Effective Java

Creating and Destroying Objects

Este capítulo trata de la creación y destrucción de objetos: cuándo y cómo crearlos, cuándo y cómo evitar su creación, cómo asegurar que se destruyan a tiempo y cómo gestionar cualquier acción de limpieza que deba preceder a su destrucción.

Item 1: Consider static factory methods instead of constructors

La forma tradicional de que una clase permita a un cliente obtener una instancia es proporcionar un constructor público. Pero hay otra técnica y es proveer un método público *'static factory'* que es simplemente un **método estático que retorna una instancia** de la clase.

```
// Retorna una instancia de Boolean usando el parámetro de tipo boolean
public static Boolean valueOf(boolean b) {
    return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Hay que tener en cuenta que *'static factory method'* no es lo mismo que el patrón ***'Factory Method'*** de los patrones de diseño *"Design Patterns: Elements of Reusable Object-Oriented Software"*.

No es incompatible que una clase suministre *'static factory methods'* además de constructores públicos.

El uso de estos métodos tiene ventajas:

- **Una ventaja de los *'static factory methods'* es que, a diferencia de los constructores, tienen nombres**. Podemos elegir nombres que sean mucho más descriptivos que los constructores.
- **Una segunda ventaja es que, a diferencia de los constructores, no tienen que crear un nuevo objeto cada vez que se les invoca.** Esto permite clases inmutables que retornen instancias ya creadas, mejorando el rendimiento ya que podemos evitar la creación de nuevos objetos.
- **Una tercera ventaja es que, a diferencia de los constructores, estos métodos pueden devolver un objeto de cualquier subtipo de su tipo de devolución.** Esto da una gran flexibilidad para elegir la clase del objeto devuelto.
- **Una cuarta ventaja de las fábricas estáticas es que la clase del objeto devuelto puede variar de una llamada a otra en función de los parámetros de entrada.** Se permite cualquier subtipo del tipo de retorno declarado. La clase del objeto devuelto también puede variar de una liberación a otra.
- **Una quinta ventaja de las fábricas estáticas es que la clase del objeto devuelto no necesita existir cuando se escribe la clase que contiene el método.**

Como inconvenientes destacar:

- **La principal limitación de proporcionar sólo métodos estáticos de fábrica es que las clases sin constructores públicos o protegidos no pueden ser heredadas.**
- **Otra limitación es que no es fácil detectar estas factorías en la documentación de la clase.** Esto es debido a como funciona la herramienta de Javadoc. Los constructores aparecen en un lugar destacado a diferencia de los métodos. Normalmente, estos métodos suelen seguir ciertas convenciones:
    - **from**: un método *'type-conversion'* que toma un parámetro y retorna la correspondiente instancia de ese tipo:

```
Date d = Date.from(instant);
```

- **of**: un método de agregación que toma múltiples parámetros y devuelve una instancia de ese tipo que los incorpora:

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

- **valueOf**: una forma más descriptiva de *'from'* y *'of'*:

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

- **instance** or **getInstance**: retorna una instancia que se describe por sus parámetros (si los hay) pero que no puede decirse que tenga el mismo valor:

```
StackWalker luke = StackWalker.getInstance(options);
```

- **create** or **newInstance**: como el anterior salvo que esta vez de garantiza que en cada llamada se devuelve una nueva instancia

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

- **getType**: como **getInstance**, pero se usa si el método de fábrica está en una clase diferente. *'Type'* es el tipo de objeto devuelto por el método de fábrica:

```
FileStore fs = Files.getFileStore(path);
```

- **newType** como **newInstance**, pero se usa si el método de fábrica está en una clase diferente. *'Type'* es el tipo de objeto devuelto por el método de fábrica:

```
BufferedReader br = Files.newBufferedReader(path);
```

- **type** una alternativa concisa a **getType** y **newType**:

```
List<Complaint> litany = Collections.list(legacyLitany);
```

Item 2: Consider a builder when faced with many constructor parameters

Las factorías estáticas y los constructores comparten una limitación: no se adaptan bien a un gran número de parámetros opcionales.

Tradicionalmente los programadores han usado el patrón *'telescoping constructor'* en el cual se provee a la clase de un constructor con los parámetros requeridos, otro constructor con uno de los parámetros opcionales, otro con dos y así sucesivamente hasta completar la lista y tener un constructor con todos los opcionales. De esta forma cuando se desea crear

una instancia, se utiliza el constructor con la lista de parámetros más corta que contiene todos los parámetros que se desean configurar. Los parámetros que no se utilizan se suele pasar como 0, 'null', etc..

```
public class NutritionFacts {
  private final int servingSize;  // (mL) required
  private final int servings;     // (per container) required
  private final int calories;     // (per serving) optional
  private final int fat;          // (g/serving) optional
  private final int sodium;       // (mg/serving) optional
  private final int carbohydrate; // (g/serving) optional

  public NutritionFacts(int servingSize, int servings) {
    this(servingSize, servings, 0);
  }

  public NutritionFacts(int servingSize, int servings, int calories) {
    this(servingSize, servings, calories, 0);
  }

  public NutritionFacts(int servingSize, int servings, int calories, int fat) {
    this(servingSize, servings, calories, fat, 0);
  }

  public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium) {
    this(servingSize, servings, calories, fat, sodium, 0);
  }

  public NutritionFacts(int servingSize, int servings, int calories, int fat, int sodium, int carbohydrate) {
    this.servingSize = servingSize;
    this.servings = servings;
    this.calories = calories;
    this.fat = fat;
    this.sodium = sodium;
    this.carbohydrate = carbohydrate;
  }
}
```

En resumen, el patrón *'telescoping constructor'* funciona, pero es difícil escribir código cliente cuando hay muchos parámetros, y es más difícil de leer. Además, es propenso a errores ya que cuanto más extensa es la lista de parámetros mayores probabilidades de equivocarse en el orden de los mismos al invocar un constructor. Si los parámetros son del mismo tipo, el compilador no mostrará ningún error.

Otro patrón que permite trabajar con muchos parámetros opcionales en un constructor es el patrón *'JavaBean'*. En este patrón se invoca un constructor sin parámetros para crear un objeto y luego se invocan los metódos `setters` de cada parámetro tanto requerido como opcional que sea necesario para construir correctamente el objeto:

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Este patrón es más fácil de leer y mantener pero tiene el inconveniente de que debido a que la construcción se divide en múltiples llamadas, un *'JavaBean'* puede estar en un estado inconsistente a lo largo de su construcción. La clase no tiene la opción de hacer cumplir la consistencia simplemente comprobando la validez de los parámetros del constructor. Intentar usar un objeto cuando está en un estado inconsistente puede causar fallos que están lejos del código que contiene el fallo y por lo tanto son difíciles de depurar.

Afortunadamente, existe una tercera alternativa que combina la seguridad *'telescoping constructor'* con la legibilidad del patrón *'JavaBeans'*. Es una forma del patrón ***'Builder'*** incluido en *"Design Patterns: Elements of Reusable Object-Oriented Software"*.

En lugar de hacer el objeto deseado directamente, el cliente llama a un constructor (o fábrica estática) con todos los parámetros requeridos y consigue un objeto **'Builder'**. Luego el cliente llama a los métodos similares a los `setters` en el objeto constructor para establecer cada parámetro opcional de interés. Finalmente, el cliente llama a un método `build()` sin parámetros para generar el objeto, que es típicamente inmutable.

```
// Builder Pattern
public class NutritionFacts {
  private final int servingSize;
  private final int servings;
  private final int calories;
  private final int fat;
  private final int sodium;
  private final int carbohydrate;

  public static class Builder {
    // Required parameters
    private final int servingSize;
    private final int servings;
    // Optional parameters - initialized to default values
    private int calories = 0;
    private int fat = 0;
    private int sodium = 0;
    private int carbohydrate = 0;

    public Builder(int servingSize, int servings) {
      this.servingSize = servingSize;
      this.servings = servings;
    }

    public Builder calories(int val) {
      calories = val;
      return this;
    }

    public Builder fat(int val) {
      fat = val;
      return this;
    }

    public Builder sodium(int val) {
      sodium = val;
      return this;
    }

    public Builder carbohydrate(int val) {
      carbohydrate = val;
      return this;
    }

    public NutritionFacts build() {
      return new NutritionFacts(this);
    }
  }

  private NutritionFacts(Builder builder) {
    servingSize = builder.servingSize;
    servings = builder.servings;
    calories = builder.calories;
    fat = builder.fat;
    sodium = builder.sodium;
    carbohydrate = builder.carbohydrate;
  }
}
```

Este código de cliente es fácil de escribir y, lo que es más importante, fácil de leer. La clase es inmutable, y todos los valores por defecto de los parámetros están en un solo lugar. Los métodos `set` del **'Builder'** devuelven al constructor mismo (con `return`

`this`) para que las invocaciones puedan ser encadenadas, resultando en una API fluida. Para detectar parámetros no válidos lo antes posible, podemos verificar la validez de los parámetros en el constructor y los métodos del constructor:

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).calories(100).sodium(35).carbohydrate(27).build();
```

El patrón ***'Builder'*** simula los parámetros opcionales con nombre que se encuentran en Python, Kotlin o Scala.

Item 3: Enforce the singleton property with a private constructor or an enum type

Una clase *singleton* es simplemente una clase que se instancia exactamente una vez. Los *'singletons'* normalmente representan un objeto sin estado, como una función o un componente del sistema que es intrínsecamente único. Hacer que una clase sea un *'singleton'* puede dificultar la prueba de sus clientes porque es imposible sustituir una implementación simulada por un *'singleton'* a menos que implemente una interfaz que sirva como su tipo.

Hay dos formas comunes de implementar *'singletons'*. Ambos se basan en mantener el constructor privado y exportar un miembro estático público para proporcionar acceso a la única instancia.

En primer lugar, hacer que la variable miembro sea un campo final:

```
// Singleton with public final field
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public void leaveTheBuilding() { ... }
}
```

El constructor privado es llamado una única vez para inicializar el campo públic, estático y final `Elvis.instance`. En teoría sólo habrá un único `Elvis` aunque mediante reflexión, un cliente con suficientes privilegios podría invocar al método privado haciéndolo accesible. Para evitar esto, hay que modificar el constructor para que lance una excepción si se intenta crear una segunda instancia.

La principal ventaja del enfoque de campo público es que la API deja claro que la clase es una clase *'singleton'*: el campo estático público es final, por lo que siempre contendrá la misma referencia de objeto. La segunda ventaja es que es más simple.

Una segunda forma es hacer que el miembro público sea un método *'static factory'*:

```
// Singleton with static factory
public class Elvis {
    private static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }
    public static Elvis getInstance() { return INSTANCE; }
    public void leaveTheBuilding() { ... }
}
```

Todas las llamadas a `Elvis.getInstance()` devuelven la misma referencia de objeto, y nunca se creará ninguna otra instancia de Elvis (con el mismo problema mencionado anteriormente).

Una de las ventajas de este enfoque es que brinda la flexibilidad de cambiar de opinión sobre si la clase es un singleton sin cambiar su API. El método *'static factory'* devuelve la única instancia, pero podría modificarse para devolver, por ejemplo, una instancia separada para cada hilo que lo invoque. Una segunda ventaja es que puede escribir una fábrica de singleton genérica si su aplicación lo requiere.

A menos que una de estas ventajas sea relevante, **el primer enfoque de campo público es preferible al segundo enfoque**.

Una tercera forma de implementar una clase *'singleton'* es declarar una enumeración de un solo elemento. Ese enfoque es parecido al enfoque de campo público sin el inconveniente del problema de la reflexión. Es un enfoque más conciso y directo pero es también un enfoque poco natural. Un tipo de enumeración de un solo elemento es a menudo la mejor manera de implementar un *'singleton'*. Tenga en cuenta que no puede usar este enfoque si su *'singleton'* debe extender una superclase que no sea `Enum`.

Item 4: Enforce noninstantiability with a private constructor

Ocasionalmente, querrá escribir una clase que sea solo una agrupación de métodos estáticos y campos estáticos. Estas clases han adquirido una mala reputación debido a que algunas personas abusan de ellas para evitar pensar en términos de objetos, pero tienen usos válidos.

Se pueden utilizar para agrupar métodos relacionados en valores primitivos o arrays como en `java.lang.Math` o `java.util.Arrays`. También se pueden usar para agrupar métodos estáticos, incluidas factorías estáticas, para objetos que implementan alguna interfaz como en `java.util.Collections`. Por último, estas clases se pueden usar para agrupar métodos en una clase final, ya que no se pueden colocar en una subclase.

Tales clases de utilidad no fueron diseñadas para ser instanciadas: una instancia no tendría sentido. Sin embargo, en ausencia de constructores explícitos, el compilador proporciona un constructor público, sin parámetros y predeterminado. Para un usuario, este constructor es indistinguible de cualquier otro.

Intentar imponer la no instanciabilidad haciendo que la clase sea abstracta no funciona. Se podría instanciar una subclase. Además, induce a error al usuario al pensar que la clase fue diseñada para herencia.

Existe, sin embargo, un *'idiom'* simple para garantizar la no instanciación. Un constructor predeterminado se genera solo si una clase no contiene constructores explícitos, por lo que se puede hacer que una clase no sea instanciable al incluir un constructor privado:

```
// Noninstantiable utility class
public class UtilityClass {
    // Suppress default constructor for noninstantiability
    private UtilityClass() {
        throw new AssertionError();
    }
    // ....
}
```

Debido a que el constructor explícito es privado, es inaccesible fuera de la clase. El `AssertionError` no se requiere estrictamente, pero proporciona un mecanismo seguro en caso de que el constructor sea invocado accidentalmente desde dentro de la clase. Garantiza que la clase nunca será instanciada bajo ninguna circunstancia.

Este *'idiom'* es ligeramente contrario a la intuición porque el constructor se proporciona expresamente para que no se pueda invocar. Por lo tanto, es aconsejable incluir un comentario, como se mostró en el ejemplo.

Como efecto secundario, este *'idiom'* también evita que la clase sea heredada. Todos los constructores deben invocar un constructor de superclase, explícita o implícitamente, y una subclase no tendría un constructor de superclase accesible para invocar.

Item 5: Prefer dependency injection to hardwiring resources

Muchas clases dependen de recursos subyacentes. Por ejemplo, un corrector ortográfico depende de un diccionario. No es raro ver estas clases implementadas como clases de utilidad estáticas (Item 4):

```
// Inappropriate use of static utility - inflexible & untestable!
public class SpellChecker {
    private static final Lexicon dictionary = ...;
    private SpellChecker() {} // Noninstantiable
    public static boolean isValid(String word) { ... }
    public static List<String> suggestions(String typo) { ... }
}
```

Del mismo modo, no es raro verlos implementados como *singletons* (Item 3):

```
// Inappropriate use of singleton - inflexible & untestable!
public class SpellChecker {
    private final Lexicon dictionary = ...;
    private SpellChecker(...) {}
    public static INSTANCE = new SpellChecker(...);
    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

Ninguno de estos enfoques es satisfactorio porque suponen que sólo será útil utilizar un único diccionario. La realidad es que cada idioma tendrá su propio diccionario. Además, a efectos de pruebas puede ser necesario el uso de un diccionario especial.

Puede intentar que `SpellChecker` admita varios diccionarios haciendo que el campo `dictionary` no sea final y agregando un método para cambiar el diccionario en un corrector ortográfico existente, pero esto sería incómodo, propenso a errores e inviable en una configuración concurrente. **Las clases de utilidad estática y los singletons son inapropiados para las clases cuyo comportamiento está parametrizado por un recurso subyacente**.

Lo que se requiere es la capacidad de admitir varias instancias de la clase (en nuestro ejemplo, `SpellChecker`), cada una de las cuales utilice l recurso deseado por el cliente (en nuestro ejemplo, el diccionario). Un patrón simple que satisface este requisito es **pasar el recurso al constructor al crear una nueva instancia**. Esta es una forma de inyección de dependencia: el diccionario es una dependencia del corrector ortográfico y se inyecta en el corrector ortográfico cuando se crea:

```
// Dependency injection provides flexibility and testability
public class SpellChecker {
    private final Lexicon dictionary;
    public SpellChecker(Lexicon dictionary) {
        this.dictionary = Objects.requireNonNull(dictionary);
    }
    public boolean isValid(String word) { ... }
    public List<String> suggestions(String typo) { ... }
}
```

En el ejemplo la clase `SpellChecker` sólo tiene un recurso pero la inyección de dependencias funciona con un número arbitrario de recursos. La inyección de dependencias es igualmente aplicable a constructores, factorías estáticas (Item 1) y *builders* (Item 2).

Aunque la inyección de dependencias mejora en gran medida la flexibilidad y la capacidad de prueba, puede saturar grandes proyectos, que generalmente contienen miles de dependencias. Este desorden puede eliminarse utilizando un framework de inyección de dependencias como Dagger, Guice o Spring.

En resumen, no utilice una clase de utilidad estática o un *singleton* para implementar una clase que dependa de uno o más recursos subyacentes cuyo comportamiento afecte al de la clase, y no haga que la clase cree estos recursos directamente. En cambio, pase los recursos, o las factorías para crearlos, al constructor (o fábrica estática o *builder*). Esta práctica, conocida como inyección de dependencia, mejorará en gran medida la flexibilidad, la reutilización y la capacidad de prueba de una clase.

Item 6: Avoid creating unnecessary objects

(todo)