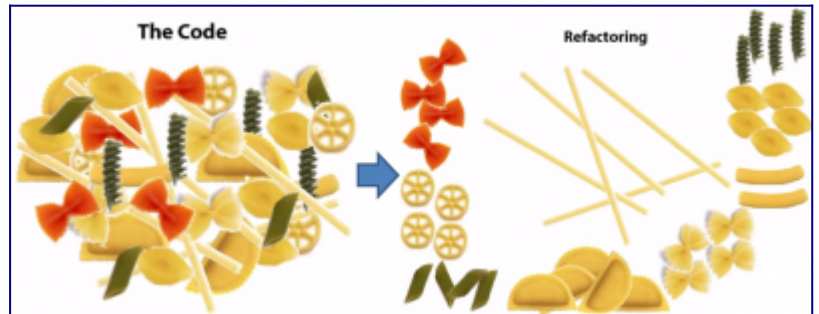


Refactorizacion

El término refactorizar dentro del campo de la Ingeniería del Software hace referencia a la modificación del código sin cambiar su funcionamiento. Se emplea para crear un código más claro y sencillo, facilitando la posterior lectura o revision de un programa. Se podría entender como el mantenimiento del código, para facilitar su comprensión, pero sin añadir ni eliminar funcionalidades. **Refactorizar código consiste en crear un código más limpio.**

La refactorización debe ser un paso aislado en el diseño de un programa, para evitar introducir errores de código al reescribir o modificar algunas partes del mismo. Si después de refactorizar hemos alterado el funcionamiento del código, hemos cometido errores al refactorizar.



Se refactoriza para:

- Limpieza del código, mejorando la consistencia y la claridad.
- Mantenimiento del código, sin corregir errores ni añadir funcionalidades.
- Elimina el código “muerto”, y se modulariza.
- Facilita el futuro mantenimiento y modificación del código.

Los siguientes apartados están inevitablemente relacionados entre si, ya que todas las técnicas o reglas persiguen el mismo fin.

Convenciones de escritura de Java

Las convenciones de código existen debido a que la mayoría del coste del código de un programa se usa en su mantenimiento, (casi ningún programa se mantiene toda su vida con el código original), y mejoran la lectura del código permitiendo entender código nuevo mucho más rápido y a fondo. En la web de *Sun Microsystems* se recogen dichas convenciones en una guía en [inglés](#), o una traducción al [castellano](#) cortesía de *javaHispano*. Por su parte Google también ha creado recientemente una [Guía de estilo](#) para Java.

Para que las convenciones funcionen, cada programador debe tratar de ser lo más fiel posible a estas.

Ficheros

Todos los ficheros fuente de java son ficheros de texto plano cuyo nombre termina con la extensión *.java* Dentro de cada fichero *.java* tenemos 4 partes en el siguiente orden:

1. Posibles comentarios sobre la clase (autor, fecha, licencias, etc)
2. Sentencia *package*. Toda clase debe estar en un paquete.
3. Sentencias *import*. Importar cada clase en una linea separada.
4. La definición de una única clase o interface **cuyo nombre es idéntico al nombre del fichero sin la extensión.**

```
/* Correcto */  
import java.awt.Frame;  
import java.awt.Graphics;
```

```
/* Incorrecto */  
import java.awt.*;
```

Posteriormente dentro de la definición de la clase, aplicamos el siguiente orden:

1. Sentencia *class* o *interface*
2. Variables de clase (static)
3. Variables de instancia (Atributos de la clase)
4. Constructores (Si hay sobrecarga deben ir seguidos)
5. Métodos (Si hay sobrecarga deben ir seguidos)

Declaraciones de variables

- Una sola declaración por línea.

```
int edad;  
int cantidad;
```

- Las variables locales se deben inicializar en el momento de declararlas o justo después. Se declaran justo antes de su uso, para reducir su ámbito.
- Las variables de instancia o de clase se declaran al comienzo de la definición de la clase.
- Los arrays se pueden inicializar en bloque:

```
int[] array =  
{  
    0, 1, 2, 3  
};  
// ó  
int[] array = { 0, 1, 2, 3 };
```

- Los arrays tienen los corchetes [] unidos a su tipo de datos:

```
String[] nombres; //correcto  
String nombres[]; //incorrecto
```

Nombres de identificadores

Para los identificadores podemos usar las letras anglosajonas y números de la tabla . No se debe usar caracteres con tilde ni la (ñ). Las barras bajas o guiones tampoco se usan. **Los nombres de los identificadores deben ser siempre lo más descriptivos posible, ya sea variable, método o clase.** Solo se usan identificadores de un solo carácter para representar los contadores del bucle for, y comienzan en la letra *i*.

- Nombre de **Package**: siempre en minúsculas.
- Nombre de las **clases o interfaces**: *UpperCamelCase*.
- Nombre de los **métodos**: *lowerCamelCase*. Suelen ser verbos o frases.
- Nombres de **constantes**: *CONSTANT_CASE*. Todo el mayúsculas, separando con barra baja.
- **Variables locales, atributos de la clase, nombres de parámetros**: *lowerCamelCase*.

Magic Numbers

Se conoce bajo este nombre a cualquier valor literal (“texto” o numérico) empleado en el código sin ninguna explicación. Se deben sustituir siempre que se pueda por una constante que identifique su finalidad.

```
//incorrecto
int precioConIva = precioBase + (0.21 * precioBase);

//correcto
//Se define en la clase
final static double IVA = 0.21;
//Se utiliza en un método
int precioConIva = precioBase + (IVA * precioBase);
```

Estructura del código

- Debemos usar la codificación UTF-8
- En las sentencias de control de flujo (if, else, for, do-while, try-catch-finally) se incluyen llaves { }, incluso si no contienen código o es una sola instrucción. Se alinean las llaves {} al inicio de línea.

```
if (final < indice) {
    filaInicial = indice - numeroFilas;
} else if (indice < filaInicial) {
    filaInicial = indice;
}
```

- Una sola instrucción por línea.
- Las líneas de código no deben superar los 100 caracteres. Si no, se deben **romper** antes de algún operador.
- Si la declaración del método es demasiado larga, o una expresión aritmética es demasiado larga, o en una sentencia *if*, debo romper.
- Si una operación aritmética o lógica se compone de distintos tipos de operaciones con distinta jerarquía, se deben usar paréntesis para facilitar su legibilidad.

```
public void ejecutarAccion(
    TipoParametro parametro1, TipoParametro parametro2, TipoParametro
    parametro3){
    ...
}

if ((condicion1 && condicion2)
    || (condicion3 && condicion4)
    || !(condicion5 && condicion6)) {
    llamarMetodo();
}

longName1 = longName2 * (longName3 + longName4 - longName5)
+ (4 * longname6);           //Siempre con el operador al principio de línea
```

- Los espacios en blanco mejoran la legibilidad. Se deben colocar entre operadores, después de los puntos y coma de los bucles for, después de los operadores de asignación, etc.

```
cantidadTotal = cantidadInicial + cantidadFinal;

for(int i = 0; i < cantidadTotal; i++){
    ...
}

public String getItem(int fila, int columna) {
    ...
}

getItem(cantidadInicial, cantidadFinal);
```

Debemos estar familiarizados y poner en práctica las convenciones recogidas en alguna de las guías de estilo indicadas.

Bad Smells

Se conoce como **Bad Smell o Code Smell** (mal olor) ¹⁾ a algunos indicadores o síntomas del código que posiblemente oculten un problema más profundo. Los *bad smells* no son errores de código, bugs, ya que no impiden que el programa funcione correctamente, pero son indicadores de fallos en el diseño del código que dificultan el posterior mantenimiento del mismo y aumentan el riesgo de errores futuros. Algunos de estos síntomas son:

- **Código duplicado** (*Duplicated code*). Si se detectan bloques de código iguales o muy parecidos en distintas partes del programa, se debe extraer creando un método para unificarlo.
- **Métodos muy largos** (*Long Method*). Los métodos de muchas líneas dificultan su comprensión. Un método largo probablemente está realizando distintas tareas, que se podrían dividir en otros métodos. Las funciones deben ser las más pequeñas posibles (3 líneas mejor que 15). Cuanto más corto es un método, más fácil es reutilizarlo. *Un método debe hacer solo una cosa, hacerla bien, y que sea la única que haga.*
- **Clases muy grandes** (*Large class*). Problema anterior aplicado a una clase. Una clase debe tener solo una finalidad. Si una clase se usa para distintos problemas tendremos clases con demasiados métodos, atributos e incluso instancias. Las clases deben el menor número de responsabilidades y que estén bien delimitadas.
- **Lista de parámetros extensa** (*Long parameter list*). Las funciones deben tener el mínimo número de parámetros posible, siendo 0 lo perfecto. Si un método requiere muchos parámetros puede que sea necesario crear una clase con esa cantidad de datos y pasarle un objeto de la clase como parámetro. Del mismo modo ocurre con el valor de retorno, si necesito devolver más de un dato.
- **Cambio divergente** (*Divergent change*). Si una clase necesita ser modificada a menudo y por razones muy distintas, puede que la clase esté realizando demasiadas tareas. Podría ser eliminada y/o dividida.
- **Cirugía a tiros** (*Shotgun surgery*). Si al modificar una clase, se necesitan modificar otras clases o elementos ajenos a ella para compatibilizar el cambio. Lo opuesto al *smell* anterior.
- **Envidia de funcionalidad** (*Feature Envy*). Ocurre cuando una clase usa más métodos de otra clase, o un método usa más datos de otra clase, que de la propia.
- **Legado rechazado** (*Refused bequest*). Cuando una subclase extiende (hereda) de otra clase, y utiliza pocas características de la superclase, puede que haya un error en la jerarquía de clases.

En la siguiente [página web](#) tenemos la mayoría de Bad Smells agrupados en 5 tipos.

Buenas prácticas

- **Manejo de Strings:** Los Strings son objetos, por lo que crearlos es costoso. Es mucho más rápido instanciarlos con una asignación, que con el operador *new*.
 - Concatenar String con el operador '+' también genera mucha carga, ya que crea un nuevo String en memoria (Los objetos String son [inmutables](#)). Se debe tratar de evitar siempre las concatenaciones (+) dentro de un **bucle**, o usar otras clases en ese caso (p.e. StringBuilder)

```
//instanciación lenta
String lenta = new String("objeto string");
```

```
//instanciación rápida
String rapida = "objeto string";
```

- **Tipos primitivos mejor que clases wrapper (envoltorio)** : Las clases wrapper al ser objetos, proveen de métodos para trabajar mejor con ellas, pero al igual que los Strings, son más lentos que los tipos primitivos.
- **Comparación de objetos:** Recordar que tanto los *Strings* como las tipos *Wrapper* son objetos y sus variables solo contienen sus referencias (direcciones). **Los objetos no se comparan con ==.**
- Evitar la creación innecesaria de objetos. Como se ha dicho, generan mucha carga.

```
int x = 10;
int y = 10;
```

```
Integer x1 = new Integer(10);
Integer y1 = new Integer(10);
```

```
String x2="hola";
String y2 = new String("hola");
```

```
System.out.println(x == y);    //TRUE
System.out.println(x1 == y1);  //FALSE, ya que son 2 objetos distintos
System.out.println(x2 == y2);  //FALSE, ya que son 2 objetos distintos
```

- **Visibilidad de atributos:** Los campos de una clase 'estándar' no deben declararse nunca como public, ni mucho menos no indicarle un modificador de visibilidad. Se usan sus *setters* y *getters* para su acceso.
- **Limitar siempre el alcance de una variable local.** Crear la variable local e inicializarla lo más cerca posible de su uso.
- **Usar siempre una variable para un único propósito.** A veces sentimos la tentación de reutilizar una variable, pero complica la legibilidad.

```
...
int resultadoTotal = resultadoInicial - resultadoFinal;
...
```

- **Bucle for.** Optar por el *for* siempre que se pueda (frente a while, do-while). Las ventajas son que reúne todo el control del bucle en la misma línea (inicio, fin, e incremento), y la variable de control ('i') no es accesible desde fuera de él. Si se necesita modificar su variable de control, usar otro bucle.

- **Constantes:** Cualquier valor literal debe ser definido como constante, excepto 1, -1, 0 ó 2 que son usados por el bucle for.
- **Switch:** Siempre debe llevar un *break* despues de cada caso, y tambien el caso *default* que ayudará a corregir futuros aumentos del número de casos.
- El *copiado defensivo* es salvador. Cuando creamos un constructor que recibe el mismo tipo de objeto de la clase, debemos tener cuidado y crear un nuevo objeto a partir del recibido.

Introducción al refactoring con IntelliJ IDEA (Android Studio)

Introduction

IntelliJ IDEA tiene muchas capacidades de refactorización automática, pero como desarrolladores no es suficiente saber cómo realizarlas, necesitamos entender cuáles son estas refactorizaciones, cuándo queríamos aplicarlas y cualquier posible inconveniente o cosas a considerar antes de usarlas.

La refactorización, como la define Martin Fowler, "... es una técnica disciplinada para reestructurar un cuerpo de código existente, alterando su estructura interna sin cambiar su comportamiento externo". Por lo tanto, es importante antes de realizar cualquier refactorización en el código de producción tener una cobertura de prueba completa para demostrar que no ha cambiado el comportamiento sin darse cuenta.

El objetivo de este tutorial es presentar a aquellos que puedan ser nuevos en la idea de refactorización, en particular la refactorización automática, las capacidades de IntelliJ IDEA y mostrar cuándo es posible que desee aplicar tres de los tipos básicos de refactorización: cambio de nombre, extracción y eliminación.

Renombrado

El cambio de nombre puede parecer una refactorización trivial, pero el cambio de nombre mediante una simple búsqueda y reemplazo a menudo significa que los elementos no relacionados con el mismo nombre se cambian involuntariamente. El uso de refactorizaciones de cambio de nombre de IntelliJ IDEA minimiza estos errores.

¿Por qué renombrar?

1. El nombre no es lo suficientemente descriptivo
2. El nombre de la clase / método / variable no coincide con lo que realmente es
3. Se ha introducido algo nuevo, que requiere que el código existente tenga un nombre más específico

Cambiar el nombre a medida que codifica

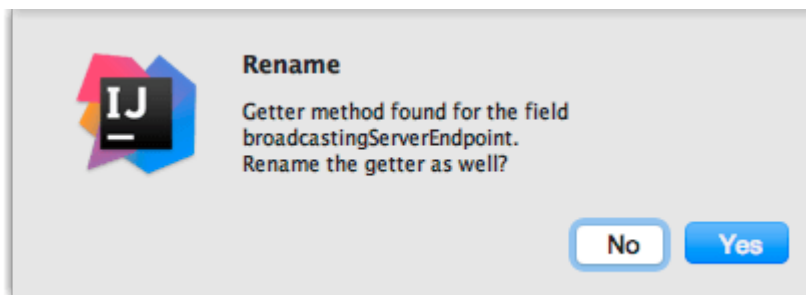
Imagina que te encuentras con el siguiente código mientras implementas alguna función o arreglas algún error:

```
server = new Server(path, port, endpoint);
server.init();
server.run();
```

Supongamos que queremos:

- Cambiar el nombre de `endpoint`, un parámetro, para describir qué tipo de “punto final” es.
 - Renombrar `init()`, un método de `Server`, para que el nombre sea más descriptivo de lo que realiza el método.
 - Renombrar `Server`, una clase, a algo más específico.
1. Para renombrar el parámetro `endpoint`, coloca tu cursor sobre la palabra `endpoint` y pulsa **Shift+F6**. IntelliJ IDEA mostrará mediante un menú pop-up una serie de propuestas, basadas en el nombre de la clase y en otros aspectos del código. En este caso, también se sugiere incluso el propio nombre del parámetro.

Seleccione una de las propuestas o escriba la suya propia. Si el parámetro tiene un “getter”, IntelliJ IDEA preguntará si queremos cambiarlo también.



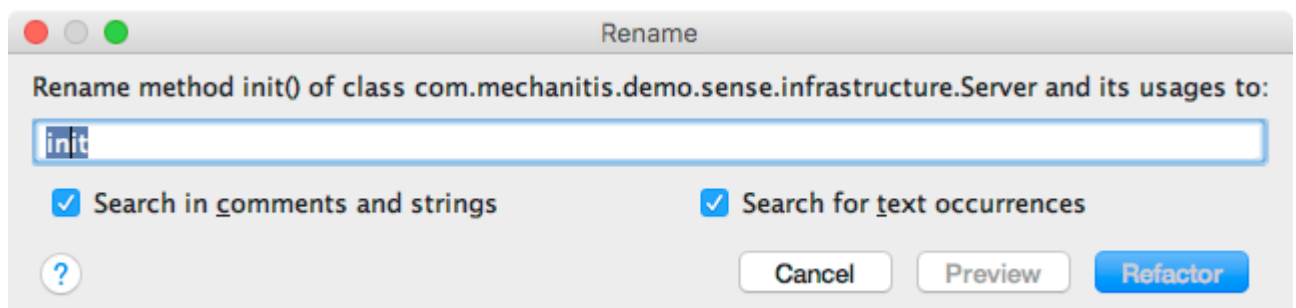
2. Se habrá dado cuenta que todos los usos de este campo se cambian al nuevo nombre, y si ha elegido cambiar el nombre del getter, otras clases en su proyecto se actualizarán para usar el nuevo nombre. Consulte el siguiente paso para obtener más información sobre el cambio de nombre de métodos.
3. Para cambiar el nombre del método, el proceso es el mismo: coloque el cursor en `init` y presione **Shift+F6**. Aquí tendrás menos sugerencias, así que escriba el nuevo nombre:

```

@Override
public void run() {
    try {
        server = new Server(path, port, broadcastingServerEndpoint);
        server.validate();
        server No suggestions
    } catch (E Press ⌘F6 to show dialog with more options
        LOGGER.severe(e.getMessage());
    }
}

```

- Además de cambiar el nombre del método, esto cambia el nombre de todas las llamadas del método y todos los métodos reemplazados / implementados en las subclases. IntelliJ IDEA también puede cambiar el nombre de los usos del nombre que no son de código, lo cual es útil si tiene una configuración XML u otros archivos que no son de Java que se refieren a clases o métodos. Puede configurar qué cambia de nombre si presiona Shift+F6 por segunda vez para abrir el cuadro de diálogo de cambio de nombre.



Si el cambio de nombre se va a aplicar a algo más que al código fuente, IntelliJ IDEA obtendrá una vista previa de la refactorización para que pueda seleccionar los cambios que desea realizar. A menudo, en estos casos, puede optar por no cambiar el nombre de las ocurrencias en los comentarios, especialmente si el nombre del método original era una palabra común como nombre.



Si no desea realizar algunos de estos cambios, presione la tecla **Backspace** en los usos que no desea cambiar.

5. Cambiar el nombre de una clase es similar, pero también se puede realizar a través de la ventana de herramientas Proyecto. En este caso, debido a que hemos descubierto que queremos cambiar el nombre de la clase donde la usamos, usaremos **Shift+F6** en el nombre de la clase en el código.

```
@Override
public void run() {
    try {
        server = new WebSocketServer(path, port, broadcastingServerEndpoint);
        server.valid
        server.run()
    } catch (Exception e) {
        LOGGER.severe(e.getMessage());
    }
}
```

Press ⌘F6 to show dialog with more options

Por supuesto, también se cambiará el nombre de cualquier código que use esta clase, pero también tiene la opción de cambiar el nombre de las variables, los herederos y otras partes del código para que estén alineados con el nuevo nombre. Nuevamente, estas opciones se pueden configurar presionando **Shift+F6** por segunda vez.

Impacto del renombrado

El cambio de nombre de las variables locales o los métodos privados se puede realizar de forma bastante segura sobre la marcha. Por ejemplo, mientras trabaja en una función que toca esta área del código, puede realizar esta refactorización sabiendo que el impacto tiene un alcance limitado.

Cambiar el nombre de las clases o los métodos públicos podría afectar a muchos archivos. Si este es el caso, este tipo de refactorización debería, como mínimo, estar en su propia confirmación separada para que los cambios estén claramente separados de cualquier funcionalidad modificada o adicional en la que pueda haber estado trabajando en ese momento (con respecto al control de versiones).

Extracciones

Las refactorizaciones de extracción de IntelliJ IDEA brindan a los desarrolladores el poder de remodelar su código cuando queda claro que el diseño actual, ya sea a pequeña o gran escala, ya no es adecuado para su propósito.

Extracción de una variable

Extraer variable es un cambio de bajo impacto para hacer que su código se auto-documente. También se puede utilizar para reducir la duplicación de códigos.

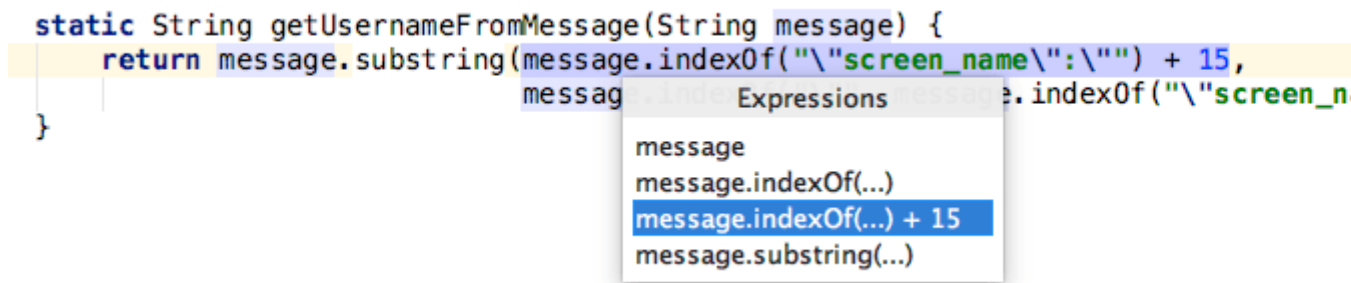
Imagina que te encuentras con el siguiente código:

```
static String getUsernameFromMessage(String message) {
    return message.substring(message.indexOf("\"screen_name\":") + 15,
        message.indexOf("\"", message.indexOf("\"screen_name\":") + 15));
}
```

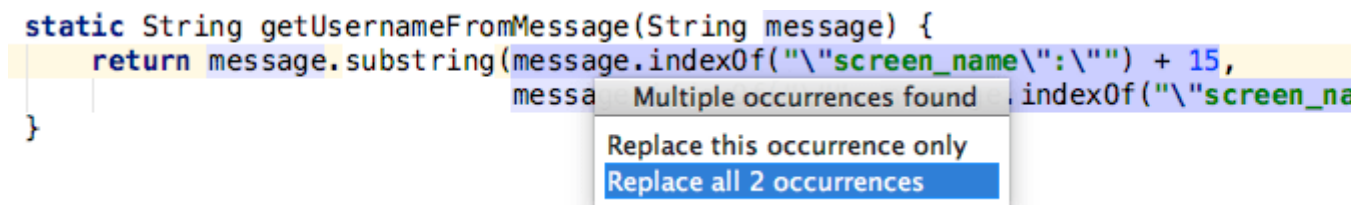
}

Podemos usar una extracción para mejorar este código así:

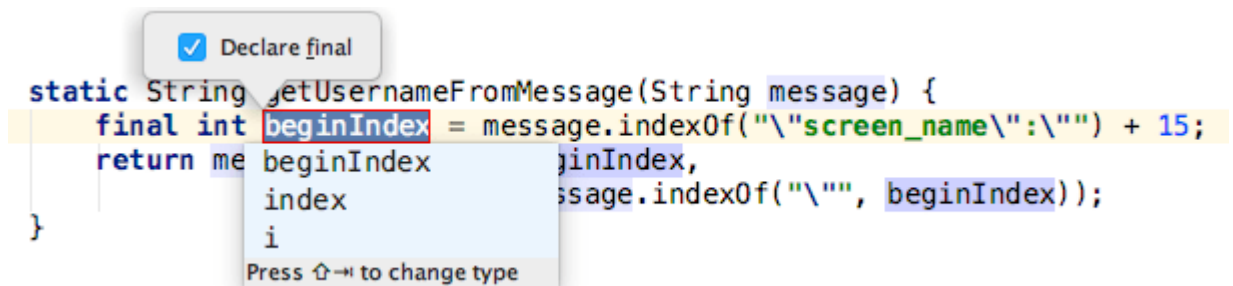
- El código `message.indexOf("\screen_name\":"") + 15` está duplicado, podemos eliminar ese uso doble.
 - Introduciendo variables para describir lo que representa cada una de las llamadas `indexOf`
 - Borrando el número literal (número mágico) 15.
1. Primero, reduzcamos la duplicación e introduzcamos una variable que describa lo que está haciendo esta operación. Coloque el cursor en cualquier lugar de la expresión `message.indexOf("\screen_name\":"") + 15` y presione `Ctrl + Alt + V`. IntelliJ IDEA le sugerirá un contexto para esta refactorización, y desea elegir el que encapsula esta expresión:



A continuación, si IntelliJ IDEA ha detectado que esta expresión ocurre más de una vez, tiene la opción de reemplazar todas las ocurrencias o solo la que seleccionó.



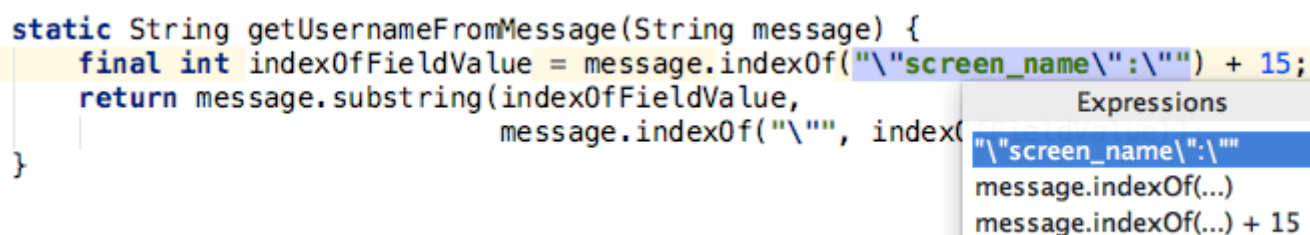
Una vez que se extrae la variable, IntelliJ IDEA sugiere posibles nombres basados en cosas como el parámetro en el que se usó la expresión..



Usaremos nuestro propio nombre, `indexOfFieldValue`, para describir lo que esto realmente representa. Tenga en cuenta que puede decidir si desea o no que esta variable sea “final”.

A continuación, vamos a introducir una variable para el valor de cadena. Hay dos razones para esto: en primer lugar, para documentar lo que representa el valor String y, en segundo lugar, porque nos ayudará a eliminar el número mágico.

Posiciona el cursor encima del literal `screen_name` y presiona `Ctrl+Alt+V`.



```
static String getUsernameFromMessage(String message) {
    final int indexOffieldValue = message.indexOf("screen_name\\":\\") + 15;
    return message.substring(indexOffieldValue,
        message.indexOf("\\", indexOffieldValue));
}
```

Expressions

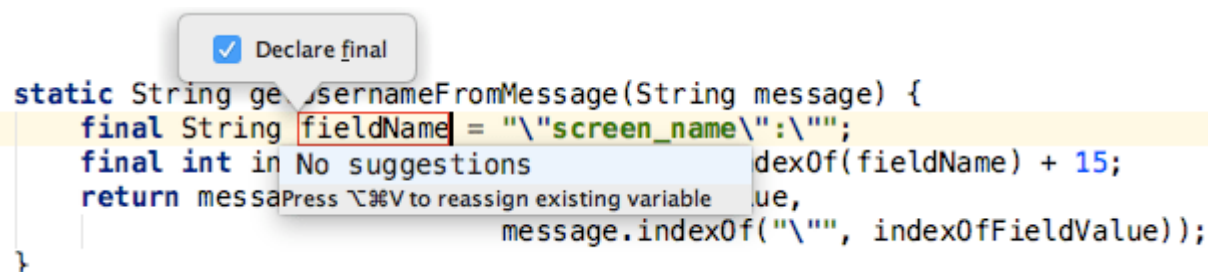
- "screen_name\\":\\"
- message.indexOf(...)
- message.indexOf(...) + 15

Vamos a proporcionarle un nombre con mejor significado, `fieldName`.



```
@Override
public void run() {
    try {
        server = new Server(path, port, endpoint);
        server.init();
        server.run();
    } catch (Exception e) {
        LOGGER.severe(e.getMessage());
    }
}
```

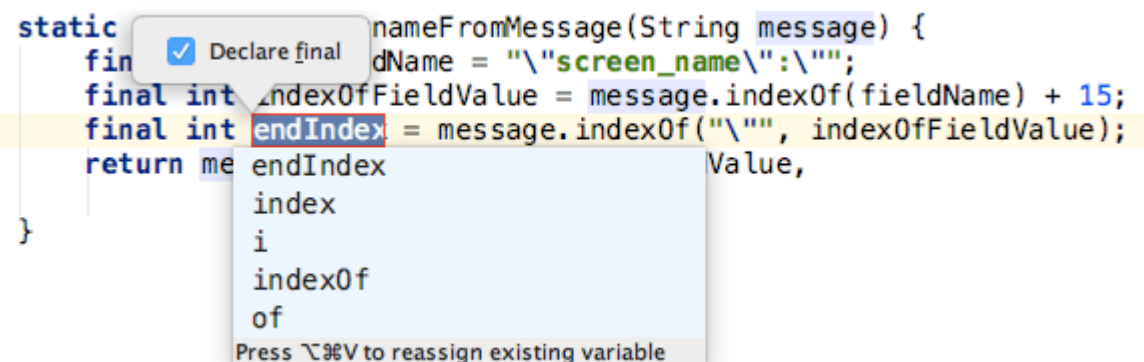
serviceEndpoint
stringBroadcastingServerEndpoint
broadcastingServerEndpoint
serverEndpoint
endpoint
Press `⇧F6` to show dialog with more options



```
static String getUsernameFromMessage(String message) {
    final String fieldName = "screen_name\\":\\";
    final int indexOffieldValue = message.indexOf(fieldName) + 15;
    return message.substring(indexOffieldValue,
        message.indexOf("\\", indexOffieldValue));
}
```

✓ Declare final
No suggestions
Press `⌘%V` to reassign existing variable

- Ahora, vamos a crear una variable para la otra expresión usada como parámetro de `substring()`, usando el mismo proceso, y lo llamaremos `indexOfEndOffieldValue`.



```
static String getUsernameFromMessage(String message) {
    final String fieldName = "screen_name\\":\\";
    final int indexOffieldValue = message.indexOf(fieldName) + 15;
    final int endIndex = message.indexOf("\\", indexOffieldValue);
    return message.substring(indexOffieldValue,
        message.indexOfEndOffieldValue());
}
```

✓ Declare final
index
i
indexOf
of
Press `⌘%V` to reassign existing variable

- Finalmente, podemos eliminar el número mágico, ya que es solo la longitud del nombre del campo. El código final se ve así:

```
static String getUsernameFromMessage(String message) {
    final String fieldName = "\"screen_name\":\":";
    final int indexOfFieldValue = message.indexOf(fieldName) +
                                fieldName.length();
    final int indexOfEndOfFieldValue = message.indexOf("\"",
                                indexOfFieldValue);
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);
}
```

4. Es más largo que el original, pero es mucho más descriptivo, lo cual es particularmente importante en un código como este, donde no está claro qué representa cada expresión. La elección de aplicar `final` o no depende de usted, y depende de sus estándares de codificación.

Extracción de un parámetro

Extraer o agregar un parámetro permite a un desarrollador cambiar un método para que sea más fácil de usar. Es posible que desee cambiar los parámetros, por ejemplo, pasando algunos valores de un objeto en lugar del objeto en sí, o puede que desee introducir un valor del cuerpo del método como parámetro para permitir que el método se utilice en más lugares. Veremos un ejemplo de esto último.

Para este ejemplo, usaremos el mismo código que el último ejemplo, después de que se haya refactorizado, y lo ampliaremos ligeramente para mostrar otro método en la misma clase:

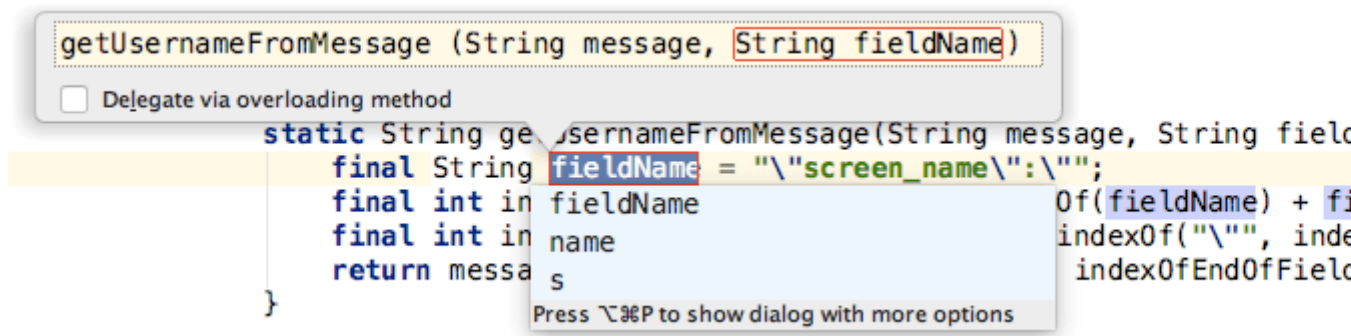
```
static String getTextFromMessage(String message) {
    final String fieldName = "\"text\":\":";
    final int indexOfFieldValue = message.indexOf(fieldName) + fieldName.length();
    final int indexOfEndOfFieldValue = message.indexOf("\"", indexOfFieldValue);
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);
}

static String getUsernameFromMessage(String message) {
    final String fieldName = "\"screen_name\":\":";
    final int indexOfFieldValue = message.indexOf(fieldName) + fieldName.length();
    final int indexOfEndOfFieldValue = message.indexOf("\"", indexOfFieldValue);
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);
}
```

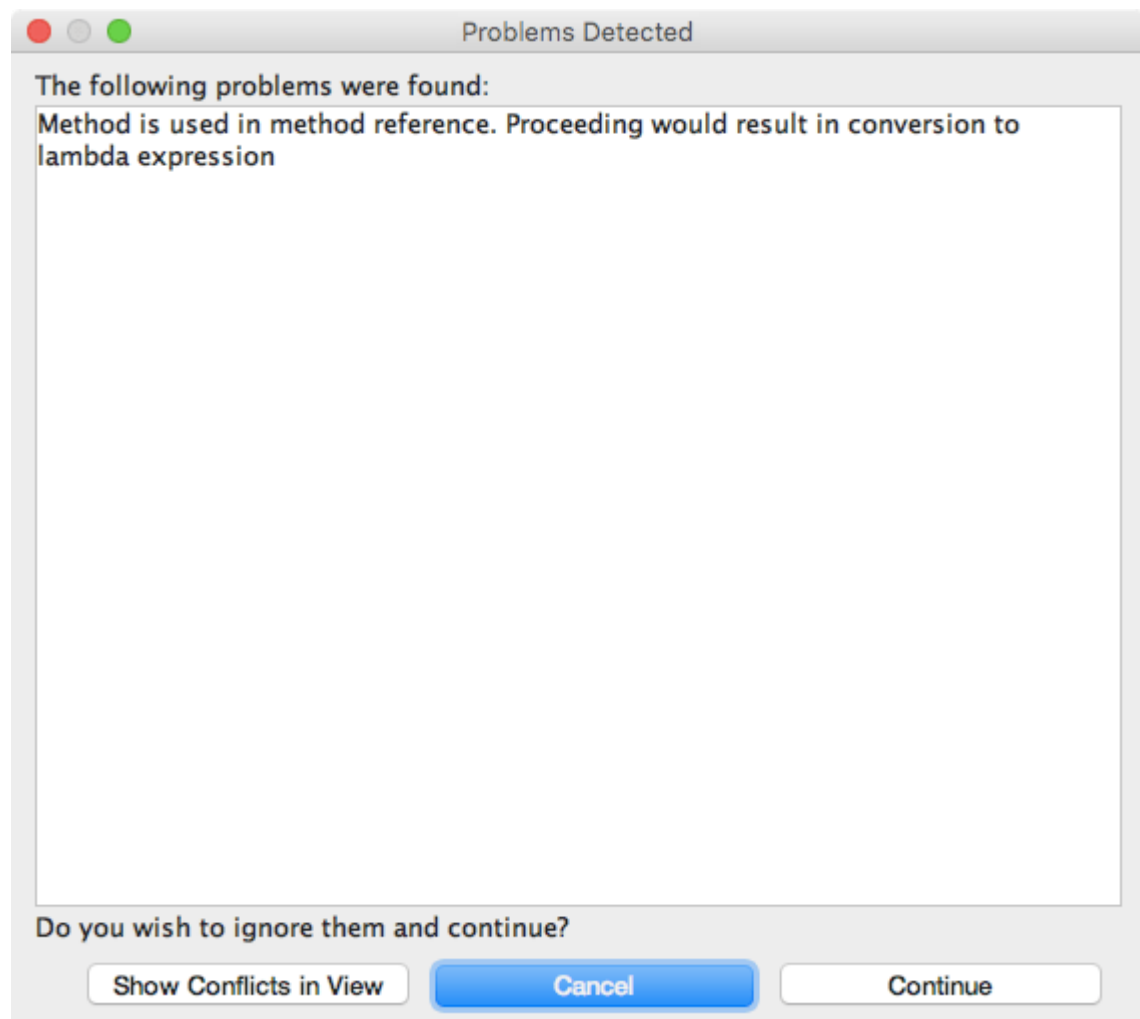
Nuestro objetivo es eliminar la duplicación de código que vemos en estos dos métodos. Para hacer eso, vamos a:

- Cambiar `fieldName` para que sea un parámetro y así conseguir que el método `getUsernameFromMessage` sea aplicable a cualquier campo.
- Renombrar `getUsernameFromMessage` a algo que represente su sentido más general.
- Eliminar el código duplicado en el método `getTextFromMessage`.

1. Coloca el cursor sobre `fieldName` y presiona `Ctrl+Alt+P`



Al igual que con las otras refactorizaciones, puede escribir un nuevo nombre para el parámetro si lo desea. IntelliJ IDEA también muestra una vista previa de la firma del método actualizado. Presione Enter para aprobar los cambios.



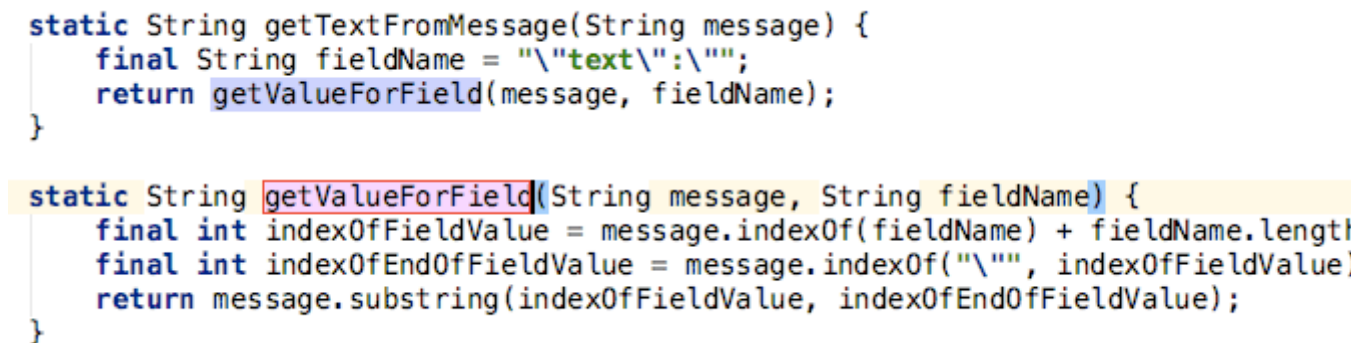
Este problema en particular nos dice que el método se está utilizando como una referencia de método, y este cambio dará como resultado que la referencia del método se convierta en una expresión lambda. Este mensaje podría ser una señal de que esta no es la refactorización que desea realizar. Si este es el caso, el siguiente ejemplo muestra un enfoque que podríamos tomar usando el método de extracción. Sin embargo, para este ejemplo asumiremos que estamos contentos con las consecuencias de introducir un nuevo parámetro, por lo que solo seleccionaremos Continuar.

A continuación, IntelliJ IDEA detectará cualquier código que ahora se pueda reemplazar con una llamada a la nueva nomenclatura del método.



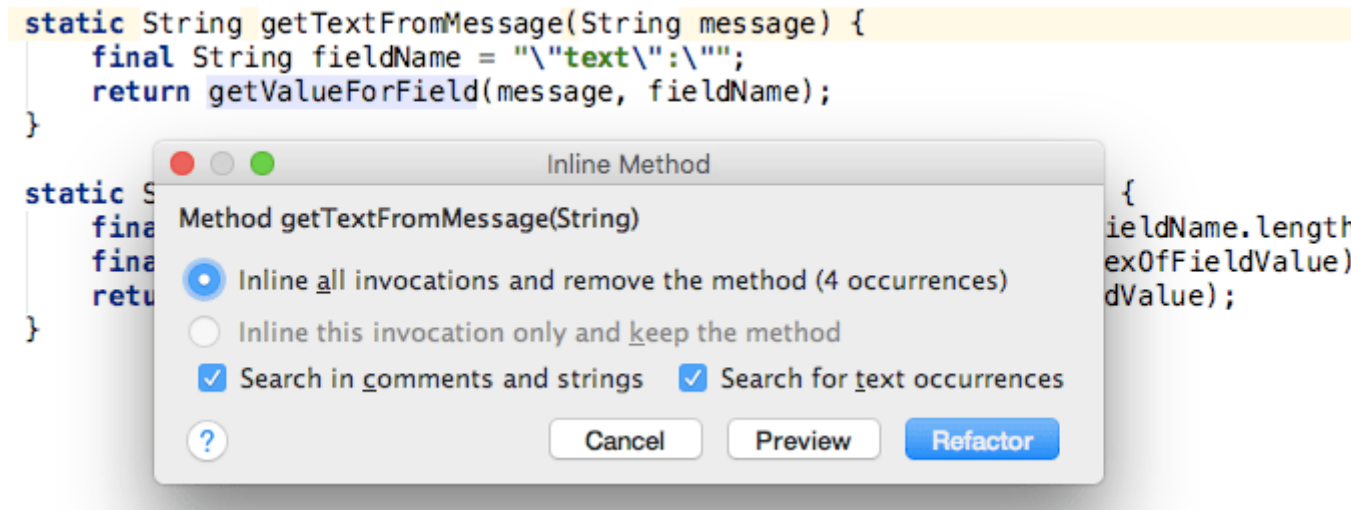
Si selecciona Reemplazar en este caso, todo el código duplicado será reemplazado e IntelliJ IDEA seleccionará el valor apropiado para pasar al nuevo parámetro.

2. En este punto, el método original getUsernameFromMessage es más general de lo que era, por lo que deberíamos cambiarle el nombre. Colocamos nuestro cursor sobre el nombre y usamos Shift+F6, como se muestra en la sección anterior.



3. Podemos simplificar aún más el código. Inline es lo contrario de extraer, y en el código que tenemos aquí puede ser apropiado incluir nuestra variable temporal inline, ya que el nombre de la variable nos da poco más que tener el valor pasado directamente al método. O, dado que getTextFromMessage realmente es una delegación simple a getValueForField, podemos usar inline para eliminar este método por completo.

Para usar inline, pase con el cursor por encima de getTextFromMessage y presione Ctrl+Alt+N



Nuestro código ahora tendrá este aspecto:

```
static String getValueForField(String message, String fieldName) {
    final int indexOfFieldValue = message.indexOf(fieldName) +
                                fieldName.length();
    final int indexOfEndOfFieldValue = message.indexOf("\"",
                                indexOfFieldValue);
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);
}
```

4. Nuestro código que llamaba al método getUsernameFromMessage original era:

```
Parser::getUsernameFromMessage
```

y es ahora

```
(message) -> Parser.getValueForField(message,
                                "\"screen_name\":\")
```

Nuestro código que llamaba al método getTextFromMessage original era:

```
String[] wordsInMessage =
Parser.getTextFromMessage(message).split("\\s");
```

y es ahora

```
String[] wordsInMessage = Parser.getValueForField(message,
 "\"text\":\").split("\\s");
```

Tenga en cuenta que la forma en que aplicamos esta refactorización obliga a todas las personas que llaman a pasar el nombre del campo y a) distribuye el uso de un valor de cadena alrededor de su código y b) puede introducir la duplicación de uno o más de estos valores de cadena. Esto puede ser apropiado para su código, especialmente si se trata la duplicación de cadenas o el método no se usa con frecuencia. Sin embargo, si esto no es una compensación que desea hacer para reducir la duplicación de código, consulte el siguiente capítulo para obtener un enfoque alternativo.

Extracción de método

Una forma de ayudar a la legibilidad del código es tenerlo en secciones pequeñas y comprensibles. El método de extracción permite que un desarrollador haga precisamente eso, moviendo segmentos de código a su propio método, con un nombre descriptivo, cuando sea apropiado.

Algunos desarrolladores pueden encontrarse escribiendo métodos largos que realizan la operación que tienen en mente, y cuando hayan completado (y probado) la funcionalidad, miren el código para ver dónde se puede refactorizar y simplificar y dividir estos métodos más largos. O, cuando un desarrollador encuentra código cuando está implementando una nueva función, se da cuenta de que extraer algo de código en su propio método les permite reutilizar la funcionalidad existente.

Consejo

Cuando IntelliJ IDEA detecta código duplicado, este es un muy buen candidato para crear un nuevo método al que pueden llamar todos los lugares que tienen el código duplicado.

Vamos a ver el mismo ejemplo que en la sección anterior, pero adoptamos un enfoque ligeramente diferente al anterior.

```
static String getTextFromMessage(String message) {
    final String fieldName = "\"text\\":\\"";
    final int indexOffFieldValue = message.indexOf(fieldName) +
    fieldName.length();
    final int indexOffEndOffFieldValue = message.indexOf("\\"", indexOffFieldValue);
    return message.substring(indexOffFieldValue, indexOffEndOffFieldValue);
}

static String getUsernameFromMessage(String message) {
    final String fieldName = "\"screen_name\\":\\"";
    final int indexOffFieldValue = message.indexOf(fieldName) +
    fieldName.length();
    final int indexOffEndOffFieldValue = message.indexOf("\\"", indexOffFieldValue);
    return message.substring(indexOffFieldValue, indexOffEndOffFieldValue);
}
```

Como vimos anteriormente, la refactorización anterior tenía algunas ventajas y desventajas: era necesario convertir una referencia de método en una expresión lambda, y todo el código de llamada necesitaba saber el nombre de campo requerido. Podemos optar por eliminar la duplicación de código entre los dos métodos de una manera diferente:

- Extrayendo el código común a su propio método.
 - “Inlining” variables para simplificar el código restante.
1. Primero, resalta el código común entre los dos métodos:

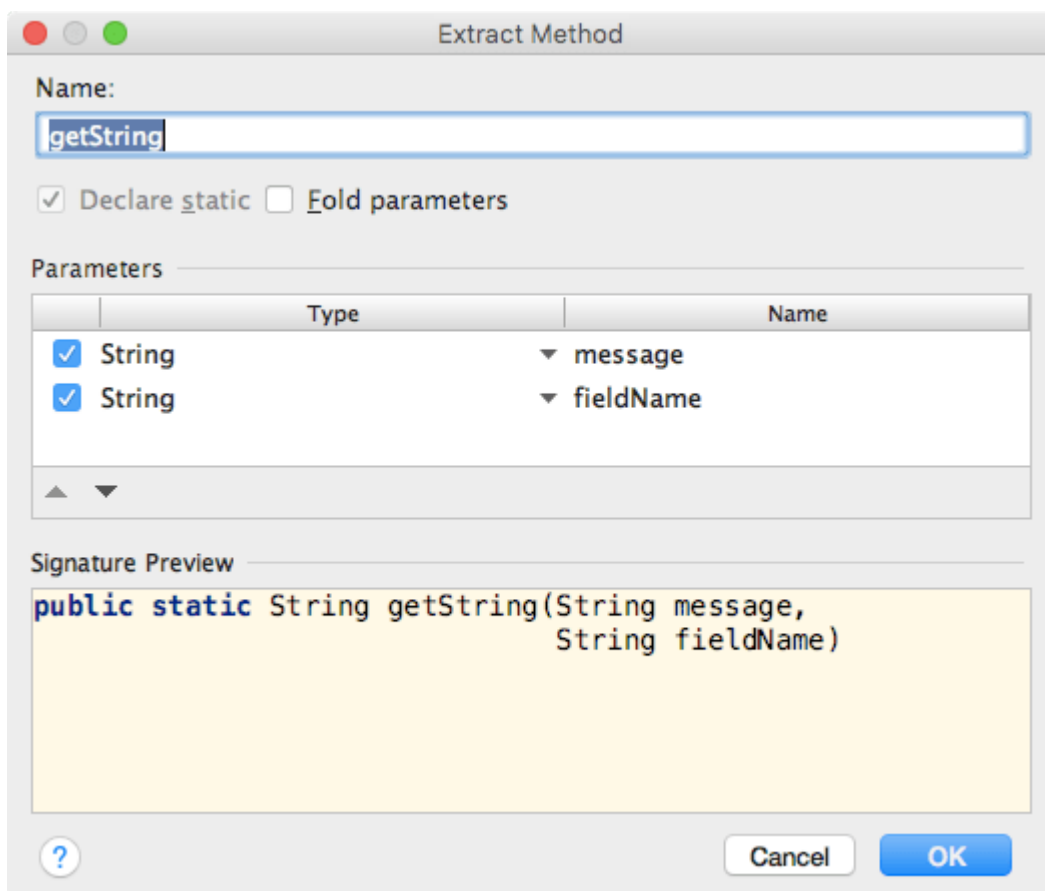

```

static String getTextFromMessage(String message) {
    final String fieldName = "\"text\":";
    final int indexOfFieldValue = message.indexOf(fieldName) + fieldName.length();
    final int indexOfEndOfFieldValue = message.indexOf("\"", indexOfFieldValue);
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);
}

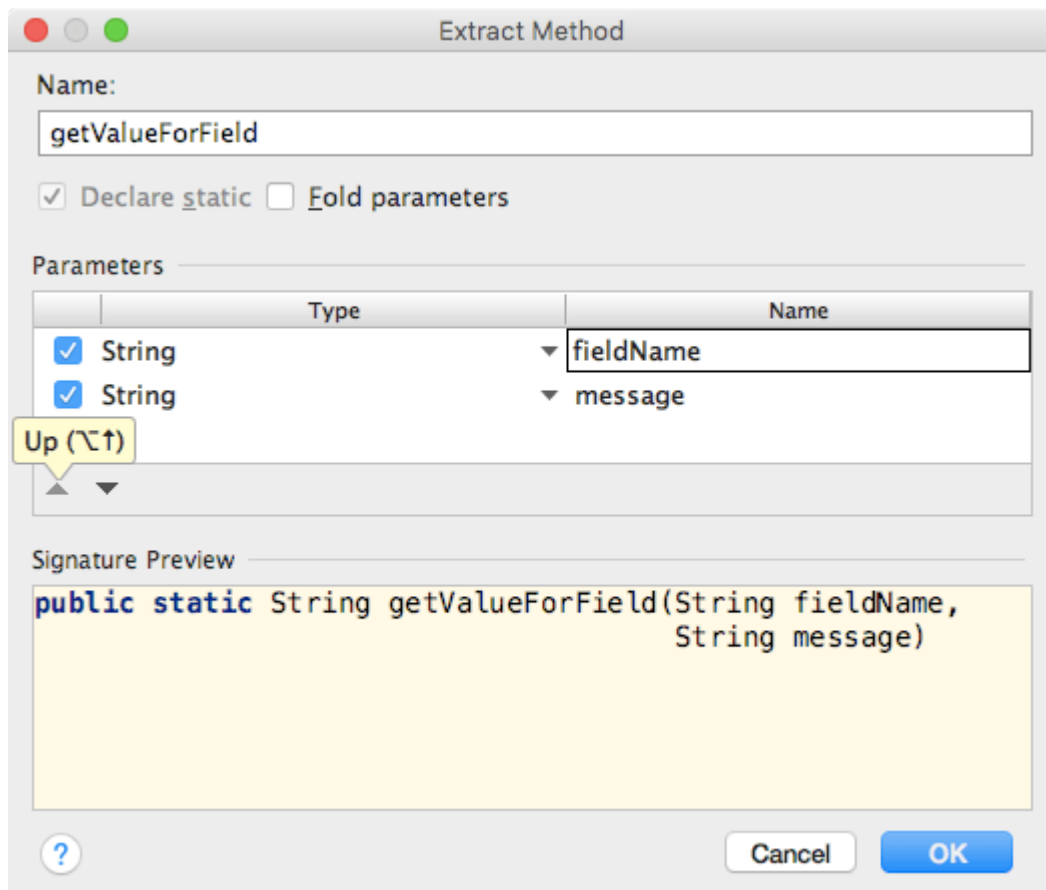
static String getUsernameFromMessage(String message) {
    final String fieldName = "\"screen_name\":";
    final int indexOfFieldValue = message.indexOf(fieldName) + fieldName.length();
    final int indexOfEndOfFieldValue = message.indexOf("\"", indexOfFieldValue);
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);
}

```

Presionando Ctrl+Alt+M abrirás el diálogo de “Extracción de método”.



Escriba el nombre del nuevo método, `getValueForField`, y verifique los nombres y el orden de los parámetros. En este caso, vamos a cambiar el orden de los parámetros porque preferimos que el parámetro `fieldName` esté más cerca del nombre del método. Esto dependerá de su estilo de código y de las preferencias del equipo, es posible que desee leer el nombre y los parámetros en voz alta para ver si tiene sentido como una declaración en lenguaje natural.

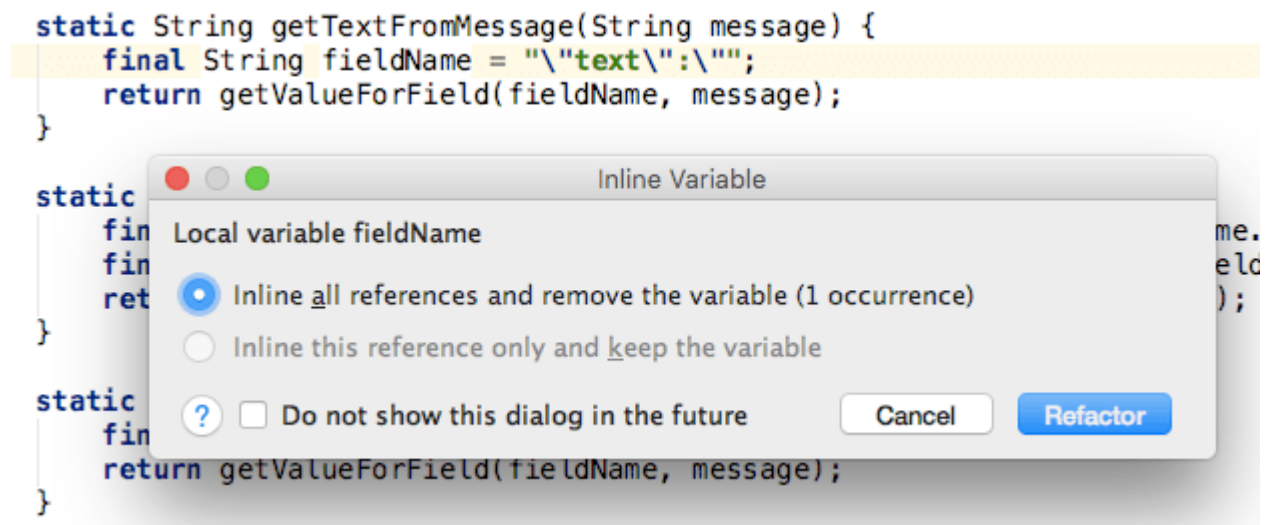


Cuando presiona Aceptar, IntelliJ IDEA detectará el código que se puede reemplazar con una llamada a este nuevo método y ofrecerá refactorizarlo también. Vamos a seleccionar Sí.



- En este punto, nuestros métodos `getTextFromMessage` y `getUsernameFromMessage` son dos líneas de código simple, y aquí tiene sentido incluir la variable `fieldName`, ya que el nombre del método es lo suficientemente descriptivo

como para eliminar la variable temporal. Presione Ctrl+Alt+N en `fieldName` y seleccione Refactor.



3. Como último toque, es posible que desee agrupar todos los métodos similares. Dependiendo de su configuración, IntelliJ IDEA puede haber colocado el nuevo método directamente debajo del método en el que estaba cuando eligió extraer el método, como en nuestro caso aquí. Para poner los métodos auxiliares uno al lado del otro, coloque el cursor en el nombre del método `getValueForField` y presione Ctrl + Shift + Abajo. Esto colocará su nuevo método, `getValueForField`, bajo el método `getUsernameFromMessage` existente.

Nuestro código final se ve así:

```
static String getTextFromMessage(String message) {
    return getValueForField("\"text\":\"", message);
}

static String getUsernameFromMessage(String message) {
    return getValueForField("\"screen_name\":\"", message);
}

static String getValueForField(String fieldName, String message) {
    final int indexOfFieldValue = message.indexOf(fieldName) +
    fieldName.length();
    final int indexOfEndOfFieldValue = message.indexOf("\"",
    indexOfFieldValue);
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);
}
```

3. Ahora tenemos dos métodos de ayuda muy específicos que obtienen el cuerpo del mensaje y el nombre de usuario, y un método más general que puede usarse para obtener el valor de cualquier campo del mensaje. Se pueden agregar métodos de ayuda adicionales cuando hay otros campos que se necesitan con frecuencia.

Tenga en cuenta que los ejemplos de Extraer parámetro y Extraer método comienzan con el mismo código, pero terminan con un código que se ve muy diferente. Esto no se debe solo a que usamos una refactorización diferente, sino a que tomamos decisiones diferentes; en el primer ejemplo, elegimos eliminar la duplicación por completo y mover parte de la toma de decisiones al llamador del método. En el segundo ejemplo, elegimos proporcionar una API que ocultaba los detalles del nombre del campo detrás de los pequeños métodos auxiliares, pero que también proporcionaba el

método más general. También podríamos haber mezclado y combinado los enfoques, la refactorización que elegimos para comenzar puede habernos llevado en una dirección particular, pero podemos dictar nuestro destino final. Debemos recordar el objetivo de nuestra refactorización (en este caso, reducir la duplicación) y comprender las compensaciones que hacemos cuando elegimos una dirección sobre otra, por ejemplo, decidir si queremos o no que el código de llamada sepa qué nombre de campo están preguntando.

Impacto de las extracciones

La buena noticia es que puede deshacer un extracto con bastante facilidad. No solo seleccionando Ctrl + Z, por supuesto, sino insertando el método creado para que el código vuelva a estar donde solía estar.

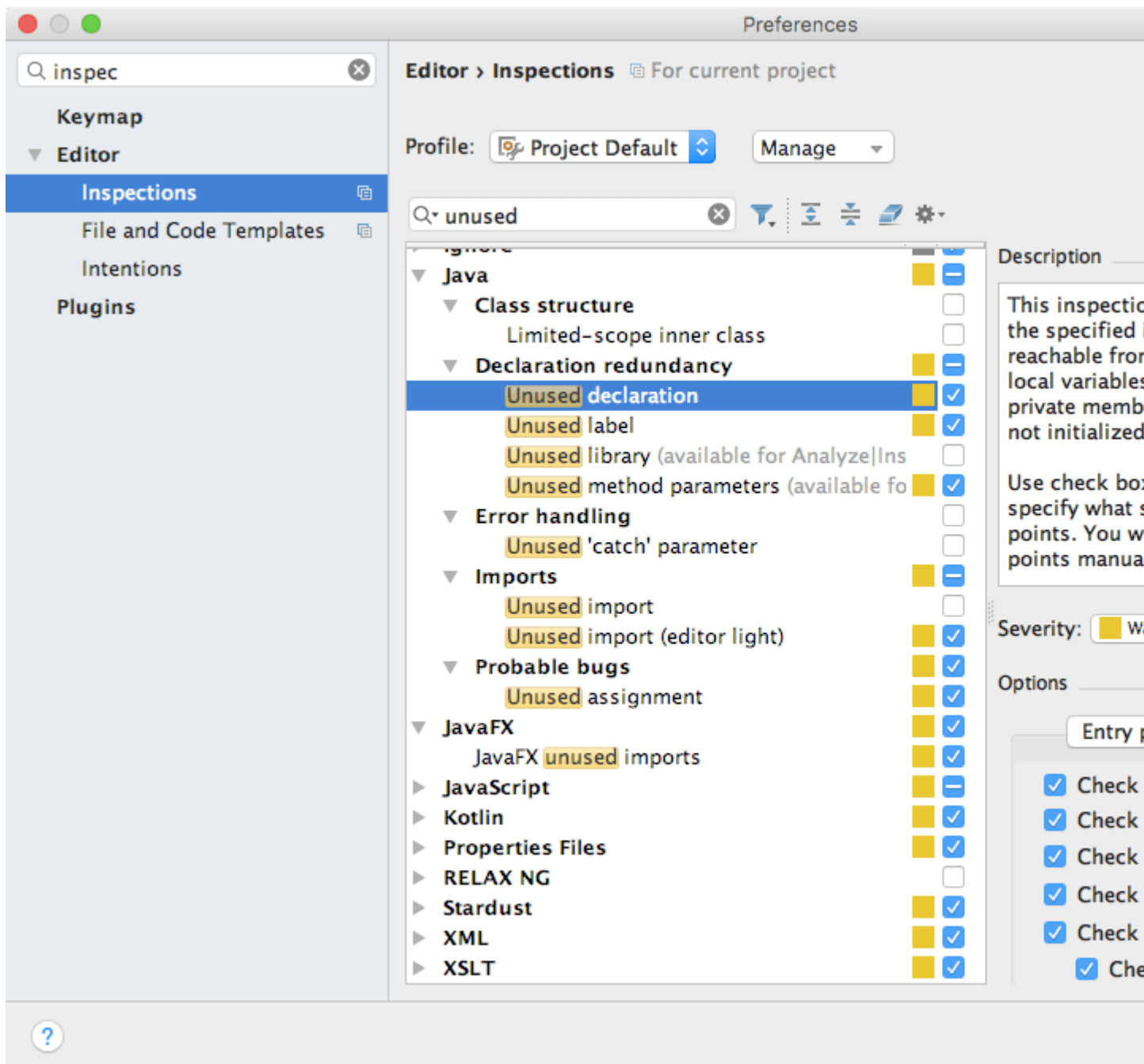
Los desarrolladores experimentados usan regularmente las refactorizaciones de extractos que hemos mencionado aquí para dar forma al código a medida que evoluciona, y no sería raro usarlas en mayor o menor medida cada vez que se toca el código. Algunos de los que no se trataron, como la interfaz de extracción y la superclase de extracción, pueden tener un impacto más amplio en el diseño general, y se debe tener más cuidado con ellos.

Borrado (Deleting)

A veces, cuando ha refactorizado el código en varios pasos, puede terminar con un código que ya no se usa o, idealmente, no debería usarse. Dado que el objetivo de la refactorización es la simplificación, siempre debe intentar eliminar el código no utilizado donde pueda, independientemente del impacto (o no) que tenga el código no utilizado en el rendimiento de su aplicación, el código no utilizado definitivamente afecta a los desarrolladores que trabajan y tratan para entender la aplicación.

Borrado seguro (Safe delete)

IntelliJ IDEA le permite eliminar de forma segura fragmentos de código no utilizados o archivos completos, informándole si es seguro eliminar el código y brindándole la opción de obtener una vista previa de los cambios antes de realizarlos. La forma más rápida de identificar y lidiar con el código no utilizado es asegurarse de que las inspecciones relevantes estén habilitadas, que generalmente están por defecto:



Continuemos con el ejemplo de nuestra refactorización anterior. Asumiendo que terminamos con este código:

```
static String getUsernameFromMessage(String message) {
    return getValueForField("\"screen_name\":\"", message);
}

static String getValueForField(String fieldName, String message) {
    final int indexOfFieldValue = message.indexOf(fieldName) +
    fieldName.length();
    final int indexOfEndOfFieldValue = message.indexOf("\"", indexOfFieldValue);
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);
}
```

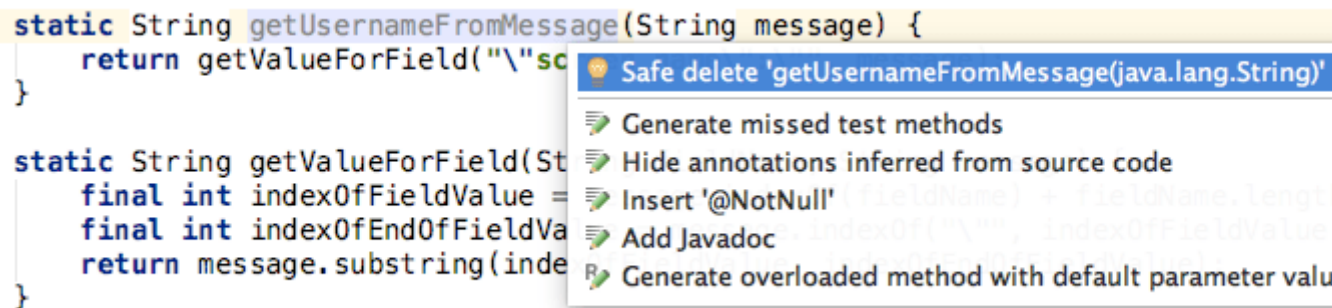
Es posible que algún tiempo después, cuando volvamos a este código, el método `getUsernameFromMessage` ya no se use, tal vez ya no sea necesario, o tal vez la gente se sienta

cómoda llamando a `getValueForField` con el parámetro relevante. Asumiendo que nos sentimos cómodos con estas razones, podemos continuar y eliminar este método..

1. Si la inspección de la declaración no utilizada está activada, el nombre del método aparecerá en gris para indicar que no se utiliza.

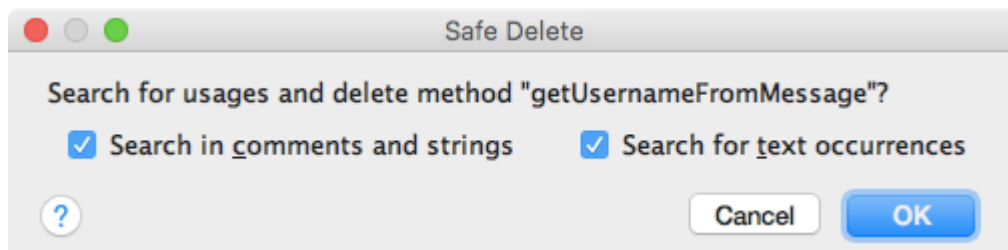
```
static String getUsernameFromMessage(String message) {  
    return getValueForField("\"screen_name\":\"", message);  
}  
  
static String getValueForField(String fieldName, String message) {  
    final int indexOfFieldValue = message.indexOf(fieldName) + fieldName.length();  
    final int indexOfEndOfFieldValue = message.indexOf("\"", indexOfFieldValue);  
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);  
}
```

2. Coloque el cursor en `getUsernameFromMessage` y presione `Alt+Enter`. Esto le dará la opción de borrar el método.



```
static String getUsernameFromMessage(String message) {  
    return getValueForField("\"screen_name\":\"", message);  
}  
  
static String getValueForField(String fieldName, String message) {  
    final int indexOfFieldValue = message.indexOf(fieldName) + fieldName.length();  
    final int indexOfEndOfFieldValue = message.indexOf("\"", indexOfFieldValue);  
    return message.substring(indexOfFieldValue, indexOfEndOfFieldValue);  
}
```

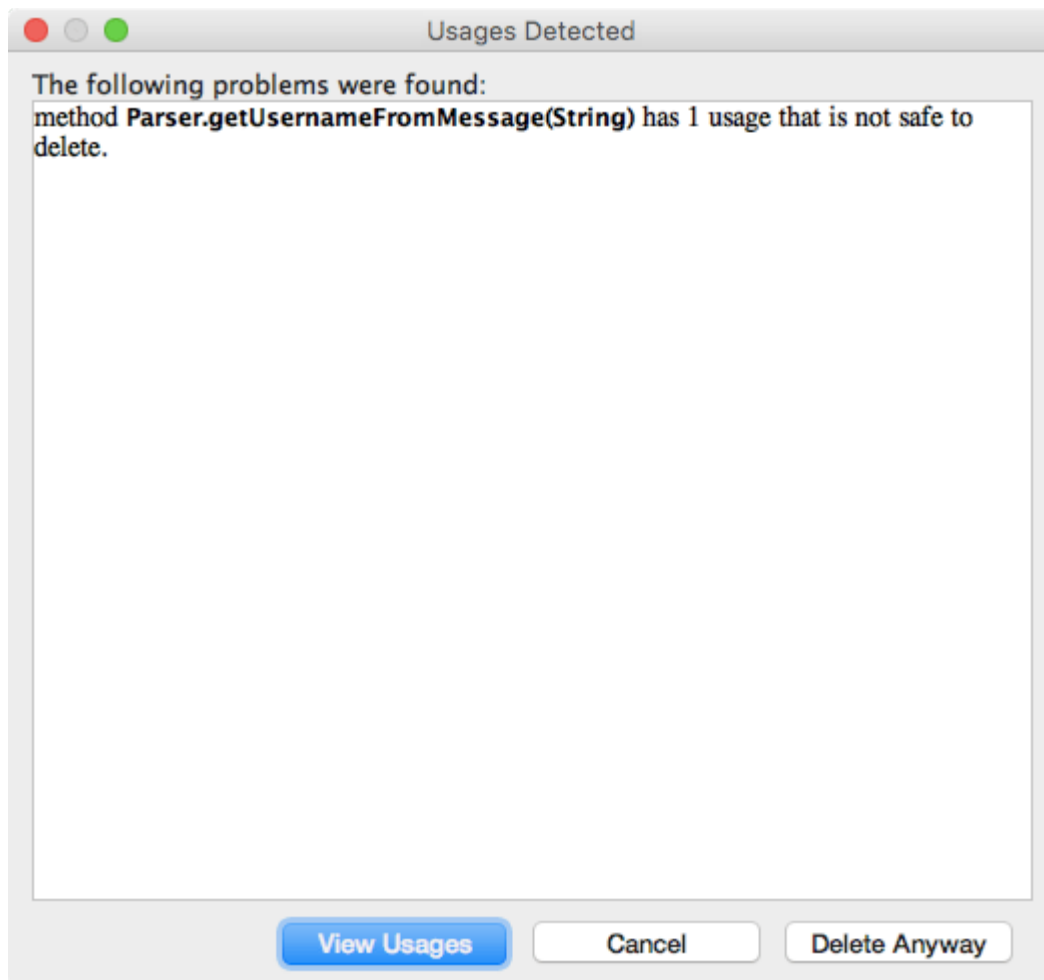
3. Al seleccionar la eliminación segura, aparecerá el cuadro de diálogo de eliminación segura, que le permitirá buscar usos de este método.



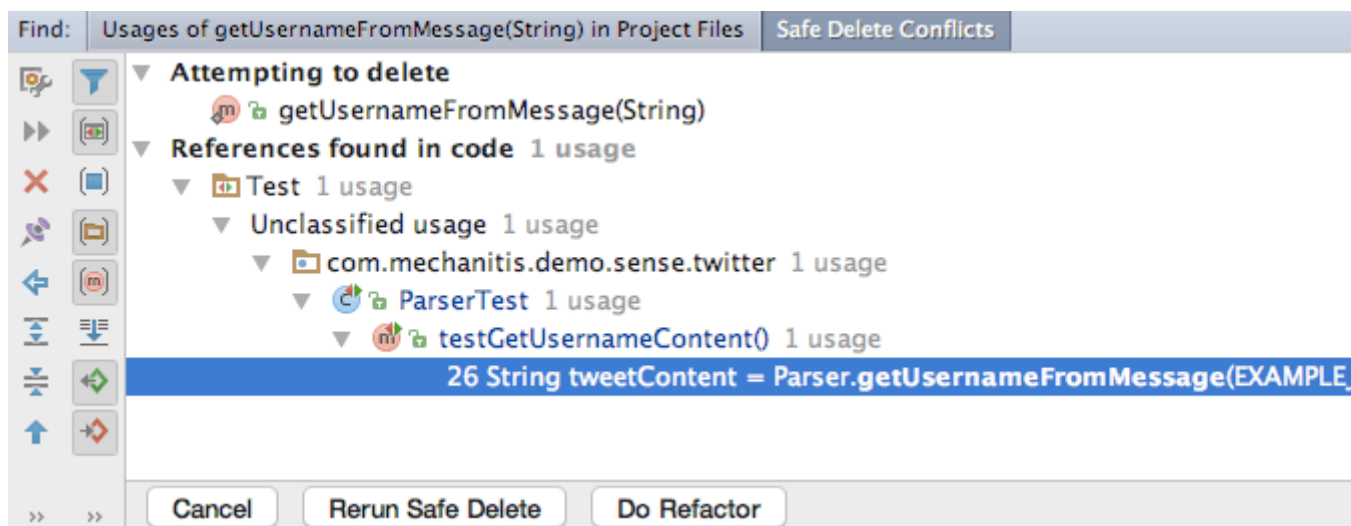
Presione `OK` para continuar y hacer la búsqueda. En nuestro caso, era completamente seguro eliminarlo, por lo que se elimina el método.

4. Es posible que nuestro método "no utilizado" no esté marcado como no utilizado, ya que puede estar cubierto por una prueba. Pero si aún sabemos que no está en uso, o lo hemos verificado mediante `Alt + F7`, aún podemos eliminarlo de forma segura.

Coloque el cursor sobre el nombre del método y presione `Alt + Suprimir`. Aparecerá el cuadro de diálogo de eliminación segura como antes, y esta vez, cuando presione `OK`, IntelliJ IDEA le advertirá que este método tiene usos.



Pulse Ver usos para comprobar cuáles son

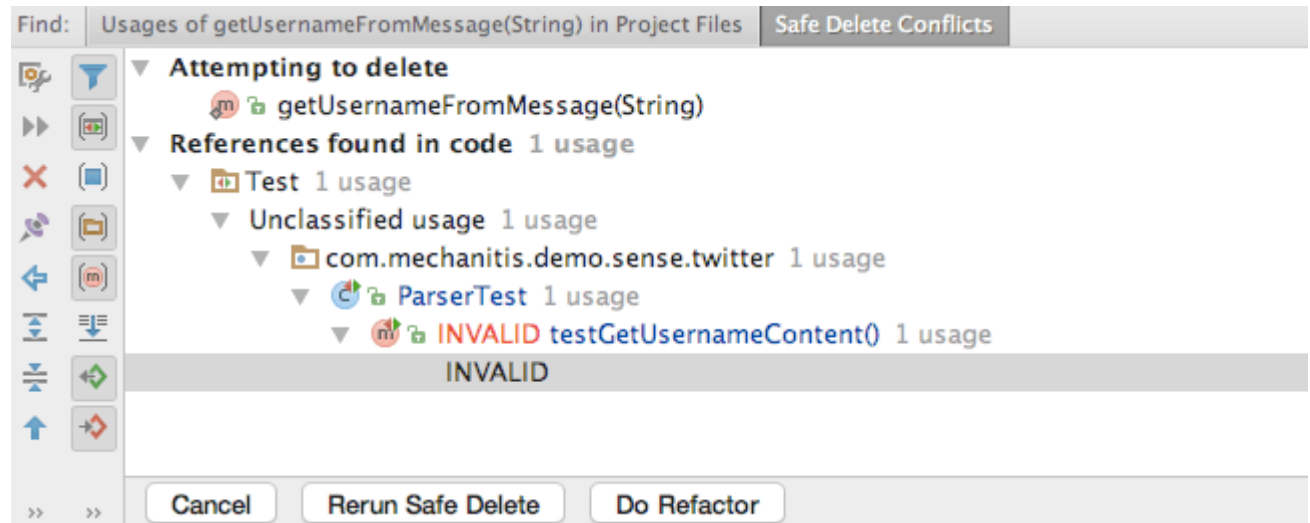


5. Utilice el panel de resultados para navegar a los usos haciendo doble clic en cada uso. En nuestro caso, vemos que hay una prueba que llama al método que queremos eliminar.

```
@Test
@DisplayName("should return the username from the full JSON")
void testGetUsernameContent() {
    String tweetContent = Parser.getUsernameFromMessage(EXAMPLE_INPUT);
    assertEquals(MESSAGE_BODY, tweetContent);
}
```


Dado que esta prueba está ahí para garantizar la exactitud de un método que ya no queremos, también podemos eliminar esta prueba. En la ventana del editor, presione Alt + Eliminar en el nombre del método de prueba y diga Aceptar en el cuadro de diálogo Eliminación segura. Se eliminará el método de prueba.

6. Ahora veremos en nuestra ventana Safe Delete Conflicts que este código ya no es válido.



Dado que este fue nuestro único uso del método que originalmente queríamos eliminar, podemos seleccionar el botón Volver a ejecutar la eliminación segura. Esta vez, cuando presione Aceptar en el cuadro de diálogo Eliminación segura, se habrá eliminado el método getUsernameFromMessage.

Impacto de los borrados

Consejo

Si encuentra un método público que parece no estar en uso pero que forma parte de una API pública, debería estar cubierto por pruebas. Es posible que la advertencia "no utilizado" no signifique que deba eliminar el método, sino que le indica que este método debe probarse.

Las inspecciones de IntelliJ pueden mostrarle código que no está en uso, pero si su código se empaquetará como una biblioteca para que otros lo usen, o si expone una API pública de alguna otra manera, es posible que algunos símbolos públicos se marquen como no usados cuando estén en De hecho, son utilizados por un código que usted no controla. Si parece que los símbolos públicos no se utilizan, debe comprobar si otros sistemas los utilizan de alguna manera.

Los parámetros no utilizados, las variables locales y los campos privados son buenos candidatos para la eliminación, ya que debería ser fácil ver que eliminarlos no afecta ninguna funcionalidad.

El uso de la eliminación segura para eliminar símbolos, ya sea que no se hayan utilizado o no, le permite verificar antes de realizar la refactorización que las áreas afectadas son las que espera y le brinda control sobre los cambios que desea aplicar. Sin embargo, tenga en cuenta la advertencia de que los símbolos públicos pueden ser utilizados por sistemas fuera de su control, por lo que siempre tenga cuidado al eliminarlos.

Conclusión

IntelliJ IDEA tiene una serie de refactorizaciones automáticas disponibles, todas las cuales tienen como objetivo permitirle a usted, el desarrollador, remodelar su código de la manera más de bajo impacto posible. El objetivo es realizar pequeños cambios incrementales, todo el tiempo manteniendo el código en un estado que se pueda compilar. El poder de las capacidades de refactorización radica en encadenar cambios más pequeños para mover el código en la dirección de algún objetivo que tenga en mente: reducir la duplicación, eliminar el código innecesario, esforzarse por la simplicidad, mejorar la legibilidad o una remodelación mayor del diseño.

Los cambios pequeños y simples son posibles, incluso deseables, mientras se trabaja en nuevas funciones o en la corrección de errores, pero recuerde que es posible que los cambios más grandes deban aplicarse por separado para diferenciar entre la refactorización que no debería afectar la funcionalidad existente y los cambios funcionales.