ArgentinaPrograma
YoPrograma

Conceptos básicos del lenguaje Java

por Leonardo Blautzik, Federico Gasior y Lucas Videla

Julio / Diciembre 2021

Ministerio de
Desarrollo Productivo
Argentina

Motivación

- ¿Por qué queremos conocer mejor **la API de un lenguaje**?
- ¿Qué significa ser **buenos ciudadanos** del lenguaje Java?

Ventajas

- No **reinventaremos** la rueda

Ventajas

- No **reinventaremos** la rueda
- Podremos apoyarnos en **estructuras ya establecidas**

Ventajas

- No **reinventaremos** la rueda
- Podremos apoyarnos en **estructuras ya establecidas**
- Nuestras clases **se acoplarán mejor** a los modelos de objetos de la API de Java

Ventajas

- No **reinventaremos** la rueda
- Podremos apoyarnos en **estructuras ya establecidas**
- Nuestras clases **se acoplarán mejor** a los modelos de objetos de la API de Java
- Podremos **comunicarnos eficientemente** con colegas

final

- Es un modificador que implica la **imposibilidad de cambio** de una definición dada.
- Puede aplicarse a
  - Variables
  - Métodos
  - Clases

Variables `final`

Una vez asignado su valor, no puede cambiar

```
final int value = 10;
value = 11; // error de compilación
```

- Pueden inicializarse en línea, en un bloque `static` o en todos los constructores.

Métodos `final`

No pueden sobreescribirse

```
final void doStuff() { ... }
```

Clases `final`

No pueden extenderse (heredarse)

```
final class String { ... }
```

static

- Se define como **contrapartida a un contexto dinámico**. Si un contexto dinámico necesita de un objeto, un contexto estático no lo necesita.
- Puede aplicarse a:
  - Variables
  - Bloques
  - Métodos
  - Clases anidadas

Variables `static`

Se comparten entre todos los objetos que comparten clase.

Son variables globales, a efectos prácticos.

```
class Some {
  public static int value;
}
// ...
System.out.println(Some.value);
```

Bloques `static`

Sirve para inicializar variables `static`

```
class Some {
  public static int value;
  static {
    value = 10;
  }
}
```

Métodos `static`

Sirve para definir servicios y funciones provistas por una clase, sin involucrar a ningún objeto en particular.

Sólamente acceden a variables `static`. Un ejemplo es el `main`.

```
class Some {
  public static int doStuff() {...}
}
// ...
Some.doStuff();
```

Clases anidadas `static`

¡Por ahora no! :)

Organizando el código: los `package`

Para que las clases puedan encontrarse más fácilmente, evitar conflictos de nombre y garantizar el control de acceso, **se reunen clases afines en paquetes coherentes**.

¿Por qué usar `package`?

- Utilizar un mismo paquete comunica claramente que esas **clases están relacionadas**.

¿Por qué usar `package`?

- Utilizar un mismo paquete comunica claramente que esas **clases están relacionadas**.
- Podemos suponer que dentro del mismo paquete encontraremos **clases que cumplan tareas afines**.

¿Por qué usar package?

- Utilizar un mismo paquete comunica claramente que esas **clases están relacionadas**.
- Podemos suponer que dentro del mismo paquete encontraremos **clases que cumplan tareas afines**.
- Los nombres de clase no entrarán en conflicto con otros, ya que el paquete dará el **namespace** necesario.

¿Por qué usar `package`?

- Utilizar un mismo paquete comunica claramente que esas **clases están relacionadas**.
- Podemos suponer que dentro del mismo paquete encontraremos **clases que cumplan tareas afines**.
- Los nombres de clase no entrarán en conflicto con otros, ya que el paquete dará el **namespace** necesario.
- Se puede otorgar, mediante los modificadores necesarios, **acceso irrestricto** a métodos y atributos para clases que compartan paquete.

Annotations

Una anotación es una forma de metadata. Proporciona **información sobre un programa**, que no es parte del programa.
No tienen efecto directo sobre el código que anotan.

Para qué utilizar anotaciones

- Proporcionar **información al compilador**: ayudar a detectar errores, o suprimir advertencias

Para qué utilizar anotaciones

- Proporcionar **información al compilador**: ayudar a detectar errores, o suprimir advertencias
- **Procesamiento en tiempo de compilación** y de despliegue: algún software podría utilizar anotaciones para generar código dinámicamente

Para qué utilizar anotaciones

- Proporcionar **información al compilador**: ayudar a detectar errores, o suprimir advertencias
- **Procesamiento en tiempo de compilación** y de despliegue: algún software podría utilizar anotaciones para generar código dinámicamente
- **Procesamiento en tiempo de ejecución**: Algunas anotaciones pueden inspeccionarse durante la ejecución de un programa por medio de **reflection**.

Anotaciones en la práctica

Una anotación luce de la siguiente manera:

```
@SuppressWarnings(value = "unchecked")
void doStuff() { ... }
```

Un ejemplo que ya conocemos:

```
@Test
public void testSomething() { ... }
```

¿Dónde pueden utilizarse las anotaciones?

Se pueden aplicar en:

- Clases
- Atributos
- Métodos
- Otros...

La familia de `Number`

- Ya conocemos los tipos primitivos `int`, `double`, `boolean`, etc.

La familia de `Number`

- Ya conocemos los tipos primitivos `int`, `double`, `boolean`, etc.
- Existen equivalentes en forma de “clase”: `Integer`, `Double`, `Boolean`, y así.

La familia de `Number`

- Ya conocemos los tipos primitivos `int`, `double`, `boolean`, etc.
- Existen equivalentes en forma de “clase”: `Integer`, `Double`, `Boolean`, y así.
- Todos los numéricos pertenecen a la familia de `Number`.

La familia de `Number`

- Ya conocemos los tipos primitivos `int`, `double`, `boolean`, etc.
- Existen equivalentes en forma de “clase”: `Integer`, `Double`, `Boolean`, y así.
- Todos los numéricos pertenecen a la familia de `Number`.
- La utilidad es poder emplearlos en sitios donde sólo pueden almacenarse objetos y no primitivos (por ejemplo, listas)

La familia de `Number`

- Ya conocemos los tipos primitivos `int`, `double`, `boolean`, etc.
- Existen equivalentes en forma de “clase”: `Integer`, `Double`, `Boolean`, y así.
- Todos los numéricos pertenecen a la familia de `Number`.
- La utilidad es poder emplearlos en sitios donde sólo pueden almacenarse objetos y no primitivos (por ejemplo, listas)
- A partir de Java 1.5, existe el **autoboxing** y **unboxing**, para adaptar nuestros tipos primitivos a clases y viceversa cuando sea necesario.

Un miembro destacable: `Integer`

```
Integer.MAX_VALUE; // retorna el máximo entero
Integer.max(a, b); // retorna el máximo entre a y b
Integer.min(a, b); // retorna el mínimo entre a y b
```

Manejando cadenas de caracteres: `String`

Los `String` son **cadenas de caracteres**, y se utilizan ámpliamente en los lenguajes de programación.
En Java, los `String` son objetos. Por lo tanto tienen métodos que podremos utilizar.

Algunos métodos de `String`

```
String greeting = "¡Hola!";
greeting.length();              // 6
greeting.toLowerCase();         // ¡hola!
greeting.replaceAll("l", "j"); // ¡Hoja!
greeting.charAt(2);             // o
```

Palíndromos: para explorar los `String`

- Palabras
  - oro
  - reconocer
  - orejero
- Frases
  - Dábale arroz a la zorra el abad
  - Ana, la tacaña catalana

Partiendo String: split

```
String a = "una frase muy canchera";

a.split(" "); // ["una", "frase", "muy", "canchera"]
```

Uniendo varios `String`: `concat` y `+`

```
String a = "balon";
String b = "cesto";

a + b;         // baloncesto
a.concat(b);   // baloncesto
```

Convirtiendo de números a cadenas... y viceversa

```
String original = "1234";

int a = Integer.parseInt(original);
Integer b = Integer.valueOf(original);

String c = Integer.toString(a);
String d = b.toString();
```

La clase `Object`

`Object` es el **ancestro** implícito (o explícito) **de cualquier clase que programemos**.

Esto fue una **decisión de diseño** del lenguaje de programación. No es el único lenguaje en el que se ha tomado esta decisión.

Que sea ancestro de todas, implica que puede **definir un comportamiento base** común a todas, y vamos a explorarlo.

El método `toString()`

Todo objeto tiene una **representación en forma de cadena de caracteres**.
La misma se puede acceder mediante el método `toString()`.
Para nuestras clases, podemos definir el comportamiento de este método **sobreescribiéndolo**.

El método `clone()`

Este método servirá para obtener una réplica de un objeto dado. Para ello deberá:

- Implementar la interfaz `Cloneable`, para dar soporte explícito a la funcionalidad
- Llamar a `super.clone()` para obtener la referncia al objeto nuevo
- Poblar dicho objeto con los atributos del objeto actual

Clonado superficial o profundo

- Un clonado superficial implica replicar sólamente los **atributos inmediatos del objeto** a clonar.
- Un clonado profundo implica, en cambio, **replicar los atributos y todo aquel objeto que lo componga**.

Los métodos `equals` y `hashCode`

Ambos métodos **deben** definirse al mismo tiempo y teniendo en cuenta el mismo criterio.
Dicha definición conjunta suele llamarse contrato contrarrecíproco, y debe cumplirse para garantizar una buena integración con las clases provistas por el lenguaje.

equals

```
public boolean equals(Object o) { ... }
```

Debe cumplir con las siguientes premisas:

- Reflexividad

equals

```
public boolean equals(Object o) { ... }
```

Debe cumplir con las siguientes premisas:

- Reflexividad
- Simetría

equals

```
public boolean equals(Object o) { ... }
```

Debe cumplir con las siguientes premisas:

- Reflexividad
- Simetría
- Transitividad

equals

```
public boolean equals(Object o) { ... }
```

Debe cumplir con las siguientes premisas:

- Reflexividad
- Simetría
- Transitividad
- Consistencia

equals

```
public boolean equals(Object o) { ... }
```

Debe cumplir con las siguientes premisas:

- Reflexividad
- Simetría
- Transitividad
- Consistencia
- Falsedad ante `null` (nada es igual a `null`)

hashCode

```
public int hashCode() { ... }
```

Debe cumplir con:

- Consistencia a través del tiempo

hashCode

```
public int hashCode() { ... }
```

Debe cumplir con:

- Consistencia a través del tiempo
- Consistencia ante la igualdad

hashCode

```
public int hashCode() { ... }
```

Debe cumplir con:

- Consistencia a través del tiempo
- Consistencia ante la igualdad
- No necesariamente dos objetos distintos deben arrojar hashCode distintos (problema de dominio)

Comparando objetos: `Comparable`

¿Cuál es el orden natural de nuestros objetos? Para los tipos primitivos, es simple.

Podemos definir un criterio de comparación para aquellas clases que implementemos.

Deberemos implementar la interfaz `Comparable`.

El método `compareTo`

```
public int compareTo(Object o) { ... }
```

Arrojará un entero n, tal que:

- `n > 0` si el objeto actual **es mayor** que el parámetro
- `n < 0` si el objeto actual **es menor** que el parámetro
- `n = 0` si ambos objetos son iguales

Flexibilidad, por favor: `Comparator` al rescate

Cuando necesitamos más de un criterio de comparación, podemos utilizar comparadores. Para ello, debemos implementar la interfaz `Comparator` en una clase cuya responsabilidad es **saber comparar objetos mediante un criterio**.

Aplican las mismas reglas que para el método `compareTo`.

Los `enum` y su importancia

- Los `enum` son, salvando la distancia, **clases de las cuales sus objetos ya están predefinidos**.

Los `enum` y su importancia

- Los `enum` son, salvando la distancia, **clases de las cuales sus objetos ya están predefinidos**.
- Cuando conocemos todos los posibles valores en tiempo de compilación, utilizamos `enum`.

Los `enum` y su importancia

- Los `enum` son, salvando la distancia, **clases de las cuales sus objetos ya están predefinidos**.
- Cuando conocemos todos los posibles valores en tiempo de compilación, utilizamos `enum`.
- Los `enum` permiten las **bondades de los objetos**, mezcladas con la **conveniencia de las constantes**.

Recapitulando...

- Conceptos generales
  - `final`, `static`
  - `package`
  - Annotations
- `Number` / `String`
- `Object`
  - `toString`
  - `clone`
  - `equals` / `hashCode`
- `Comparable` / `Comparator`
- `enum`

Para profundizar

- Learning the Java Language (The Java™ Tutorials). Accedido 23 de Julio de 2021, desde https://docs.oracle.com/javase/tutorial/java/TOC.html

¡Muchas Gracias!

continuará...

Ministerio de
Desarrollo Productivo
Argentina