Mastering 'Effective Java'

> ⚠ DOCUMENTO EN DESARROLLO ⚠

Introducción

"Effective Java" es un influyente libro escrito por **Joshua Bloch**, un reconocido ingeniero de software que ha desempeñado un papel fundamental en el desarrollo de la plataforma Java. Publicado por primera vez en **2001** y posteriormente actualizado para reflejar las versiones más recientes del lenguaje, "Effective Java" se ha convertido en un recurso invaluable para desarrolladores Java de todos los niveles de experiencia.

Este libro proporciona una serie de **prácticas recomendadas** y **patrones de diseño** que ayudan a los programadores a escribir código Java más robusto, eficiente y fácil de mantener. A través de su enfoque práctico y con ejemplos claros, Bloch aborda diversas áreas, desde el manejo de objetos y la concurrencia hasta la gestión de excepciones y la creación de clases y métodos.

"Effective Java" no solo se centra en la sintaxis del lenguaje, sino que también aborda cuestiones más profundas relacionadas con la calidad del código y la toma de decisiones de diseño. Este libro se ha convertido en un recurso esencial para aquellos que buscan mejorar sus habilidades de programación en Java y construir software más eficiente y confiable.

> Introducción generada por ChatGPT

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 estático 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 este concepto de ***'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 estáticos como factoría tiene ventajas:

- Una ventaja es que, a diferencia de los constructores, estos **métodos 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 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 o ***'Fluent API'***. 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 otros lenguajes como 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 objetos *'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 el 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 *'static factory'* 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

A menudo conviene reutilizar un mismo objeto en lugar de crear un nuevo objeto funcionalmente equivalente cada vez que se necesita. La reutilización puede ser más rápida y elegante. Un objeto siempre puede reutilizarse si es inmutable (Item 17).

Un ejemplo extremo sería la creación de un `String` utilizando el constructor:

```
String s = new String("bikini"); // DON'T DO THIS!
```

La sentencia crea una nueva instancia de `String` cada vez que se ejecuta, y ninguna de esas creaciones de objetos es necesaria. El argumento del constructor es en sí mismo una instancia de `String`, funcionalmente idéntica a todos los objetos creados por el constructor. Si este uso se produce en un bucle o en un método frecuentemente invocado se crearán millones de instancias de String innecesariamente.

La forma correcta y más simple sería:

```
String s = "bikini";
```

Esta versión utiliza una única instancia de `String`, en lugar de crear una nueva cada vez que es ejecutada como en el caso anterior. Además, se garantiza que el objeto será reutilizado por cualquier otro código que se ejecute en la misma máquina virtual y que por casualidad contenga la misma cadena literal.

A menudo se puede evitar la creación innecesaria de objetos utilizando métodos 'factoría' estáticos o ***'static factory methods'*** (Item 1) preferentemente a utilizar constructores en clases inmutables que proveen de ambos.

Por ejemplo, el método de fábrica estático `Boolean.valueOf(String)` es preferible al constructor `Boolean(String)`, el cual fue marcado como *'deprecated'* en Java 9. El constructor debe crear un nuevo objeto cada vez que se llama, mientras que el método

de fábrica estático nunca está obligado a hacerlo y no lo hará en la práctica. Además de reutilizar objetos inmutables, también se puede reutilizar objetos mutables si se tiene conocimiento de que no se modificarán.

Algunas creaciones de objetos son mucho más caras que otras. Si va a necesitar un "objeto caro" de este tipo repetidamente, puede ser recomendable almacenarlo en caché para reutilizarlo. Desafortunadamente, no siempre es obvio cuándo se crea un objeto de este tipo.

Supongamos un método que determine cuando una cadena es un número romano válido:

```
// Performance can be greatly improved!
static boolean isRomanNumeral(String s) {
  return s.matches("^(?=.)M*(C[MD]|D?C{0,3})(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

El problema con esta implementación es que se basa en el método `String.matches(...)`. Si bien este método es la forma más sencilla de comprobar si una cadena coincide con una expresión regular, **no es adecuado para su uso repetido en situaciones críticas de rendimiento**. El problema es que crea internamente una instancia de *'Pattern'* para la expresión regular y la usa solo una vez, después de lo cual se vuelve elegible para la recolección de basura. Crear una instancia de *'Pattern'* es "caro" porque requiere compilar la expresión regular en una máquina finita de estados.

Para mejorar el rendimiento, compile explícitamente la expresión regular en una instancia de *'Pattern'* (que es inmutable) como parte de la inicialización de la clase, guárdela en caché y reutilice la misma instancia para cada invocación del método `isRomanNumeral(String s)`:

```
// Reusing expensive object for improved performance
public class RomanNumerals {
  private static final Pattern ROMAN = Pattern.compile("^(?=.)M*(C[MD]|D?C{0,3})"
      + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

  static boolean isRomanNumeral(String s) {
    return ROMAN.matcher(s).matches();
  }
}
```

Esta versión mejorada del método provee una ganancia significativa a nivel de eficiencia y rendimiento si se invoca repetidamente. Y no solamente rendimiento, si no que también gana en claridad. Crear un campo final estático para la instancia de *'Pattern'* que de otro modo sería invisible permite darle un nombre, que es mucho más legible que la propia expresión regular.

Por otro lado, si la clase que contiene la versión mejorada del método `isRomanNumeral(String s)` se inicializa pero el método nunca se invoca, el campo `ROMAN` con el patrón compilado se inicializará innecesariamente. Sería posible eliminar esta inicialización inicializando de forma diferida el campo (Item 83) la primera vez que se invoca el método `isRomanNumeral(String s)`, pero esto no se recomienda. Como suele ser el caso con la inicialización diferida, complicaría la implementación sin una mejora mensurable del rendimiento (Item 67).

Item 7: Eliminate obsolete object references

Si has cambiado de un lenguaje con gestión manual de memoria, como C o C++, a un lenguaje con recolección de basura, como Java, tu trabajo como programador se ha vuelto mucho más sencillo gracias a que los objetos se recuperan automáticamente cuando ya no se necesitan.

Al principio, esto puede parecer casi mágico. Sin embargo, esto puede llevar fácilmente a la errónea impresión de que no es necesario preocuparse por la gestión de memoria.

En el ejemplo tenemos una implementación de una **Pila**:

```
public class Stack {
  private Object[] elements;
  private int size = 0;
  private static final int DEFAULT_INITIAL_CAPACITY = 16;

  public Stack() {
    elements = new Object[DEFAULT_INITIAL_CAPACITY];
  }

  public void push(Object e) {
    ensureCapacity();
    elements[size++] = e;
  }

  public Object pop() {
    if (size == 0) {
      throw new EmptyStackException();
    }
    return elements[--size];
  }

  private void ensureCapacity() {
    if (elements.length == size) {
      elements = Arrays.copyOf(elements, 2 * size + 1);
    }
  }
}
```

No hay nada aparentemente incorrecto en este programa. Se podría someter a pruebas exhaustivas, y pasaría todas con éxito, pero hay un problema oculto. En términos generales, el programa tiene una **fuga de memoria** (***"memory leak"***), que puede manifestarse silenciosamente como una reducción del rendimiento debido a una mayor actividad del recolector de basura o un aumento en el consumo de memoria. En casos extremos, este tipo de fugas pueden provocar paginación en disco e incluso fallos del programa con un `OutOfMemoryError`, aunque estos fallos son relativamente raros.

Entonces, ¿dónde está la fuga de memoria? Si una pila (*"stack"*) crece y luego se reduce, los objetos que fueron eliminados (*"popped off"*) de la pila no serán recolectados por el recolector de basura, incluso si el programa que usa la pila ya no tiene referencias a ellos. Esto ocurre porque la pila mantiene referencias obsoletas a estos objetos.

Una **referencia obsoleta** es simplemente una referencia que nunca volverá a ser utilizada. En este caso, cualquier referencia fuera de la "porción activa" del arreglo de elementos es obsoleta. La porción activa está compuesta por los elementos cuyo índice es menor que `size`.

Las fugas de memoria en lenguajes con recolección de basura (más propiamente conocidas como **retenciones involuntarias de objetos**) son difíciles de detectar. Si una referencia a un objeto se mantiene accidentalmente, **no solo ese objeto queda excluido de la recolección de basura**, sino también todos los objetos referenciados por él, y así sucesivamente. Incluso si solo se retienen unas pocas referencias de manera involuntaria, **pueden impedir la recolección de una gran cantidad de objetos**, afectando significativamente el rendimiento.

La solución a este tipo de problema es sencilla: **asignar `null` a las referencias una vez que se vuelven obsoletas**. En el caso de nuestra clase `Stack`, la referencia a un elemento **se vuelve obsoleta tan pronto como se extrae de la pila**. La versión corregida del método `pop` se ve así:

```
public Object pop() {
  if (size == 0) {
    throw new EmptyStackException();
  }
  Object result = elements[--size];
  elements[size] = null; // Eliminate obsolete reference
  return result;
}
```

Un beneficio adicional de asignar `null` a las referencias obsoletas es que, si luego se intentan utilizar por error, el programa **fallará de inmediato con una** `NullPointerException`, en lugar de comportarse de manera incorrecta de forma silenciosa.

> 👉 **Siempre es mejor detectar los errores de programación lo antes posible**.

Cuando los programadores se encuentran por primera vez con este problema, pueden reaccionar de forma exagerada, asignando `null` a cada referencia de objeto en cuanto dejan de usarla. Esto no solo es innecesario, sino también poco recomendable, ya que **ensucia el código** sin aportar beneficios reales. **Asignar** `null` **a referencias debería ser la excepción, no la norma**.

La mejor forma de eliminar una referencia obsoleta es permitir que la variable que la contiene **salga naturalmente de su ámbito** (*"scope"*). Esto sucede automáticamente si se define **cada variable en el ámbito más reducido posible**.

Entonces, ¿cuándo se debería asignar `null` a una referencia? ¿Qué aspecto de la clase `Stack` la hace susceptible a fugas de memoria? En pocas palabras, **gestiona su propia memoria**.

El pool de almacenamiento está compuesto por los elementos del array `elements` (es decir, las celdas que contienen referencias a objetos, no los objetos en sí). Los elementos en la **porción activa** del array (como se definió anteriormente) están en uso, mientras que el resto están libres.

El **recolector de basura no tiene forma de saber esto**; para él, todas las referencias de `elements` son igualmente válidas. Solo el programador sabe que la porción inactiva del array es irrelevante. Para informar esto al recolector de basura, el programador debe **asignar** `null` **manualmente a los elementos del array en cuanto pasen a formar parte de la porción inactiva**.

En términos generales, **si una clase gestiona su propia memoria, el programador debe estar atento a posibles fugas de memoria**. Cada vez que se libere un elemento, cualquier referencia a objetos contenida en él debe ser asignada a `null`.

Otra fuente común de fugas de memoria son los **cachés**. Una vez que se agrega una referencia a un objeto en un caché, es fácil olvidarse de que está ahí y **dejarla almacenada mucho después de que se haya vuelto irrelevante**.

Existen varias soluciones para este problema. En el caso de una caché en el que una entrada **debe permanecer mientras haya referencias a su clave fuera del caché**, se puede representarlo con un `WeakHashMap`. En este caso, las entradas **se eliminarán automáticamente cuando se vuelvan obsoletas**. Sin embargo, un `WeakHashMap` solo es útil si la duración de una entrada en caché está determinada por referencias externas a la clave, y no al valor.

Más comúnmente, la vida útil de una entrada en caché **no está tan claramente definida**, y su importancia disminuye con el tiempo. En estos casos, el caché **debe limpiarse periódicamente** para eliminar las entradas en desuso. Esto se puede hacer de dos maneras:

- Mediante un hilo en segundo plano, como un `ScheduledThreadPoolExecutor`.
- Como un efecto colateral al agregar nuevas entradas**. La clase `LinkedHashMap` facilita este enfoque con su método `removeEldestEntry`.

Una tercera fuente común de fugas de memoria son los ***listeners* y otros *callbacks***. En una API en la que los clientes **registran *callbacks* pero no los desregistran explícitamente**, estos se acumularán a menos que se tomen medidas para evitarlos. Una forma de asegurarse de que los *callbacks* sean recolectados por el recolector de basura a tiempo es **almacenarlos solo como referencias débiles**, por ejemplo, usándolos como claves en un `WeakHashMap`.

Dado que las fugas de memoria no suelen manifestarse como fallos evidentes, pueden permanecer en un sistema durante años sin ser detectadas. Normalmente, solo se descubren mediante una **inspección minuciosa del código** o con la ayuda de una herramienta de depuración conocida como ***"heap profiler"***.

Existen herramientas para el análisis de memoria como Eclipse Memory Analyzer (MAT) de la Eclipse Foundation o herramientas propias de la JDK como Java VisualVM (sin embargo, a partir de Java 9, las distribuciones JDK ya no se envían

con Java VisualVM), **JConsole** o Java Flight Recorder (JFR).

Item 8: Avoid finalizers and cleaners

Los ***"finalizers"*** son impredecibles, a menudo peligrosos y, en general, innecesarios. Su uso puede provocar comportamientos erráticos, bajo rendimiento y problemas de portabilidad. Existen algunos casos en los que su uso es válido, pero como regla general, se deberían evitar.

A partir de Java 9, los *"finalizers"* han sido desaprobados (*deprecated*), aunque todavía se usan en algunas bibliotecas de Java. Su reemplazo en Java 9 son los ***"cleaners"***. Los *"cleaners"* son menos peligrosos que los *"finalizers"*, pero siguen siendo impredecibles, lentos y, en general, innecesarios.

Los programadores de C++ deben tener cuidado de no considerar los *"finalizers"* o *"cleaners"* como el equivalente en Java de los destructores de C++. En C++, los destructores son el mecanismo estándar para liberar los recursos asociados a un objeto, funcionando como complemento necesario de los constructores. En Java, en cambio, el recolector de basura se encarga de liberar la memoria automáticamente cuando un objeto se vuelve inaccesible, sin necesidad de que el programador haga nada especial.

Los destructores en C++ también se utilizan para liberar otros recursos no relacionados con la memoria. En Java, este propósito se logra mediante bloques `try-with-resources` o `try-finally`.

Uno de los grandes inconvenientes de los *"finalizers"* y *"cleaners"* es que no hay garantía de que se ejecuten de inmediato. Puede pasar un tiempo arbitrario desde que un objeto se vuelve inaccesible hasta que el método se ejecute. Esto significa que **nunca se debe realizar operaciones críticas en tiempo dentro de un *"finalizer"* o *"cleaner"***. Por ejemplo, depender de estos métodos para cerrar archivos es un grave error, ya que los descriptores de archivos son un recurso limitado. Si el sistema tarda demasiado en ejecutar los *"finalizers"* o *"cleaners"*, un programa podría fallar por no poder abrir más archivos.

La rapidez con la que se ejecutan los *"finalizers"* y *"cleaners"* depende del algoritmo de recolección de basura, que varía entre distintas implementaciones de la JVM. Por lo tanto, un programa que dependa de la ejecución rápida de estos métodos podría funcionar perfectamente en una JVM durante las pruebas, pero fallar catastróficamente en otra JVM.

La finalización tardía no es solo un problema teórico. Proporcionar un *"finalizer"* para una clase puede retrasar arbitrariamente la recolección de sus instancias. La especificación del lenguaje no garantiza qué hilo ejecutará los *"finalizers"*. El hilo encargado de ejecutar los *"finalizers"* pueden estar corriendo con una prioridad más baja que otro hilo de la aplicación, lo que puede provocar que los objetos no se finalicen con la rapidez que necesita la aplicación, pudiendo provocar errores de tipo `OutOfMemoryError`.

Los *"cleaners"* son una ligera mejora en este aspecto, ya que los autores de una clase pueden controlar sus propios hilos de limpieza. Sin embargo, los *"cleaners"* siguen ejecutándose en segundo plano, bajo el control del 'Recolector de Basura', por lo que tampoco garantizan una limpieza inmediata.

La especificación no solo no garantiza que los *"finalizers"* o *"cleaners"* se ejecuten de manera rápida, sino que **tampoco garantiza que se ejecuten en absoluto**. Es totalmente posible, e incluso probable, que un programa termine sin haber ejecutado los *"finalizers"* o *"cleaners"* en algunos objetos que ya no son accesibles. Como consecuencia, **nunca se debe depender de un *"finalizer"* o *"cleaner"* para actualizar un estado persistente**. Por ejemplo, confiar en uno de estos métodos para liberar un bloqueo persistente sobre un recurso compartido, como una base de datos, **es una receta segura para paralizar por completo un sistema distribuido**.

No hay que dejarse engañar por los métodos `System.gc()` y `System.runFinalization()`. Aunque pueden aumentar la probabilidad de que los *"finalizers"* o *"cleaners"* se ejecuten, **no lo garantizan**. Hubo dos métodos que en su momento afirmaban hacer esta garantía. Sin embargo, son métodos son **fatalmente defectuosos y han estado obsoletos durante décadas**:

- `System.runFinalizersOnExit()`
- `Runtime.runFinalizersOnExit()` (su "gemelo malvado")

Otro problema con los *"finalizers"* es que si se lanza una **excepción no capturada** durante la finalización, esta será **ignorada**, y la finalización de ese objeto **se interrumpirá**. Las excepciones no capturadas pueden dejar otros objetos en un estado corrupto. Si otro hilo intenta usar un objeto en este estado, el comportamiento resultante puede ser arbitrario e impredecible. Normalmente, una excepción no capturada **terminará el hilo y mostrará un *stack trace***, pero esto no ocurre en un *"finalizer"*, ya que **ni siquiera imprimirá una advertencia**.

Los *"cleaners"* no tienen este problema, ya que una biblioteca que use un cleaner **puede controlar el hilo en el que se ejecuta**.

Los *"finalizers"* tienen un **grave problema de seguridad**: abren la puerta a ***"finalizer attacks"***. La idea detrás de este ataque es simple. Si se lanza una excepción desde un constructor o sus equivalentes en la serialización —los métodos `readObject` y `readResolve`, el *"finalizer"* de una subclase maliciosa puede ejecutarse sobre el objeto parcialmente construido que debería haberse "muerto en el proceso". Este *"finalizer"* puede guardar una referencia al objeto en un campo estático, evitando que sea recolectado por el *"Garbage Collector"*. Una vez que el objeto malformado ha sido registrado, es cuestión de invocar métodos arbitrarios sobre este objeto, que no deberían haber existido en primer lugar.

Lanzar una excepción desde un constructor debería ser suficiente para evitar que un objeto entre en existencia; sin embargo, en presencia de *"finalizers"*, no lo es. Este tipo de ataques puede tener consecuencias graves. Las clases finales son inmunes a los ataques de *"finalizers"*, porque nadie puede escribir una subclase maliciosa de una clase final. Para proteger las clases no finales de estos ataques, **escribe un método `finalize()` final que no haga nada**.

Entonces, ¿qué se debe hacer en lugar de escribir un *"finalizer"* o *"cleaner"* para una clase cuyos objetos encapsulan recursos que requieren terminación, como archivos o hilos? Simplemente hacer que la clase implemente `AutoCloseable` y **requerir que los clientes invoquen el método `close`** en cada instancia cuando ya no sea necesaria, típicamente usando `try-with-resources` para garantizar la terminación incluso en caso de excepciones.

Un detalle importante es que la instancia debe llevar un registro de si ha sido cerrada: el método `close` debe **registrar en un campo** que el objeto ya no es válido, y otros métodos deben verificar este campo y lanzar una `IllegalStateException` si se llaman después de que el objeto haya sido cerrado.

Los *"finalizers"* y *"cleaners"* pueden tener algunos usos legítimos, aunque son limitados.

Como red de seguridad para liberar recursos olvidados. Si un recurso no es cerrado correctamente por el programador (por ejemplo, el cliente olvida llamar a `close()` en un recurso como un archivo o una conexión), estos métodos pueden ayudar a liberar ese recurso tarde, en lugar de no liberarlo en absoluto. Sin embargo, debido a la falta de garantías sobre cuándo o si se ejecutarán, esto no debería ser la solución principal. Algunas clases de bibliotecas de Java, como `FileInputStream`, `FileOutputStream`, `ThreadPoolExecutor` y `java.sql.Connection`, los utilizan como una red de seguridad en caso de que el cliente olvide cerrar el recurso.

El otro uso son con objetos con pares nativos. En Java, un par nativo es un objeto que interactúa con código nativo a través de métodos `native`. Estos objetos nativos no son gestionados por el recolector de basura de Java, por lo que, si el objeto Java asociado es reclamado, el recolector no podrá liberar el par nativo. En este caso, un *"finalizer"* o un *"cleaner"* puede servir para liberar estos pares nativos cuando el objeto Java asociado ya no sea necesario. No obstante, este enfoque solo es adecuado si el rendimiento es aceptable y si el par nativo no contiene recursos críticos que deban ser liberados de inmediato. En caso contrario, debería usarse un método `close()` para asegurar la liberación inmediata de recursos.

En resumen, no se debe utilizar *"cleaners"* ni *"finalizers"* (en versiones anteriores a Java 9), excepto como red de seguridad o para terminar recursos nativos no críticos. Incluso en estos casos, hay que tener en cuenta la indeterminación y las consecuencias de rendimiento.

Item 9: Prefer try-with-resources to try-finally

TODO