Source : https://www.baeldung.com/intellij-refactoring

Traduction : https://www-baeldung-com.translate.goog/intellij-refactoring?_x_tr_sl=auto&_x_tr_tl=fr&_x_tr_hl=fr

Une introduction au refactoring avec IntelliJ IDEA

Dernière mise à jour : 11 janvier 2024

Écrit par: François Dupire

- EDI
- IntelliJ

Démarrez avec Spring et Spring Boot grâce au cours *Learn Spring* :

>> DÉCOUVREZ LE COURS

1. Vue d'ensemble

Garder le code bien rangé n'est pas toujours facile. Heureusement pour nous, nos IDE sont aujourd'hui plutôt intelligents et peuvent nous aider à y parvenir. Dans ce tutoriel, nous allons nous concentrer sur IntelliJ IDEA, l'éditeur de code Java de JetBrains.

Nous verrons quelques fonctionnalités proposées par l'éditeur pour refactoriser le code , du renommage des variables à la modification d'une signature de méthode.

2. Renommer

2.1. Renommer de base

Tout d'abord, commençons par les bases : renommer . **IntelliJ nous offre la possibilité de renommer différents éléments de notre code : types, variables, méthodes et même packages.**

Pour renommer un élément, nous devons suivre ces étapes :

- Faites un clic droit sur l'élément
- **Déclenchez l' option *Refactor > Renommer***
- Tapez le nouveau nom de l'élément
- appuyez sur *Entrée*

À propos, **nous pouvons remplacer les deux premières étapes en sélectionnant l'élément et en appuyant sur *Shift + F6* .**

Lorsqu'elle est déclenchée, l'action de changement de nom **recherchera dans le code chaque utilisation de l'élément, puis les modifiera avec la valeur fournie** .

Imaginons une classe *SimpleClass avec une méthode d'addition mal nommée, someAdditionMethod* , appelée dans la méthode *main :*

```
public class SimpleClass {
    public static void main(String[] args) {
        new SimpleClass().someAdditionMethod(1, 2);
    }

    public int someAdditionMethod(int a, int b) {
        return a + b;
    }
}
```

Ainsi, si l'on choisit de renommer cette méthode en *add* , IntelliJ produira le code suivant :

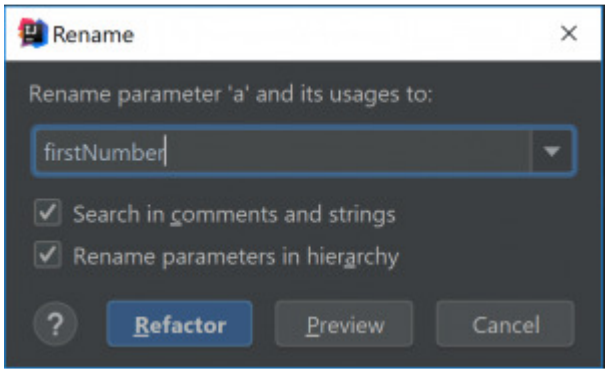
```
public class SimpleClass() {
    public static void main(String[] args) {
        new SimpleClass().add(1, 2);
    }

    public int add(int a, int b) {
        return a + b;
    }
}
```

2.2. Renommer avancé

Cependant, IntelliJ fait plus que rechercher les utilisations du code de nos éléments et les renommer. En fait, quelques options supplémentaires sont disponibles. IntelliJ **peut également rechercher des occurrences dans les commentaires et les chaînes, et même dans les fichiers ne contenant pas de code source** . Quant aux paramètres, il peut les renommer dans la hiérarchie des classes en cas de méthodes remplacées.

Ces options sont disponibles en appuyant à nouveau sur *Shift + F6* avant de renommer notre élément et une fenêtre contextuelle apparaîtra :



L' option *Rechercher dans les commentaires et les chaînes* est disponible pour tout changement de nom. Quant à l' option *Rechercher des occurrences de texte* , elle n'est pas disponible pour les paramètres de méthode et les variables locales. Enfin, l' option *Renommer les paramètres dans la hiérarchie* est disponible uniquement pour les paramètres de méthode.

Ainsi, **si une correspondance est trouvée avec l'une de ces deux options, IntelliJ la montrera et nous offrira la possibilité de désactiver certaines des modifications** (par exemple, au cas où cela correspondrait à quelque chose sans rapport avec notre changement de nom).

Ajoutons du Javadoc à notre méthode, puis renommons son premier paramètre, *a* :

```
/**
  * Adds a and b
  * @param a the first number
  * @param b the second number
  */
public int add(int a, int b) {...}
```

En cochant la première option dans la pop-up de confirmation, IntelliJ correspond à toute mention des paramètres dans le commentaire Javadoc de la méthode et propose également de les renommer :

```
/**
  * Adds firstNumber and b
  * @param firstNumber the first number
  * @param b the second number
  */
public int add(int firstNumber, int b) {...}
```

Enfin, il faut noter **qu'IntelliJ est intelligent et recherche principalement les usages dans le cadre de l'élément renommé.** Dans notre cas, cela signifierait qu'un commentaire situé en dehors de la méthode (sauf pour le Javadoc) et contenant une mention à *a* n'aurait pas été pris en compte pour le renommage.

3. Extraction

Parlons maintenant de l'extraction . **L'extraction nous permet de récupérer un morceau de code et de le mettre dans une variable, une méthode ou même une classe.** IntelliJ gère cela assez intelligemment car il recherche des morceaux de code similaires et propose de les extraire de la même manière.

Ainsi, dans cette section, nous apprendrons comment tirer parti de la fonctionnalité d'extraction offerte par IntelliJ.

3.1. Variables

Tout d'abord, commençons par l'extraction des variables. **Cela signifie des variables locales, des paramètres, des champs et des constantes.** Pour extraire une variable, il faut suivre ces étapes :

- Sélectionnez une expression qui correspond à une variable
- Cliquez avec le bouton droit sur la zone sélectionnée
- **Déclenchez l' option *Refactor > Extraire > Variable/Paramètre/Champ/Constante***
- Choisissez entre les options *Remplacer cette occurrence uniquement* ou *Remplacer toutes les x occurrences* , si elles sont proposées.
- Saisissez un nom pour l'expression extraite (si celui choisi ne nous convient pas)
- appuyez sur *Entrée*

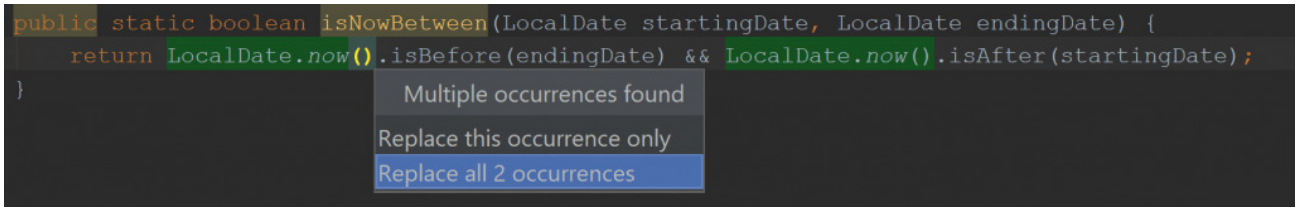
Quant au renommage, il est possible d'utiliser des raccourcis clavier au lieu d'utiliser le menu. **Les raccourcis par défaut sont respectivement *Ctrl + Alt + V, Ctrl + Alt + P, Ctrl + Alt + F* et *Ctrl + Alt + C* .**

IntelliJ essaiera de deviner un nom pour notre expression extraite, en fonction de ce que renvoie l'expression. S'il ne correspond pas à nos besoins, nous sommes libres de le modifier avant de confirmer l'extraction.

Illustrons avec un exemple. On pourrait imaginer ajouter une méthode à notre classe *SimpleClass* nous indiquant si la date du jour est comprise entre deux dates données :

```
public static boolean isNowBetween(LocalDate startingDate, LocalDate endingDate)
{
    return LocalDate.now().isAfter(startingDate) &&
LocalDate.now().isBefore(endingDate);
}
```

Disons que nous souhaitons modifier notre implémentation parce que nous utilisons *LocalDate.now()* deux fois et que nous aimerions nous assurer qu'elle a exactement la même valeur dans les deux évaluations. Sélectionnons simplement l'expression et extrayons-la dans une variable locale, *maintenant :*



Ensuite, notre appel *LocalDate.now()* est capturé dans une variable locale :

```
public static boolean isNowBetween(LocalDate startingDate, LocalDate endingDate)
{
    LocalDate now = LocalDate.now();
    return now.isAfter(startingDate) && now.isBefore(endingDate);
}
```

En cochant l' option *Remplacer tout* , nous nous sommes assurés que les deux expressions ont été remplacées en même temps.

3.2. Méthodes

Voyons maintenant comment extraire des méthodes à l'aide d'IntelliJ :

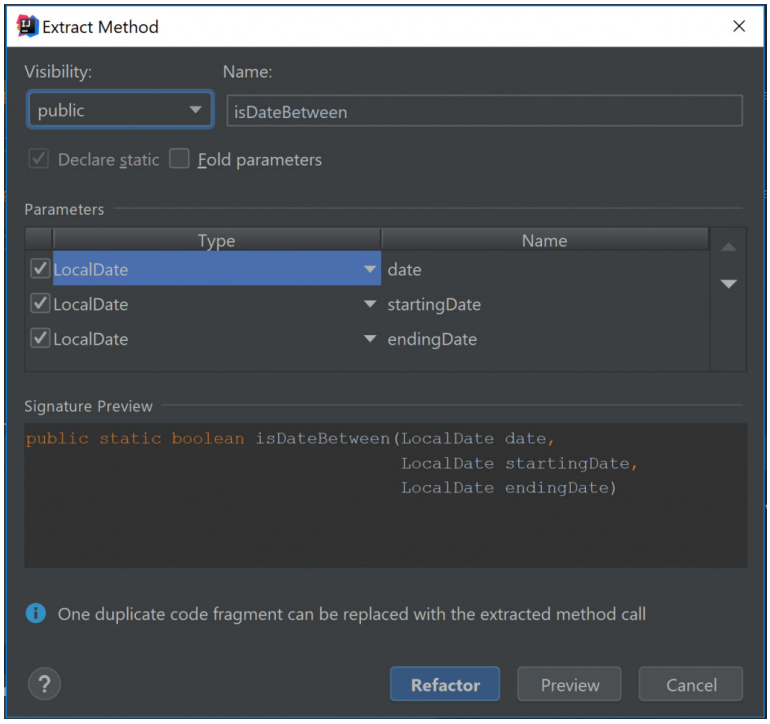
- Sélectionnez l'expression ou les lignes de code correspondant à la méthode que nous souhaitons créer
- Cliquez avec le bouton droit sur la zone sélectionnée
- **Déclenchez l' option *Refactor > Extraire > Méthode***
- Saisissez les informations de la méthode : son nom, sa visibilité et ses paramètres
- appuyez sur *Entrée*

Appuyer sur *Ctrl + Alt + M* après avoir sélectionné le corps de la méthode fonctionne également.

Réutilisons notre exemple précédent et disons que nous voulons avoir une méthode vérifiant si une date est comprise entre d'autres dates. Ensuite, il suffirait de sélectionner notre dernière ligne dans la méthode *isNowBetween* et de déclencher la fonction d'extraction de méthode.

Dans la boîte de dialogue ouverte, nous pouvons voir qu'IntelliJ a déjà repéré les trois paramètres nécessaires : *StartingDate* ,*endingDate* et *now* . Comme nous souhaitons que cette méthode soit la plus générique possible, nous renommeons le paramètre *now en date* . Et par souci de cohésion, nous le plaçons comme premier paramètre.

Enfin, nous donnons un nom à notre méthode, *isDateBetween* , et finalisons le processus d'extraction :



On obtiendrait alors le code suivant :

```
public static boolean isNowBetween(LocalDate startingDate, LocalDate endingDate)
{
    LocalDate now = LocalDate.now();
    return isDateBetween(now, startingDate, endingDate);
}

private static boolean isDateBetween(LocalDate date, LocalDate startingDate,
LocalDate endingDate) {
    return date.isBefore(endingDate) && date.isAfter(startingDate);
}
```

Comme nous pouvons le voir, l'action a déclenché la création de la nouvelle méthode *isDateBetween* , également appelée dans la méthode *isNowBetween* . La méthode est privée, par défaut. Bien entendu, cela aurait pu être modifié à l'aide de l'option de visibilité.

3.3. Des classes

Après tout cela, nous souhaiterions peut-être placer nos méthodes liées aux dates dans une classe spécifique, axée sur la gestion des dates, disons : *DateUtils* . Encore une fois, c'est assez simple :

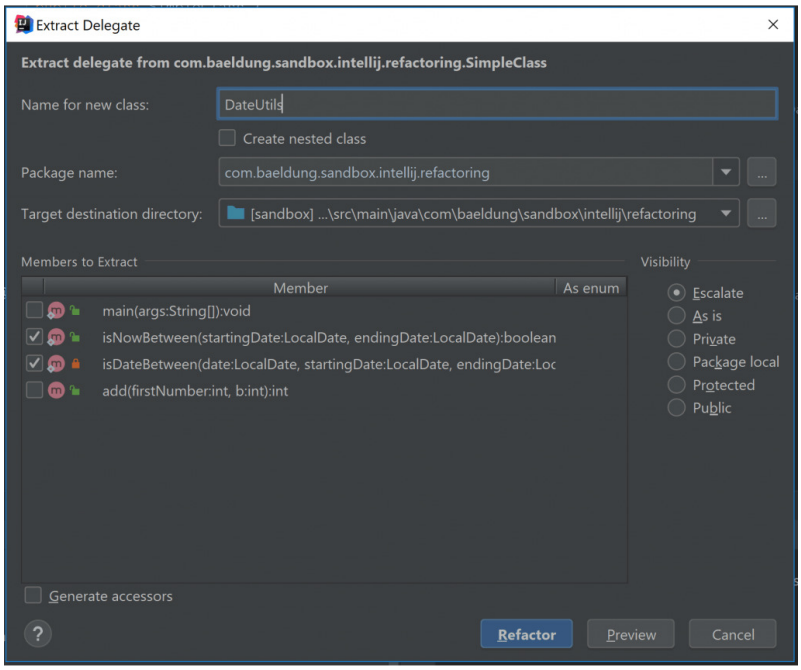
- Faites un clic droit dans la classe qui contient les éléments que nous voulons déplacer
- **Déclenchez l' option *Refactor > Extraire > Déléguer***
- Saisissez les informations de la classe : son nom, son package, les éléments à déléguer, la visibilité de ces éléments
- appuyez sur *Entrée*

Par défaut, aucun raccourci clavier n'est disponible pour cette fonctionnalité.

Disons, avant de déclencher la fonctionnalité, que nous appelions nos méthodes liées aux dates dans la méthode *main* :

```
isNowBetween(LocalDate.MIN, LocalDate.MAX);
isDateBetween(LocalDate.of(2019, 1, 1), LocalDate.MIN, LocalDate.MAX);
```

Ensuite, nous déléguons ces deux méthodes à une classe *DateUtils* en utilisant l'option délégué :



Déclencher la fonctionnalité produirait le code suivant :

```
public class DateUtils {
    public static boolean isNowBetween(LocalDate startingDate, LocalDate
endingDate) {
        LocalDate now = LocalDate.now();
        return isDateBetween(now, startingDate, endingDate);
    }

    public static boolean isDateBetween(LocalDate date, LocalDate startingDate,
LocalDate endingDate) {
```

```
        return date.isBefore(endingDate) && date.isAfter(startingDate);
    }
}
```

Nous pouvons voir que la méthode *isDateBetween* a été rendue *publique* . C'est le résultat de l'option de visibilité, qui est définie pour *augmenter* par défaut. ***Escalate*** **signifie que la visibilité sera modifiée afin de garantir que les appels en cours vers les éléments délégués sont toujours en cours de compilation.**

Dans notre cas, *isDateBetween* est utilisé dans la méthode *principale* de *SimpleClass :*

```
DateUtils.isNowBetween(LocalDate.MIN, LocalDate.MAX);
DateUtils.isDateBetween(LocalDate.of(2019, 1, 1), LocalDate.MIN, LocalDate.MAX);
```

Ainsi, lors du déplacement de la méthode, il est nécessaire de la rendre non privée.

Il est cependant possible de donner une visibilité spécifique à nos éléments en sélectionnant les autres options.

4. Inline

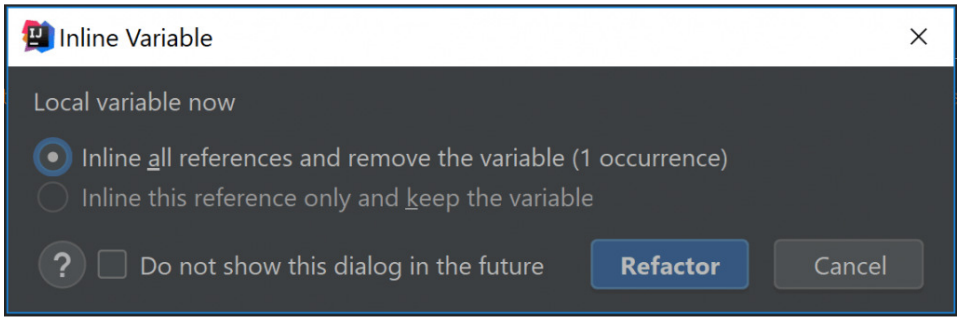
Maintenant que nous avons abordé l'extraction, parlons de son homologue : inlining . **L'inlining consiste à prendre un élément de code et à le remplacer par ce dont il est fait.** Pour une variable, ce serait l'expression qui lui a été attribuée. Pour une méthode, ce serait son corps. Plus tôt, nous avons vu comment créer une nouvelle classe et lui déléguer certains de nos éléments de code. Mais il peut arriver que **nous souhaitions déléguer une méthode à une classe existante** . C'est le sujet de cette section.

Pour intégrer un élément, nous devons cliquer avec le bouton droit sur cet élément – soit sa définition, soit une référence à celui-ci – et déclencher l' option ***Refactor > Inline*** . Nous pouvons également y parvenir en sélectionnant l'élément et en appuyant sur les touches ***Ctrl + Alt + N.***

À ce stade, IntelliJ nous proposera plusieurs options, que nous souhaitions intégrer une variable ou une méthode, que nous sélectionnions une définition ou une référence. Ces options sont :

- Insérez toutes les références et supprimez l'élément
- Insérez toutes les références, mais conservez l'élément
- Insérez uniquement la référence sélectionnée et conservez l'élément

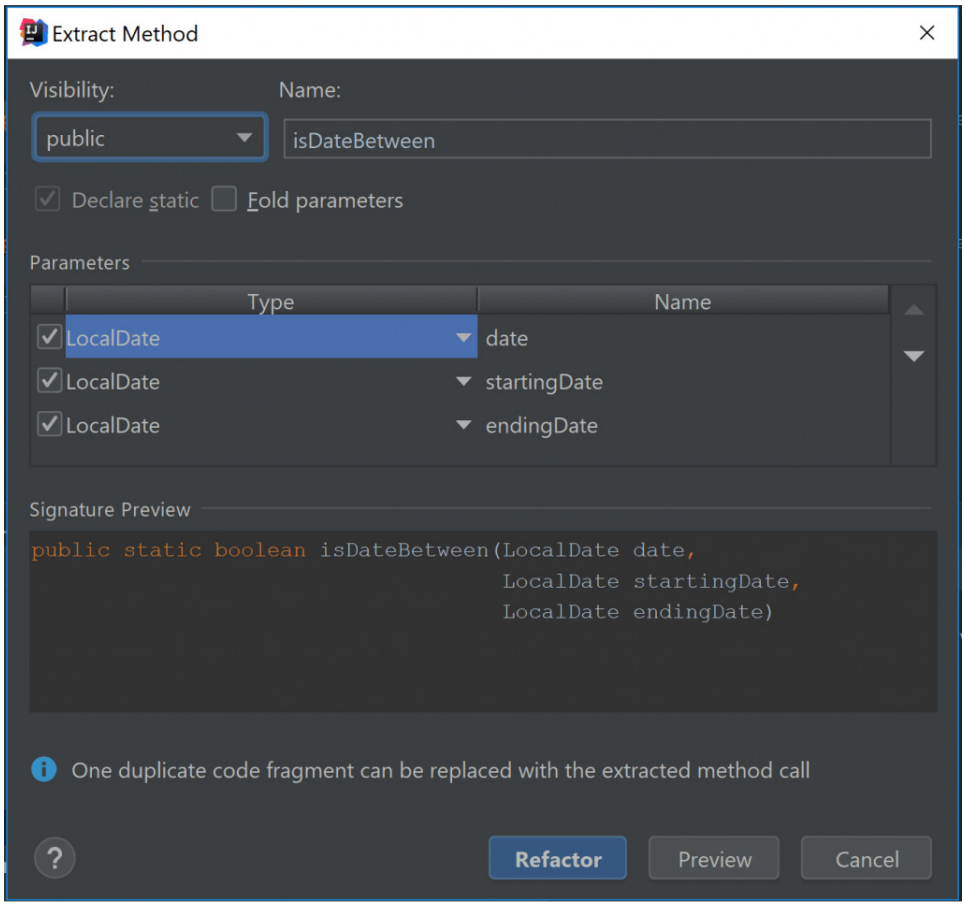
Prenons notre méthode *isNowBetween* et débarrassons-nous de la variable *now* , qui semble maintenant un peu exagérée :



En inlinant cette variable, nous obtiendrions le résultat suivant :

```
public static boolean isNowBetween(LocalDate startingDate, LocalDate endingDate)
{
    return isDateBetween(LocalDate.now(), startingDate, endingDate);
}
```

Dans notre cas, la seule option était de supprimer toutes les références et de supprimer l'élément. Mais imaginons que nous souhaitions également nous débarrasser de l' appel *isDateBetween* et choisir de l'intégrer. IntelliJ nous offrirait alors les trois possibilités dont nous avons parlé précédemment :



Choisir le premier remplacerait tous les appels par le corps de la méthode et supprimerait la méthode. Quant au second, il remplacerait tous les appels par le corps de la méthode mais conserverait la méthode. Et enfin, le dernier remplacerait uniquement l'appel en cours par le corps de la méthode :

```
public class DateUtils {
    public static boolean isNowBetween(LocalDate startingDate, LocalDate
endingDate) {
        LocalDate date = LocalDate.now();
        return date.isBefore(endingDate) && date.isAfter(startingDate);
    }

    public static boolean isDateBetween(LocalDate date, LocalDate startingDate,
LocalDate endingDate) {
        return date.isBefore(endingDate) && date.isAfter(startingDate);
    }
}
```

Notre méthode *principale* dans *SimpleClass* reste également intacte.

5. Déménager

Plus tôt, nous avons vu comment créer une nouvelle classe et lui déléguer certains de nos éléments de code. Mais il peut arriver que **nous souhaitions déléguer une méthode à une classe existante** . C'est le sujet de cette section.

Afin de déplacer un élément, nous devons suivre ces étapes :

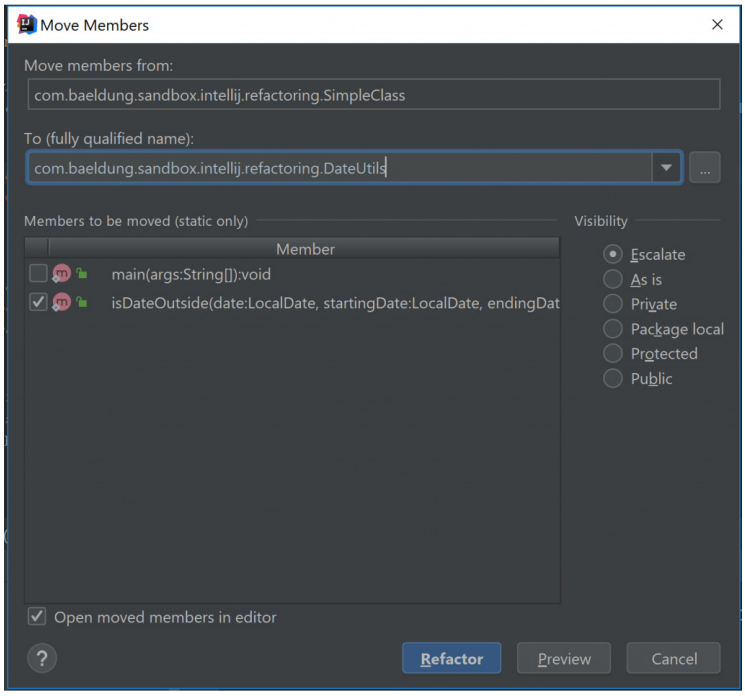
- Sélectionnez l'élément à déplacer
- Faites un clic droit sur l'élément
- **Déclenchez l' option *Refactor> Déplacer***
- Choisissez la classe destinataire et la visibilité de la méthode
- appuyez sur *Entrée*

Nous pouvons également y parvenir en appuyant sur *F6* après avoir sélectionné l'élément.

Disons que nous ajoutons une nouvelle méthode à notre *SimpleClass* , *isDateOutstide()* , qui nous dira si une date est située en dehors d'un intervalle de dates :

```
public static boolean isDateOutside(LocalDate date, LocalDate startingDate,
LocalDate endingDate) {
    return !DateUtils.isDateBetween(date, startingDate, endingDate);
}
```

On se rend alors compte que sa place devrait être dans notre classe *DateUtils* . Nous décidons donc de le déplacer :



Notre méthode est désormais dans la classe *DateUtils* . Nous pouvons voir que la référence à *DateUtils* à l'intérieur de la méthode a disparu car elle n'est plus nécessaire :

```
public static boolean isDateOutside(LocalDate date, LocalDate startingDate,
LocalDate endingDate) {
    return !isDateBetween(date, startingDate, endingDate);
}
```

L'exemple que nous venons de faire fonctionne bien car il concerne une méthode statique. Cependant, dans le cas d'une méthode d'instance, les choses ne sont pas si simples.

Si nous voulons déplacer une méthode d'instance , **IntelliJ recherchera les classes référencées dans les champs de la classe actuelle et proposera de déplacer la méthode vers l'une de ces classes** (à condition qu'elles soient à nous pour les modifier).

Si aucune classe modifiable n'est référencée dans les champs, alors IntelliJ propose de rendre la méthode *statique* avant de la déplacer.

6. Modification d'une signature de méthode

Enfin, nous parlerons d'une fonctionnalité nous permettant de changer la signature d'une méthode . **Le but de cette fonctionnalité est de manipuler tous les aspects d'une signature de méthode.**

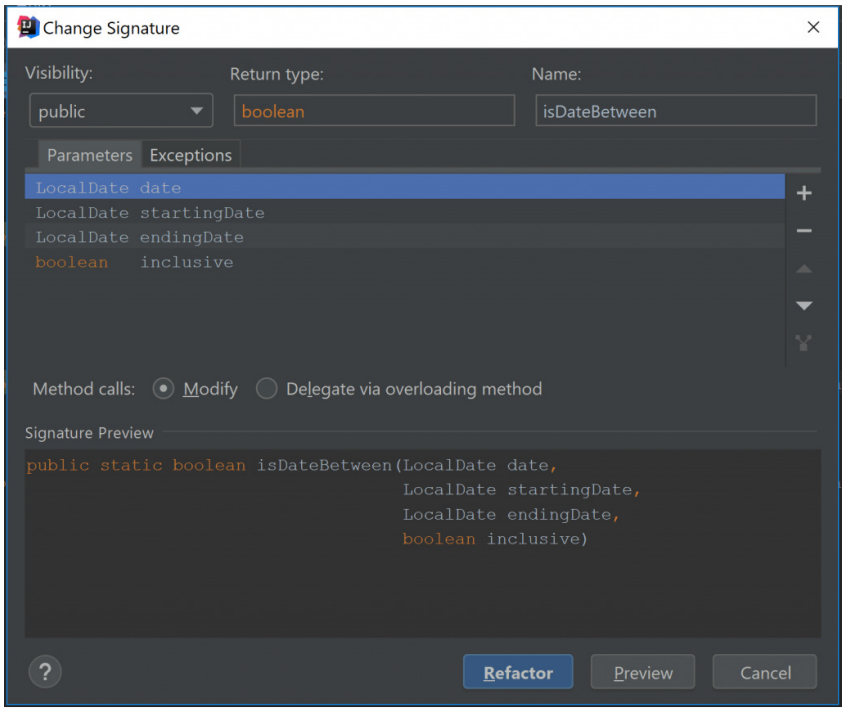
Comme d'habitude, il faut passer par quelques étapes pour déclencher la fonctionnalité :

- Sélectionnez la méthode à modifier
- Cliquez avec le bouton droit sur la méthode
- **Déclenchez l' option *Refactor > Modifier la signature***
- Apporter des modifications à la signature de la méthode
- appuyez sur *Entrée*

Si nous préférons utiliser les raccourcis clavier, il est **également possible d'utiliser *Ctrl + F6* .**

Cette fonctionnalité ouvrira une boîte de dialogue très similaire à la fonctionnalité d'extraction de méthode. On a donc **les mêmes possibilités que lorsqu'on extrait une méthode** : changer son nom, sa visibilité mais aussi ajouter/supprimer des paramètres et les affiner.

Imaginons que nous souhaitions modifier notre implémentation de *isDateBetween* pour considérer les limites de date comme inclusives ou exclusives. Pour ce faire, nous souhaitons ajouter un paramètre *booléen* à notre méthode :



En changeant la signature de la méthode, nous pouvons ajouter ce paramètre, le nommer et lui donner une valeur par défaut :

```
public static boolean isDateBetween(LocalDate date, LocalDate startingDate,
   LocalDate endingDate, boolean inclusive) {
    return date.isBefore(endingDate) && date.isAfter(startingDate);
}
```

Après cela, il ne reste plus qu'à adapter le corps de la méthode en fonction de nos besoins.

Si nous l'avions voulu, nous aurions pu cocher l' option *Déléguer via la méthode de surcharge* afin de créer une autre méthode avec le paramètre au lieu de modifier celle actuelle.

7. Tirez vers le haut et poussez vers le bas

Notre code Java a généralement des hiérarchies de classes – **les classes dérivées étendent une classe de base** .

Parfois, nous souhaitons déplacer des membres (méthodes, champs et constantes) entre ces classes. C'est là que la dernière refactorisation s'avère utile : elle nous permet d' **extraire les membres d'une classe dérivée vers la classe de base ou de les pousser vers le bas d'une classe de base vers chaque classe dérivée** .

7.1. Tirer vers le haut

Commençons par extraire une méthode de la classe de base :

- Sélectionnez un membre d'une classe dérivée à afficher
- Cliquez avec le bouton droit sur le membre
- **Déclenchez l' option *Refactor > Pull Members Up…***
- Appuyez sur le bouton *Refactoriser*

Par défaut, aucun raccourci clavier n'est disponible pour cette fonctionnalité.

Disons que nous avons une classe dérivée appelée *Derived.* Il utilise une méthode privée *doubleValue() :*

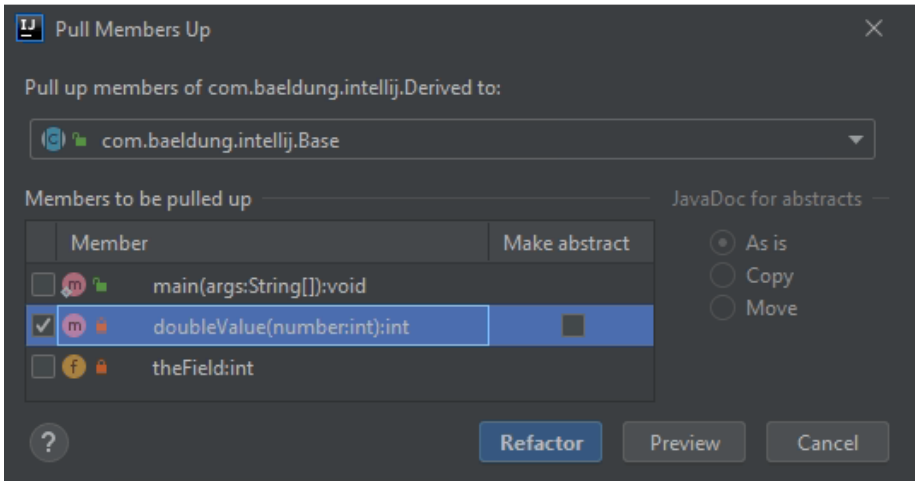
```
public class Derived extends Base {

    public static void main(String[] args) {
        Derived subject = new Derived();
        System.out.println("Doubling 21. Result: " + subject.doubleValue(21));
    }

    private int doubleValue(int number) {
        return number + number;
    }
}
```

La classe de base *Base* est vide.

Alors, que se passe-t-il lorsque nous extrayons *doubleValue()* dans *Base* ?



Deux choses arrivent à *doubleValue()* lorsque nous appuyons sur « Refactor » dans la boîte de dialogue ci-dessus :

- il passe à la classe *Base*
- sa visibilité passe de *privée* à *protégée* afin que la classe *dérivée* puisse toujours l'utiliser

La classe *Base* a ensuite maintenant la méthode :

```
public class Base {
    protected int doubleValue(int number) {
        return number + number;
    }
}
```

La boîte de dialogue permettant d'extraire les membres (photo ci-dessus) nous offre des options supplémentaires :

- nous pouvons sélectionner d'autres membres et les extraire tous en même temps
- nous pouvons prévisualiser nos modifications avec le bouton « Aperçu »
- seules les méthodes ont une case à cocher dans la colonne « Make abstract ». Si elle est cochée, cette option donnera à la classe de base une définition de méthode abstraite lors du pull-up. La méthode réelle restera dans la classe dérivée mais obtiendra une annotation *@Override* . **Par conséquent, les autres classes dérivées ne seront plus compilées** car il leur manque l'implémentation de cette nouvelle méthode de base abstraite.

7.2. Abaisser

Enfin, déplaçons un membre vers la classe dérivée. C'est l'opposé du pull up que nous venons d'effectuer :

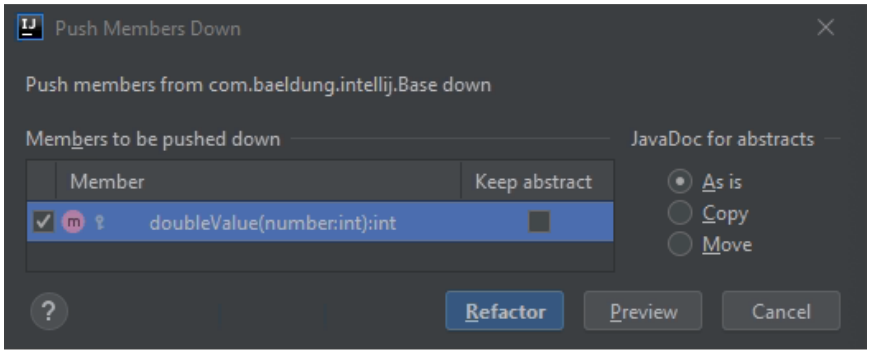
- Sélectionnez un membre d'une classe de base à pousser vers le bas
- Cliquez avec le bouton droit sur le membre
- **Déclenchez l' option *Refactor > Push Members Down…***
- Appuyez sur le bouton *Refactoriser*

Comme pour l'extraction de membres, aucun raccourci clavier n'est disponible par défaut pour cette fonctionnalité.

Abaissons à nouveau la méthode que nous venons de proposer. La classe *Base* ressemblait à ceci à la fin de la section précédente :

```
public class Base {
    protected int doubleValue(int number) {
        return number + number;
    }
}
```

Maintenant, poussons *doubleValue()* jusqu'à la classe *Derived :*



Il s'agit de la classe *Derived* après avoir appuyé sur « Refactor » dans la boîte de dialogue ci-dessus. La méthode *doubleValue()* est de retour :

```
public class Derived extends Base {
    private int theField = 5;

    public static void main(String[] args) {
        Derived subject = new Derived();
        System.out.println( "Doubling 21. Result: " + subject.doubleValue(21));
    }

    protected int doubleValue(int number) {
        return number + number;
    }
}
```

Désormais, la classe *Base* et la classe *Derived* sont de retour à leur point de départ dans la section « Pull Up » précédente. Autrement dit, *doubleValue()* a conservé la visibilité *protégée* qu'elle avait dans *Base* (elle était *privée* à l'origine).

IntelliJ 2019.3.4 affiche en fait un avertissement lorsque vous appuyez sur *doubleValue()* : « Les membres poussés ne seront pas visibles depuis certains sites d'appel ». Mais comme nous pouvons le voir dans la classe *Derived* ci-dessus, *doubleValue()* est effectivement visible par la méthode *main()* .

La boîte de dialogue permettant de pousser les membres vers le bas (photo ci-dessus) nous offre également des options supplémentaires :

- si nous avons plusieurs classes dérivées, alors IntelliJ poussera les membres dans chaque classe dérivée
- nous pouvons repousser plusieurs membres
- nous pouvons prévisualiser nos modifications avec le bouton « Aperçu »
- seules les méthodes ont une case à cocher dans la colonne « Conserver le résumé » – Cela revient à extraire les membres : si elle est cochée, cette option laissera une méthode abstraite dans la classe de base. Contrairement à l'extraction de membres, cette option placera les implémentations de méthodes dans toutes les classes dérivées. Ces méthodes obtiendront également une annotation *@Override*

8. Conclusion

Dans cet article, nous avons eu l'occasion d'approfondir certaines des fonctionnalités de refactoring proposées par IntelliJ. Bien entendu, nous n'avons pas couvert toutes les possibilités car IntelliJ est un outil très puissant. Pour en savoir plus sur cet éditeur, on peut toujours se référer à sa documentation .

Nous avons vu quelques choses comme comment renommer nos éléments de code et comment extraire certains comportements en variables, méthodes ou classes. Nous avons également appris à intégrer certains éléments si nous n'en avions pas besoin seuls, à déplacer du code ailleurs ou même à modifier complètement une signature de méthode existante.