5 secretos ocultos en Java

por Justin Albano MVB · 21 y 18 de febrero · Zona de Java

Descargue Microservices for Java Developers : una introducción práctica a frameworks y contenedores. Presentado en asociación con Red Hat .

A medida que crecen los lenguajes de programación, es inevitable que las características ocultas comiencen a aparecer y los constructos que nunca fueron concebidos por los fundadores comiencen a introducirse en el uso común. Algunas de estas características aparecen como expresiones idiomáticas y se convierten en lenguaje aceptado en el lenguaje, mientras que otras se vuelven anti-patrones y quedan relegadas a los rincones oscuros de la comunidad lingüística. En este artículo, analizaremos cinco secretos de Java que a menudo pasan desapercibidos para la gran cantidad de desarrolladores de Java (algunos por una buena razón). Con cada descripción, veremos los casos de uso y la lógica que llevaron cada característica a la existencia y veremos algunos ejemplos cuando sea apropiado usar estas características.

El lector debe tener en cuenta que no todas estas características no están realmente ocultas en el lenguaje, pero a menudo no se utilizan en la programación diaria. Mientras que algunos pueden ser muy útiles en los momentos apropiados, otros son casi siempre una mala idea y se muestran en este artículo para ver el interés del lector (y posiblemente darle una buena risa). El lector debe usar su criterio al decidir cuándo usar las funciones descritas en este artículo: el hecho de que se pueda hacer no significa que deba hacerlo

que deba hacerlo.

1. Implementación de la anotación

Desde Java Development Kit (JDK) 5, las anotaciones tienen una parte integral de muchas aplicaciones y frameworks Java. En la gran mayoría de los casos, las anotaciones se aplican a construcciones de lenguaje, como clases, campos, métodos, etc., pero hay otro caso en el que se pueden aplicar anotaciones: como interfaces implementables. Por ejemplo, supongamos que tenemos la siguiente definición de anotación:

```
1  @Retention ( RetentionPolicy . RUNTIME )
2  @Target ( ElementType . MÉTODO )
3  public  @interface  Test {
4      String  name ();
5  }
```

Normalmente, aplicaremos esta anotación a un método, como se muestra a continuación:

```
1   clase  pública MyTestFixure {
2
3      @Prueba
        vacío  público dadoFooWhenBarThenBaz () {
4
5          // ...
6      }
7  }
```

A continuación, podemos procesar esta anotación, como se describe en Crear anotaciones en Java . Si también quisiéramos crear una interfaz que permita que las pruebas se creen como objetos, tendríamos que crear una nueva interfaz, nombrándola de otra forma que no sea `Test` :

```
1   interfaz  pública TestInstance {
2      public  String  getName ();
3  }
```

Entonces podríamos instanciar un `TestInstance` objeto:

```
clase  pública FooTestInstance {

    public  String  getName () {
        devolver  "Foo" ;
    }
}

TestInstance  myTest  =  new  FooTestInstance (
```

Si bien nuestra anotación e interfaz son casi idénticas, con una duplicación muy notable, no parece haber una manera de fusionar estas dos construcciones. Afortunadamente, las apariencias engañan y existe una técnica para fusionar estos dos constructos: Implementar la anotación:

```
clase  pública FooTest  implementa  Test {

    @Anular
    public  String  name () {
        devolver  "Foo" ;
    }

    @Anular
     clase pública <?  extiende  Annotation >
        volver  prueba . clase ;
    }
}
```

Tenga en cuenta que debemos implementar el `annotationType` método y devolver el tipo de la anotación también, ya que esto es parte implícita de la `Annotation` interfaz. Aunque en casi todos los casos, la implementación de una anotación no es una buena decisión de diseño (el compilador de Java mostrará una advertencia al implementar una interfaz), puede ser útil en algunas circunstancias selectas, como en los marcos basados en anotaciones.

2. Inicialización de instancia

En Java, como ocurre con la mayoría de los lenguajes

En Java, como ocurre con la mayoría de los lenguajes de programación orientados a objetos, los objetos se instancian exclusivamente utilizando un constructor (con algunas excepciones críticas, como la deserialización de objetos Java). Incluso cuando creamos métodos de fábrica estáticos para crear objetos, simplemente estamos envolviendo una llamada al constructor de un objeto para crear una instancia. Por ejemplo:

```
clase  pública Foo {

   nombre de secuencia final  privada ;

   Foo privado ( String  name ) {
      esta . nombre  =  nombre ;
   }

   public  static  Foo  withName ( String  name
      devolver  nuevo  Foo ( nombre );
   }
}

Foo  foo  =  Foo . withName ( "Barra" );
```

Por lo tanto, cuando deseamos inicializar un objeto, consolidamos la lógica de inicialización en el constructor del objeto. Por ejemplo, establecemos el `name` campo de la `Foo` clase dentro de su constructor parametrizado. Si bien puede parecer una sólida suposición que *toda* la lógica de inicialización se encuentra en el constructor o conjunto de constructores para una clase, este no es el caso en Java. En su lugar, también podemos usar la inicialización de la instancia para ejecutar el código cuando se crea un objeto:

```
clase  pública Foo {

   {
      Sistema . a cabo . println ( "Foo: inst
   }

   público  Foo () {
      Sistema . a cabo . println ( "Foo: cons
```

```
    }
}
```

Los inicializadores de instancia se especifican agregando lógica de inicialización dentro de un conjunto de llaves dentro de la definición de una clase. Cuando se crea una instancia del objeto, sus inicializadores de instancia se llaman primero, seguidos por sus constructores. Tenga en cuenta que se puede especificar más de un inicializador de instancia, en cuyo caso, cada uno se llama en el orden en que aparece dentro de la definición de clase. Además de los inicializadores de instancias, también podemos crear inicializadores estáticos, que se ejecutan cuando la clase se carga en la memoria. Para crear un inicializador estático, simplemente prefijamos un inicializador con la palabra clave `static`:

```
clase  pública Foo {

    {
        Sistema . a cabo . println ( "Foo: inst
    }

    static {
        Sistema . a cabo . println ( "Foo: stat
    }

    público  Foo () {
        Sistema . a cabo . println ( "Foo: cons
    }
}
```

Cuando las tres técnicas de inicialización (constructores, inicializadores de instancia e inicializadores estáticos) están presentes en una clase, los inicializadores estáticos siempre se ejecutan primero (cuando la clase se carga en la memoria) en el orden en que se declaran, seguidos de los inicializadores de instancia en el orden son declarados, y finalmente por constructores. Cuando

se introduce una superclase, el orden de ejecución cambia ligeramente:

1. Inicializadores estáticos de superclase, en orden de su declaración
2. Inicializadores estáticos de subclase, en orden de su declaración
3. Iniciales de instancias de superclase, en orden de su declaración
4. Constructor de superclase
5. Iniciales de instancia de la subclase, en orden de su declaración
6. Constructor de la subclase

Por ejemplo, podemos crear la siguiente aplicación:

```
clase de resumen  público Bar {

    nombre de cadena  privada ;

   static {
       Sistema . a cabo . println ( "Barra: es
   }

   {
       Sistema . a cabo . println ( "Barra: ir
   }

   static {
       Sistema . a cabo . println ( "Barra: es
   }

    Bar público () {
       Sistema . a cabo . println ( "Barra: cc
   }

   {
       Sistema . a cabo . println ( "Barra: ir
   }

    Bar público ( String  name ) {
       esta . nombre  =  nombre ;
       Sistema . a cabo . println ( "Barra: nc
```

```
        Sistema . a cabo . println ( "Barra: no
    }
}

la  clase  pública Foo  extiende la  barra {

    static {
        Sistema . a cabo . println ( "Foo: stat
    }

    {
        Sistema . a cabo . println ( "Foo: inst
    }

    static {
        Sistema . a cabo . println ( "Foo: está
    }

    público  Foo () {
        Sistema . a cabo . println ( "Foo: cons
    }

    public  Foo ( String  name ) {
        super ( nombre );
        Sistema . a cabo . println ( "Foo: nomb
    }

    {
        Sistema . a cabo . println ( "Foo: inst
    }

    public  static  void  main ( String ... arg
        nuevo  Foo ();
        Sistema . a cabo . println ();
        nuevo  Foo ( "Baz" );
    }
}
```

Si ejecutamos este código, recibimos el siguiente resultado:

resultado:

```
Bar: estático 1
Bar: estático 2
Foo: estático 1
Foo: estático 2
Bar: instancia 1
Bar: instancia 2
Bar: constructor
Foo: instancia 1
Foo: instancia 2
Foo: constructor

Bar: instancia 1
Bar: instancia 2
Bar: nombre-constructor
Foo: instancia 1
Foo: instancia 2
Foo: nombre-constructor
```

Tenga en cuenta que los inicializadores estáticos solo se ejecutaron *una vez* , aunque `Foo` se crearon dos objetos. Mientras que la instancia y los inicializadores estáticos pueden ser útiles, la lógica de inicialización debe colocarse en constructores y los métodos (o métodos estáticos) deben usarse cuando se requiere una lógica compleja para inicializar el estado de un objeto.

3. Inicialización de llaves dobles

Muchos lenguajes de programación incluyen algún mecanismo sintáctico para crear rápida y concisamente una lista o mapa (o diccionario) sin usar un código descriptivo detallado. Por ejemplo, C ++ incluye la inicialización de llaves que permite a los desarrolladores crear rápidamente una lista de valores enumerados, o incluso inicializar objetos enteros si el constructor del objeto admite esta funcionalidad. Lamentablemente, antes de JDK 9, no se incluyó dicha característica (en breve abordaremos esta inclusión). Para crear ingenuamente una lista de objetos, haríamos lo siguiente:

```
Lista < Integer >  myInts  =  new  ArrayList <>
```

```
myInts . agregar ( 1 );
myInts . agregar ( 2 );
myInts . agregar ( 3 );
```

Si bien esto logra nuestro objetivo de crear una nueva lista inicializada con tres valores, es demasiado detallado, lo que requiere que el desarrollador repita el nombre de la variable de la lista para cada adición. Para acortar este código, podemos usar la inicialización de doble llave para agregar los mismos tres elementos:

```
Lista < Integer > myInts = new ArrayList <>
    agregar ( 1 );
    agregar ( 2 );
    agregar ( 3 );
}};
```

La inicialización de doble corchete, que toma su nombre del conjunto de dos llaves abiertas y cerradas, es en realidad un compuesto de múltiples elementos sintácticos. Primero, creamos una clase interna anónima que amplía la `ArrayList` clase. Como `ArrayList` no tiene métodos abstractos, podemos crear un cuerpo vacío para la implementación anónima:

```
Lista < Integer > myInts = new ArrayList <>
```

Al usar este código, básicamente creamos una subclase anónima `ArrayList` que es exactamente igual a la original `ArrayList` . Una de las principales diferencias es que nuestra clase interna tiene una referencia implícita a la clase contenedora (en forma de una `this` variable capturada) ya que estamos creando una clase interna no estática. Esto nos permite escribir alguna lógica interesante, si no intrincada, como agregar la `this` variable capturada a la clase interna anónima de doble llave inicializada:

```
clase pública Foo {

    public List < Foo > getListWithMeIncludec
```

```
        devolver  nuevo  ArrayList < Foo > () {
            añadir ( Foo . este );
        }};
    }

    public  static  void  main ( String ... arg
        Foo  foo  =  nuevo  Foo ();
        Lista < Foo >  fooList  =  foo . getLis
        Sistema . a cabo . println ( foo . equa
    }
}
```

Si esta clase interna estuviera definida estáticamente, no tendríamos acceso a ella `Foo.this`. Por ejemplo, el siguiente código, que crea estáticamente la `FooArrayList` clase interna nombrada, no tiene acceso a la `Foo.this` referencia y, por *lo* tanto, *no es compilable* :

```
clase  pública Foo {

    public  List < Foo >  getListWithMeIncludec
        devolver el  nuevo  FooArrayList ();
    }

     clase privada estática  FooArrayList extie
        añadir ( Foo . este );
    }}
}
```

Reanudando la construcción de nuestra llave doble inicializada `ArrayList`, una vez que hemos creado la clase interna no estática, usamos la inicialización de la instancia, como vimos anteriormente, para ejecutar la adición de los tres elementos iniciales cuando se crea una instancia de la clase interna anónima. Como las clases internas anónimas se instancian inmediatamente y solo existe un objeto de la clase interna anónima, esencialmente hemos creado un

objeto singleton interno no estático que agrega los tres elementos iniciales cuando se crea. Esto se puede hacer más obvio si separamos el par de llaves, donde una llave representa claramente la definición de la clase interna anónima y la otra llave denota el inicio de la lógica de inicialización de la instancia:

```
Lista < Integer > myInts = new ArrayList <>
{
    agregar ( 1 );
    agregar ( 2 );
    agregar ( 3 );
}
};
```

Si bien este truco puede ser útil, JDK 9 (JEP 269) ha suplantado la utilidad de este truco con un conjunto de métodos de fábrica estáticos para `List` (así como muchos de los otros tipos de colección) . Por ejemplo, podríamos haber creado lo `List` anterior utilizando estos métodos de fábrica estáticos, como se ilustra en la siguiente lista:

```
Lista < Integer > myInts = Lista . de ( 1 ,
```

Esta técnica de fábrica estática es deseable por dos razones principales: (1) No se crea una clase interna anónima y (2) la reducción en el código repetitivo (ruido) requerido para crear el `List` . La advertencia para crear una `List` de esta manera es que el resultado `List` es inmutable y, por lo tanto, no se puede modificar una vez que se ha creado. Para crear un mutable `List` con los elementos iniciales deseados, estamos atrapados con el uso de la técnica ingenua o la inicialización con doble llave.

Tenga en cuenta que la inicialización ingenua, la inicialización con doble llave y los métodos de fábrica estáticos del JDK 9 no solo están disponibles `List` . También están disponibles para `Set` y `Map` objetos, como se ilustra en el siguiente fragmento:

```
// Inicialización ingenua
Mapa < String , Integer > myMap = new HashM
```

```
myMap . put ( "Foo" , 10 );
myMap . put ( "Bar" , 15 );

// Inicialización con doble llave
Mapa < String , Integer >  myMap  =  new  HashM
    put ( "Foo" , 10 );
    put ( "Bar" , 15 );
}};

// Inicialización de fábrica estática
Mapa < String , Integer >  myMap  =  Mapa . de
```

Es importante considerar la naturaleza de la inicialización de llaves dobles antes de decidir usarla. Si bien mejora la legibilidad del código, conlleva algunos efectos secundarios implícitos.

4. Comentarios ejecutables

Los comentarios son una parte esencial de casi todos los programas y el principal beneficio de los comentarios es que no se ejecutan. Esto se hace aún más evidente cuando comentamos una línea de código dentro de nuestro programa: queremos conservar el código en nuestra aplicación, pero no queremos que se ejecute. Por ejemplo, los siguientes resultados del programa `5` se imprimen a la salida estándar:

```
public  static  void  main ( String  args []) {
    int  value  =  5 ;
    // valor = 8;
    Sistema . a cabo . println ( valor );
}
```

Si bien es una suposición fundamental que los comentarios nunca se ejecutan, no es completamente cierto. Por ejemplo, ¿qué imprime el siguiente fragmento en la salida estándar?

```
public  static  void  main ( String  args []) {
```

```
2       int  value  =  5 ;
3       // \ u000dvalue = 8;
4       Sistema . a cabo . println ( valor );
5   }
```

Una buena suposición sería `5` otra vez, pero si ejecutamos el código anterior, vemos `8` impreso a la salida estándar. La razón detrás de este error aparente es el carácter Unicode `\u000d` ; este carácter es en realidad un retorno de carro Unicode , y el compilador consume el código fuente de Java como archivos de texto con formato Unicode. Agregar este retorno de carro empuja la asignación `value = 8` a la línea que sigue directamente al comentario, asegurándose de que se ejecute. Esto significa que el fragmento de arriba es efectivamente igual a lo siguiente:

```
public  static  void  main ( String  args []) {
1
2       int  value  =  5 ;
3       //
4   valor  =  8 ;
5       Sistema . a cabo . println ( valor );
6   }
```

Aunque esto parece ser un error en Java, en realidad es una inclusión consciente en el lenguaje. El objetivo original de Java era crear un lenguaje independiente de la plataforma (de ahí la creación de la Máquina Virtual Java o JVM) y la interoperabilidad del código fuente es un aspecto clave de este objetivo. Al permitir que el código fuente de Java contenga caracteres Unicode, podemos incluir caracteres no latinos de manera universal. Esto garantiza que el código escrito en una región del mundo (que puede incluir caracteres no latinos, como en los comentarios) se puede ejecutar en cualquier otro. Para obtener más información, consulte la Sección 3.3 de la Especificación del lenguaje Java o JLS .

Podemos llevar esto al extremo e incluso escribir una aplicación completa en Unicode. Por ejemplo, ¿qué hace el siguiente programa (código fuente obtenido de Java: código de ejecución en comentarios ?!)?

```
\ u0070 \ u0075 \ u002c \ u006c \ u0069 \ u003e
\ u0063 \ u006c \ u0061 \ u0073 \ u0073 \ u0020
\ u007b \ u0070 \ u0075 \ u002c \ u006c \ u0069
\u0020\u0020\u0020\u0020\u0073\u0074\u0061\u007
\u0076\u006f\u0069\u0064\u0020\u006d\u0061\u006
\u0053\u0074\u0072\u0069\u006e\u0067\u005b\u005
\u0020\u0020\u0020\u0020\u0061\u0072\u0067\u007
\u0053\u0079\u0073\u0074\u0065\u006d\u002e\u006
\u002e\u0070\u0072\u0069\u006e\u0074\u006c\u006
\u0022\u0048\u0065\u006c\u006c\u006f\u0020\u007
\u0022\u006f\u0072\u006c\u0064\u0022\u0029\u003
```

If the above is placed in a file named Ugly.java and executed, it prints `Hello world` to standard output. If we convert these escaped Unicode characters into American Standard Code for Information Interchange (ASCII) characters, we obtain the following program:

```
public
class Ugly
{public
    static
void main(
String[]
    args){
System.out
.println(
"Hello w"+
"orld");}}
```

Although it is important to know that Unicode characters can be included in Java source code, it is highly suggested that they are avoided unless required (for example, to include non-Latin characters in comments). If they are required, be sure not to include characters, such as carriage return, that

change the expected behavior of the source code.

5. Enum Interface Implementation

One of the limitations of enumerations (enums) compared to classes in Java is that enums cannot extend another class or enum. For example, it is *not possible* to execute the following:

```
public class Speaker {

    public void speak() {
        System.out.println("Hi");
    }
}

public enum Person extends Speaker {

    JOE("Joseph"),
    JIM("James");

    private final String name;

    private Person(String name) {
        this.name = name;
    }
}

Person.JOE.speak();
```

We can, however, have our enum implement an interface and provide an implementation for its abstract methods as follows:

```
public interface Speaker {
    public void speak();
}

public enum Person implements Speaker {

    JOE("Joseph"),
    JIM("James");

    private final String name;

    private Person(String name) {
```

```
    private Person(String name) {
        this.name = name;
    }

    @Override
    public void speak() {
        System.out.println("Hi");
    }
}

Person.JOE.speak();
```

We can now also use an instance of `Person` anywhere a `Speaker` object is required. Whatsmore, we can also provide an implementation of the abstract methods of an interface on a per-constant basis (called constant-specific methods):

```
public interface Speaker {
    public void speak();
}

public enum Person implements Speaker {

    JOE("Joseph") {
        public void speak() { System.out.print
    },
    JIM("James"){
        public void speak() { System.out.print
    };

    private final String name;

    private Person(String name) {
        this.name = name;
    }

    @Override
    public void speak() {
        System.out.println("Hi");
    }
}

Person.JOE.speak();
```

Unlike some of the other secrets in this article, this technique should be encouraged where appropriate. For example, if an enum constant, such as `JOE` or `JIM`, can be used in place of an interface type, such as `Speaker`, the enum that defines the constant should implement the interface type. For more information, see Item 38 (pp. 176-9) of Effective Java, 3rd Edition.

Conclusion

In this article, we looked at five hidden secrets in Java, namely: (1) Annotations can be extended, (2) instance initialization can be used to configure an object upon instantiation, (3) double-brace initialization can be used to execute instructions when creating an anonymous inner class, (4) comments can sometimes be executed, and (5) enums can implement interfaces. While some of these features have their appropriate uses, some of them should be avoided (i.e. creating executable comments). When deciding to use these secrets, be sure to obey the following rule: Just because something can be done, does not mean that it should.

Topics: JAVA , INITIALIZATION , EXECUTABLE COMMENTS , ENUM INTERFACE , ANNOTATIONS , TUTORIAL

Java Partner Resources