

¿Qué ofrece Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
 Ese apoyo que siempre quiso tener...

1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces,
 HTML5, CSS3, JavaScript-jQuery

Control de autenticación y
 acceso (Spring Security)
 UDDI

JPA-Hibernate, MyBatis
 Motor de búsqueda empresarial (Solr)
 ETL (Talend)

Gestor portales (Liferay)
 Gestor de contenidos (Alfresco)
 Aplicaciones híbridas

Web Services
 Rest Services
 Social SSO
 SSO (Cas)

Dirección de Proyectos Informáticos.
 Metodologías ágiles
 Patrones de diseño
 TDD

Tareas programadas (Quartz)
 Gestor documental (Alfresco)
 Inversión de control (Spring)

BPM (jBPM o Bonita)
 Generación de informes (JasperReport)
 ESB (Open ESB)

» Estás en: [Inicio](#) [Tutoriales](#) [Mixins en Java y Java8 !Sí, es posible!](#)

Alejandro Pérez García

Alejandro es socio fundador de Autentia y nuestro experto en J2EE, Linux y optimización de aplicaciones empresariales.

Ingeniero en Informática y Certified ScrumMaster

Seguir a @alejandropgarcia 1,145 seguidores

Si te gusta lo que ves, puedes contratarle para darte ayuda con soporte experto, impartir **cursos presenciales** en tu empresa o para que **realicemos tus proyectos como factoría** (Madrid). Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación.


[Ver todos los tutoriales del autor](#)


Fecha de publicación del tutorial: 2015-03-06

Tutorial visitado 2 veces [Descargar en PDF](#)

Mixins en Java y Java8 !Sí, es posible!

Creación: 20-02-2015

Índice de contenidos

1. Introducción
2. Entorno
3. Java Mixin, la implementación manual
4. Java Mixin, en runtime, gracias a un proxy dinámico
 - 4.1. Ejemplo de uso de la librería java-mixins
 - 4.2. La clave de la librería java-mixins
5. Java Mixin con Java 8
6. Conclusiones
7. Sobre el autor

1. Introducción

Un **Mixin** es una forma de incluir métodos de una clase en otra, sin que exista relación de herencia entre ellas. En cierto sentido se puede ver como una especie de "herencia" múltiple, pero sin existir relación de especialización entre las clases.

De esta definición la parte más importante es la de que **no existe relación de herencia**, ya que, si bien el lenguaje Java no soporta directamente los mixins, es precisamente esta falta de relación de herencia la que nos va a permitir implementarlos (si fuera necesaria una relación de herencia entonces sí que sería totalmente imposible implementarlo en Java, ya que Java sólo permite herencia simple).

Antes de seguir avanzando vamos a definir un poco más que es un mixin, comparándolo con una interfaz y un trait:

- **Interfaz** - sólo tiene definición de métodos
- **Trait** - tiene definición de métodos + implementación de los mismos
- **Mixin** - tiene definición de métodos + implementación + estado

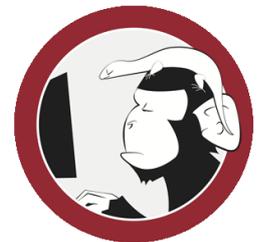
Aquí vemos otro aspecto importante de los mixins, y es que **pueden tener estado**. Es decir, las clases que vamos a usar para componer el mixin pueden tener atributos que serán "añadidos" al mixin.

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil MacBook Pro 15" (2.3 GHz Intel i7, 16GB 1600 Mhz DDR3, 500GB Flash Storage).
- NVIDIA GeForce G7 750M
- Sistema Operativo: Mac OS X Lion 10.10.2
- Java Virtual Machine (JVM) 7 y 8

Catálogo de servicios Autentia



Síguenos a través de:



Últimas Noticias

- » 2015: ¡Volvemos a la oficina!
- » [Curso JBoss de Red Hat](#)
- » Si eres el responsable o líder técnico, considérate desafortunado. No puedes culpar a nadie por ser gris
- » Portales, gestores de contenidos documentales y desarrollos a medida
- » [Comentando el libro Start-up Nation, La historia del milagro económico de Israel, de Dan Senor & Salu Singer](#)

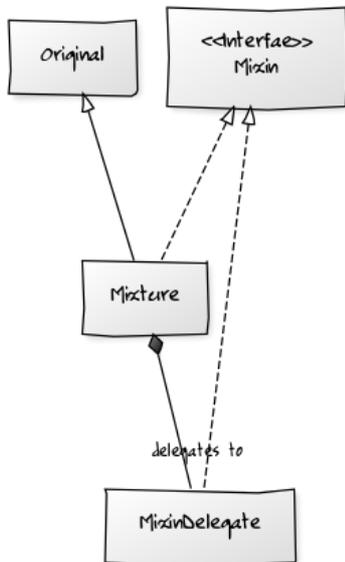
[Histórico de noticias](#)

Últimos Tutoriales

- » [Introducción a la Sandbox HDP - Hortonworks Data Platform](#)
- » [Configura e interpreta las métricas de Sonarqube para conocer la calidad de tu código](#)
- » [Jugando con Optional en Java 8](#)
- » [Novedades en Illustrator CC](#)
- » [Cómo crear un mapa interactivo en CartoDB](#)

3. Java Mixin, la implementación manual

Si no existe relación de herencia podemos ver un mixin como una especie de composición y delegación:



Java Mixin manual

Vemos como la clase `Mixture` es la mezcla de extender la clase `Original` e implementar la interfaz `Mixin`, delegando la implementación y ejecución de los métodos de esta interfaz en la clase `MixinDelegate`. `MixinDelegate` es la clase que estamos incluyendo en la `Original` a modo de *mixin*.

En código puede quedar algo similar a:

```

1  class Original {
2      private int foo = 42;
3      public int getFoo() { return foo; }
4  }
5
6  interface Mixin {
7      void print();
8  }
9
10 class MixinDelegate implements Mixin {
11     private final Object original;
12     MixinDelegate(Object original) { this.original = original; }
13
14     @Override
15     public void print() {
16         System.out.println(original.getFoo());
17     }
18 }
19
20 class Mixture extends Original implements Mixin {
21     private final Mixin mixin = new MixinDelegate(this);
22
23     @Override
24     public void print() {
25         mixin.print();
26     }
27 }
28
29 class Main {
30     public static final main(String[] args) {
31         Mixture mixture = new Mixture();
32         mixture.getFoo();
33         mixture.print();
34     }
35 }

```

En este ejemplo cabe destacar como en las líneas 32-33 estamos usando tanto los métodos de `Original` como los de `Mixin`, así que podemos decir que hemos conseguido el efecto que deseábamos.

Esta implementación tiene dos grandes problemas:

1. Si la interfaz `Mixin` tiene muchos métodos, **vamos a tener mucho código duplicado** en la clase `Mixture` (como el de las líneas 23 a 26), además de que lo vamos a tener que copiar en cada nueva clase *mezcla* que queramos hacer.
2. Para cada *mezcla* tenemos que crear una nueva clase (como la clase `Mixture`) que tiene que conocer los mixins que le queremos aplicar al *original* (líneas 21 y 25). Es decir, **hay demasiado acoplamiento** entre el mixin y la clase original donde se están incluyendo los métodos.

4. Java Mixin, en runtime, gracias a un proxy dinámico

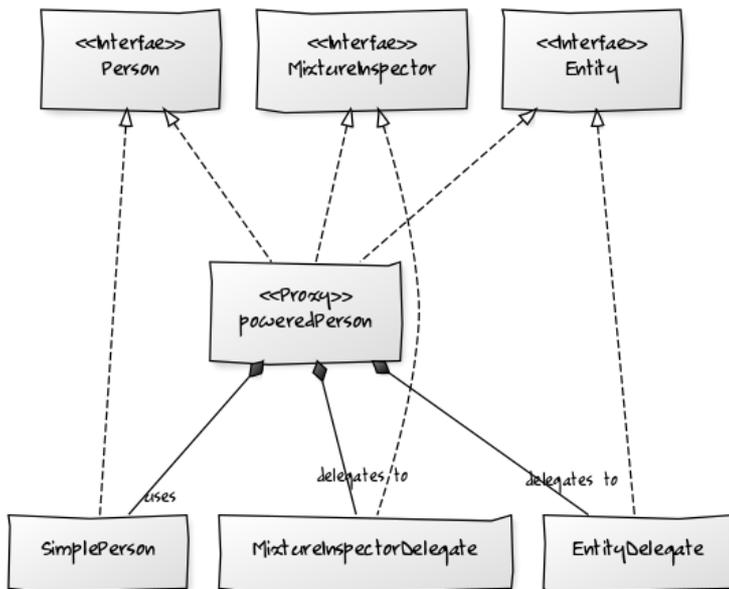
He creado una pequeña librería que, utilizando sólo los mecanismos que proporciona Java, nos permite realizar *mixtures en runtime y sin perder el tipado fuerte*, corrigiendo, además, los dos problemas que comentábamos en el apartado anterior. Es decir no tenemos que escribir código repetitivo y sobre todo, la clase *original* y el *mixin* no se conocen de nada.

La librería **Java Mixin** la podés encontrar en el [GitHub](#) de de Autentia.

El siguiente diagrama representa el ejemplo que vamos a utilizar:

Últimos Tutoriales del Autor

- » [Cómo hacer testing automático de un applet Java](#)
- » [Primeros pasos con Clojure: Leiningen y Midje](#)
- » [Cómo integrar en Gradle un servidor Jetty o Tomcat](#)
- » [Kettle no es una tetera, es la herramienta de ETL de Pentaho!](#)
- » [Crea todo un entorno de máquinas virtuales con un solo comando, gracias a Vagrant](#)



Igual que en el apartado anterior vemos como un mixin se compone de una interfaz y una clase que implementa los métodos de esta y sobre la que, la clase original, delegará la ejecución. Así en el ejemplo podemos identificar dos mixins:

- *Entity*, formado por la interfaz *Entity* y la clase delegado *EntityDelegate*.
- *MixtureInspector*, formado por la interfaz *MixtureInspector* y la clase delegado *MixtureInspectorDelegate*.

En el diagrama vemos como al final, el Proxy dinámico *poweredPerson* cumplirá las interfaces de *Person*, *Entity* y *MixtureInspector*, por lo que sobre este objeto podremos llamar a cualquier método de estas interfaces.

4.1. Ejemplo de uso de la librería java-mixins

Pero tranquilos porque aunque parece complicado, la librería que he preparado va a hacer que el uso sea muy sencillo. Veámoslo con un ejemplo de uso:

```

1 final Person originalPerson = new SimplePerson("John", "Doe");
2
3 final Person person = new MixerBuilder(Person.class)
4     .include(new Mixin(Entity.class, EntityDelegate.class))
5     .include(new Mixin(MixtureInspector.class, MixtureInspectorDelegate.class))
6     .build()
7     .mixWith(originalPerson);

```

En este código podemos hacer los siguietes comentarios:

- línea 1 - creamos el objeto donde queremos aplicar los mixins. Es una creación normal, de hecho la podríamos haber hecho directamente en la línea 7, pero he preferido separarla en una variable local para que quede más claro que se trata del objeto original.
- línea 3 - creamos el *MixerBuilder* indicando el tipo de los mixtures que devolverá el *Mixer*.
- línea 4 y 5 - creamos los *Mixin* y los incluimos en el *MixerBuilder*. Aquí queda claro como cada mixin es una pareja de interfaz más una clase delegada que implementa los métodos.
- línea 6 - construimos el *Mixer*.
- línea 7 - le decimos al *Mixer* sobre que instancia queremos aplicar los mixins.

Ya que hemos separado la construcción del *Mixer* de la construcción de las mixtures (mezcla del objeto original más los mixins), podemos cachear fácilmente el *Mixer* para reutilizarlo en la construcción de muchos mixtures. Veamos un ejemplo:

```

1 // Builds the Factory just one time
2 final Mixer mixer = new MixerBuilder(PoweredPerson.class)
3     .include(new Mixin(Entity.class, EntityDelegate.class))
4     .include(new Mixin(MixtureInspector.class, MixtureInspectorDelegate.class))
5     .build();
6
7 // Builds all the mixtures that you want!
8 final PoweredPerson poweredPerson1 = mixer.mixWith(new SimplePerson("John", "Doe"));
9 final PoweredPerson poweredPerson2 = mixer.mixWith(new SimplePerson("Jane", "Doe"));
10 final PoweredPerson poweredPerson3 = mixer.mixWith(new SimplePerson("Joe", "Public"));

```

En los tests de la librería se pueden encontrar más ejemplos de uso.

4.2. La clave de la librería java-mixins

Os animo a que echéis un vistazo a todo el código, pero podríamos decir que el quid de la cuestión está en el método privado *createProxy* de la clase *Mixer*:

```

1 private Object createProxy(final Object original, final Map<Class<?>, Object> delegate?:
2     return proxyClass.createProxy(new InvocationHandler() {
3         @Override
4         public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
5             Object objectToCall = delegatesByInterfaceType.get(method.getDeclaringClass()
6             if (objectToCall == null) {
7                 objectToCall = original;
8             }
9
10            return method.invoke(objectToCall, args);
11        }
12    });
13

```

14 | }

Aquí vemos como se está creando un Proxy dinámico cuyo `InvocationHandler` lo que hace es:

- línea 5 - buscar en un mapa el delegado que corresponde con el tipo donde está definido el método que se está ejecutando.
- línea 7 - si no encuentra un delegado supone que el método pertenece al objeto original.
- línea 10 - ejecuta el método (bien en el delegado o en el objeto original).

5. Java Mixin con Java 8

Java 8 introduce una nueva construcción en el lenguaje que permite especificar una implementación por defecto para métodos de una interfaz.

```

1 interface MessagePrinter {
2     default printMessage() {
3         System.out.println("Este es el mensaje por defecto");
4     }
5 }

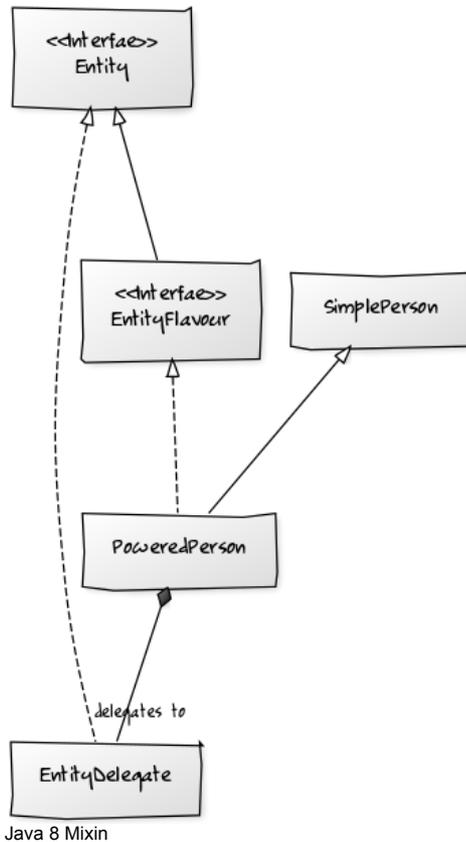
```

Según Oracle el principal objetivo de estas implementaciones por defecto es la de mantener la compatibilidad hacia atrás al añadir nuevos métodos a una interfaz ampliamente usada (como hace el propio Java con todo el API de colecciones para soportar Lambdas y Streams). De manera que al añadir estos nuevos métodos no se "rompa" todo el código que ya está escrito (antes de Java 8 si añadimos un método a una interfaz existente, todas las clases que usan esa interfaz dejarán de compilar).

Nosotros vamos a aprovechar esta capacidad para hacer una implementación de mixin. Esta implementación tendrá el mismo problema de alto acoplamiento que presentaba en el primer punto de este tutorial, cuando hacíamos los mixins a mano; pero por lo menos no tendremos el problema de la duplicación de código.

De hecho realmente lo que estamos implementando, tal como cuenta el artículo [Java 8: Now You Have Mixins?](#), es el *Virtual Field Pattern*.

El ejemplo que vamos a usar es:



El código sería:

```

1 class SimplePerson {
2     ...
3 }
4
5 interface Entity {
6     void save();
7 }
8
9 interface EntityFlavour extends Entity {
10     Entity getEntity();
11
12     @Override
13     default void save() { getEntity().save(); }
14 }
15
16 class EntityDelegate implements Entity {
17     private final Object original;
18     EntityDelegate(Object original) { this.original = original; }
19
20     @Override

```

```

21     public void save() {
22         ...
23     }
24 }
25
26 class PoweredPerson extends SimplePerson implements EntityFlavour {
27     private final EntityDelegate mixin = new MixinDelegate(this);
28
29     @Override
30     public Entity getEntity() {
31         return mixin;
32     }
33 }
34
35 class Main {
36     public static final main(String[] args) {
37         PoweredPerson mixture = new PoweredPerson();
38         mixture.getFoo();
39         mixture.save();
40     }
41 }

```

Aquí el quid de la cuestión está en la interfaz `EntityFlavour` (líneas 9 a 14) Esta interfaz añade el método `getEntity()`, que obliga a quien lo implemente a proporcionar un `Entity`, y luego sobrescribe todos los métodos de la interfaz `Entity` proporcionando una implementación por defecto en función de ese método. Es decir proporciona una **implementación por defecto que delega la ejecución del método sobre la instancia devuelta por `getEntity()`**.

Gracias a la interfaz `EntityFlavour` y sus implementaciones por defecto, ya no tenemos que repetir código; basta con que las clases donde queremos aplicar el mixin implementen esta interfaz (líneas 26 a 32).

6. Conclusiones

Este tutorial no pretende ser más que un ejercicio teórico, y no se si la librería `java-mixins` llegará muy lejos o tendrá utilidad real. Pero sí me parece interesante el estudio de como, mediante el uso de patrones, podemos implementar característica que nuestro lenguaje no soporta de forma nativa.

Os animo a que reviséis el [código del proyecto](#) aunque sólo sea como ejercicio para repasar como funciona un Proxy dinámico en Java.

Nota: Todos los diagramas UML han sido generados con yUML.

7. Sobre el autor

Alejandro Pérez García, Ingeniero en Informática (especialidad de Ingeniería del Software) y Certified ScrumMaster

Socio fundador de Autentia (Desarrollo de software, Consultoría, Formación)

<mailto:alejandropg@autentia.com>

Autentia Real Business Solutions S.L. - "Soporte a Desarrollo"

<http://www.autentia.com>

A continuación puedes evaluarlo:

[Regístrate para evaluarlo](#)

Por favor, vota +1 o compártelo si te pareció interesante

Share | 

Anímate y coméntanos lo que pienses sobre este **TUTORIAL**:

» **Regístrate** y accede a esta y otras ventajas «



Esta obra está licenciada bajo licencia [Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5](#)

IMPULSA

Impulsores

Comunidad

[¿Ayuda?](#)

sin clicks

0 personas han traído clicks a esta página

+ + + + + + + +

powered by [karmacrazy](#)

Copyright 2003-2015 © All Rights Reserved | [Texto legal y condiciones de uso](#) | [Banners](#) | [Powered by Autentia](#) | [Contacto](#)

