribir código más seguro y legible, desde Java 7 tenemos a nuestra disposición la clase java.util.Objects. Muchas personas la confunden con java.lang.Object pero son muy diferentes.



Esta clase no es más que un conjunto de métodos estáticos que actúan como una caja de herramientas. Su misión es sencilla: hacernos la vida más fácil cuando trabajamos con objetos, reduciendo el riesgo de errores y evitando escribir código repetitivo.

Validaciones más limpias con `requireNonNull`

Un uso muy habitual de Objects es en la validación de parámetros. Antes de Java 7 solíamos comprobar manualmente si una variable era null. Con Objects.requireNonNull todo se

simplifica:

```
this.nombre = Objects.requireNonNull(nombre, "El nombre no puede ser nulo");
```

De esta forma, si alguien intenta crear un objeto sin nombre, Java lanzará una excepción con un mensaje claro. Esto hace que nuestro código sea más robusto y fácil de mantener.

Comparaciones seguras con `equals`

Otro escenario común es la comparación de objetos. Usar equals directamente puede ser peligroso si una de las referencias es null. Con Objects.equals evitamos ese riesgo:

```
System.out.println(Objects.equals("hola", null)); // false
System.out.println(Objects.equals(null, null));   // true
```

Este método nos protege de excepciones y nos permite comparar de forma más segura y elegante.

`hashCode` y `toString` sin complicaciones

Cuando definimos clases de dominio, solemos sobrescribir equals, hashCode y toString. Aquí también Objects nos da la mano. Con Objects.hash podemos generar un hashCode consistente de forma sencilla:

```
@Override
public int hashCode() {
    return Objects.hash(nombre, edad);
}
```

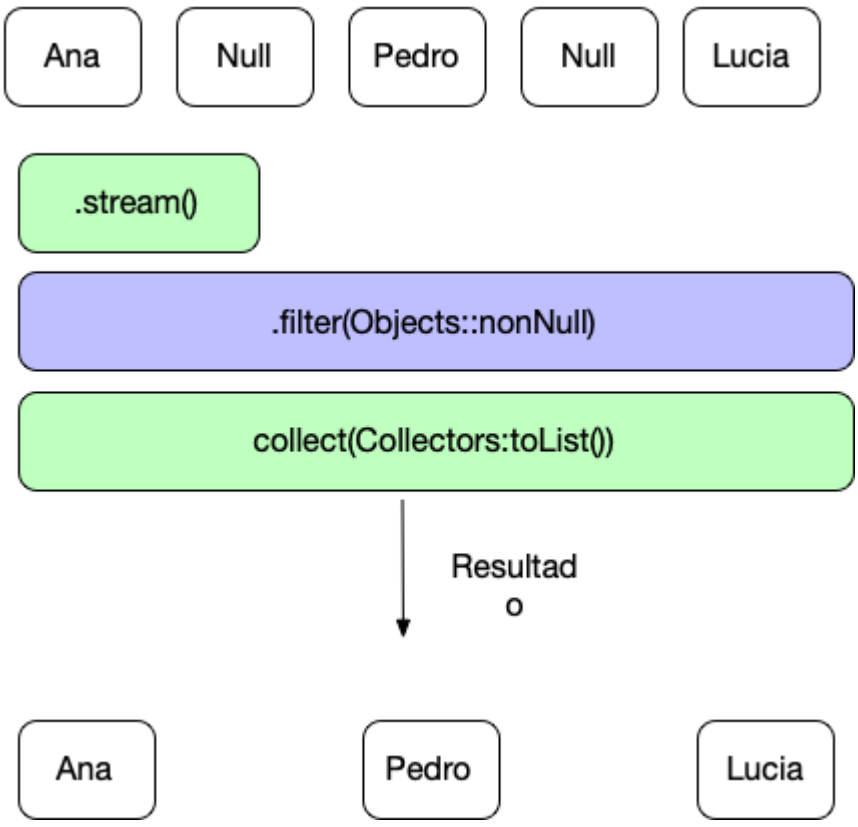
Y con Objects.toString obtenemos cadenas legibles sin preocuparnos de los nulos:

```
@Override
public String toString() {
```

```
    return "Persona{nombre=" + Objects.toString(nombre, "desconocido")
+
            ", edad=" + edad + "}";
}
```

Trabajando con Streams y `isNull`

Un ejemplo muy interesante aparece cuando usamos Streams. Supongamos que tenemos una lista con valores nulos y queremos filtrarlos. Con Objects.isNull y Objects.nonNull podemos hacerlo de manera muy limpia:




```
List<String> nombres = Arrays.asList("Ana", null, "Pedro", null,
"Lucía");

List<String> sinNulos = nombres.stream()
```

```
                                .filter(Objects::nonNull)
                                .collect(Collectors.toList());

System.out.println(sinNulos); // [Ana, Pedro, Lucía]
```

Esto hace que el código sea mucho más legible y expresivo que usar condiciones con `== null` o `!= null`.

Conclusión

La clase Objects es una de esas utilidades que a menudo pasan desapercibidas, pero que pueden marcar la diferencia en nuestro día a día como desarrolladores. Nos ayuda a escribir código más claro, seguro y conciso, evitando muchos de los problemas clásicos relacionados con los valores nulos.

La próxima vez que necesites validar parámetros, comparar objetos o limpiar una colección de nulos en un Stream, recuerda que Objects está ahí para echarte una mano.

- Generando un Java HashCode correcto
- java == vs equals comparando objetos y tipos básicos
- Java toString overriding y Eclipse
- Java Stream to List y simplificación
- Java override HashCode y curiosidades