R6.A06.D05.alt : Maintenance applicative
Feuille TD-TP n° 1

Refactoring sur IntelliJ – Prise en main
éléments de correction

Objectifs :

1.- Découverte des outils de l'IDE destinés au Refactoring chirurgical
2.- S'exercer au Refactoring chirurgical sur un exemple simple.

Exercice 1 – Refactoring sur IntelliJ – Prise en main

Ce tuto montre la mise en œuvre, avec IntelliJ, des actions de refactoring chirurgical présentées dans le cours, chapitre 2.

1. Renommer

a. Renommer basique

Exemple - TAF : Dans la classe SippleClass, renommer la méthode sommeAdditionMethod → add

Étapes

- Clic-droit sur l'élément (dans l'exemple, on peut sélectionner le nom figurant, soit dans *l'appel* soit dans la *définition* de la méthode)
- Déclencher l'option ***Refactor > Rename (ou Shift-F6)***
- Saisir le nouveau nom
- Valider avec la touche *Entrée*

Effet
Toutes les occurrences de la méthode ont été modifiées, y compris dans les fichiers de test.

b. Renommer *avancé*

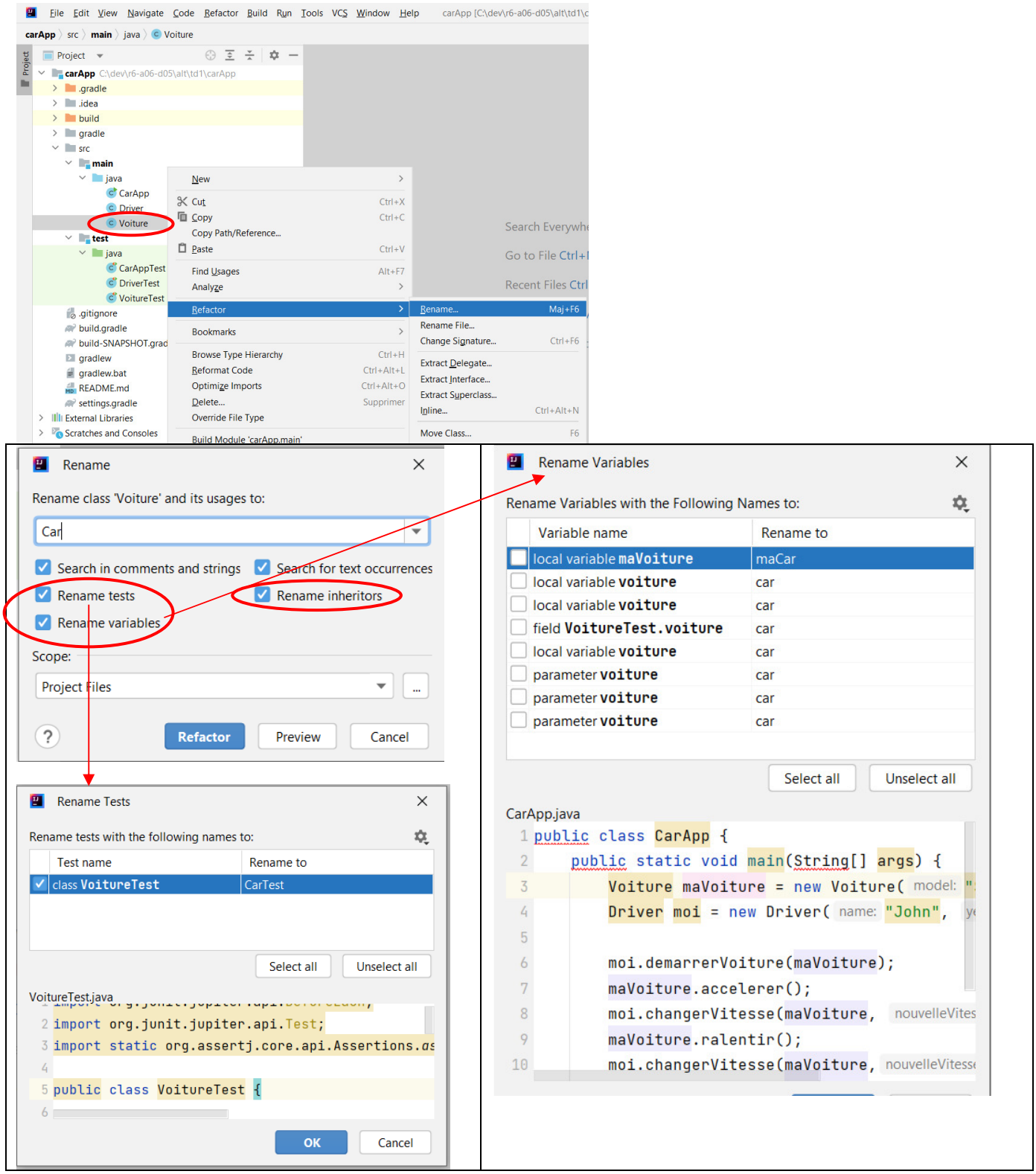
Le renommage peut aussi concerner les occurrences dans les commentaires (dont le Javadoc), les chaînes de caractères, et aussi des fichiers ne contenant pas de code source.

Démarche :

- Utiliser les menus contextuels proposés Cf exemple ci-joint	 ….Maj+F6 

c. Renommage d'une classe – Attention

- Positionner le curseur sur le nom de la classe **située sur le panneau Gauche de éléments du projet**



Différentes fenêtres de Dialogue permettent alors de répercuter automatiquement ce changement au niveau des classes de test, classes héritées de la classe renommée, et déclaration des variables de cette classe

2. Extraire - Extraction

a. Extraction de variable

Cela peut être une extraction de variable locale, de paramètre, de champ, de constante.

Étapes

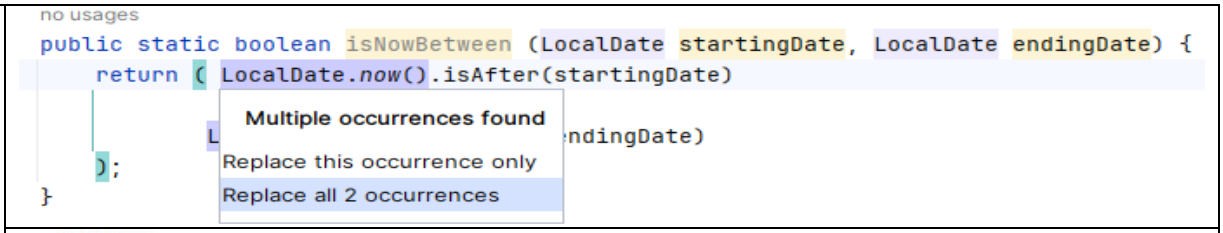
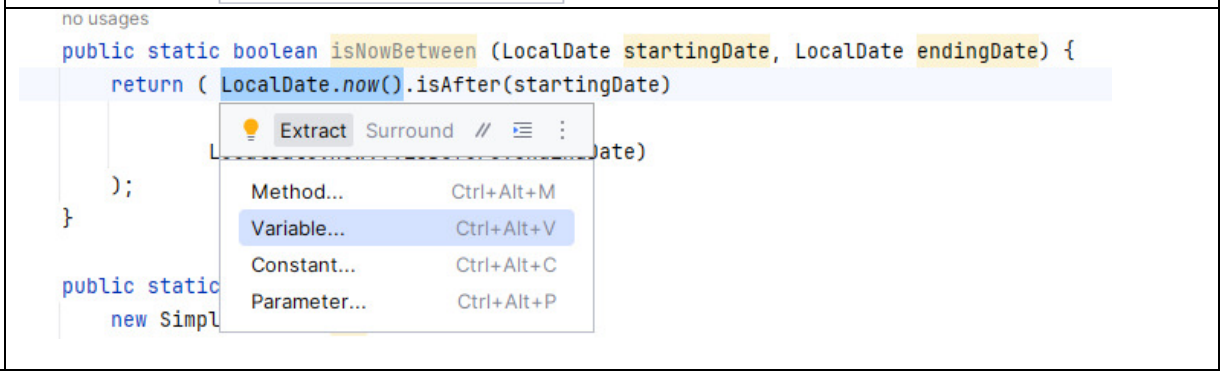
- Sélection de l'expression qui sera remplacée
- Clic-droit
- Déclencher l'option ***Refactor > Introduce Variable/Constant/Field/Parameter***
- Choisir les options, si elles sont proposées (remplacer cette occurrence , remplacer toutes les occurrences)
- Saisir le nom
- Valider avec *Entrée*

Exemple – TAF = Travail à Faire : Extraire une variable, now, calculée à partir de la méthode LocalDate.now()
Justification :
Dans la méthode isNowBetween() de la classe SimpleClass, Il y a 2 appels à la méthode LocalSate.now().

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

Pour s'assurer que ces 2 appels utiliseront la même valeurs → on va faire une extraction de variable locale (now) qui sera calculée une seule fois (par LocalDate.now()), puis utilisée deux fois..
Une fois l'extraction réalisée, la méthode devient :

Etapes :

L'extraction peut se faire via le menu classique (Refactor>…) , ou le raccourci proposé par IntelliJ	 

Version finale :

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

b. Extraction de méthode

Étapes

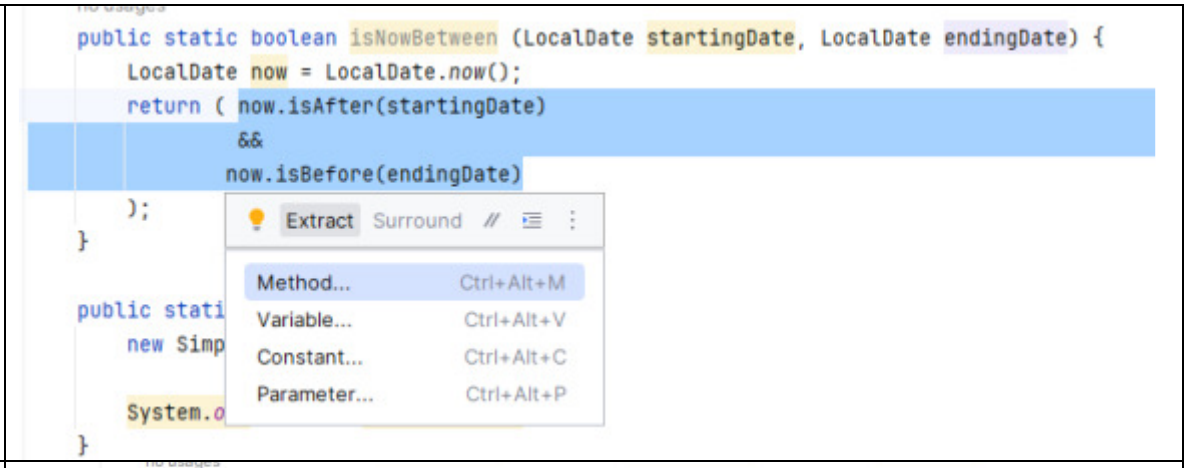
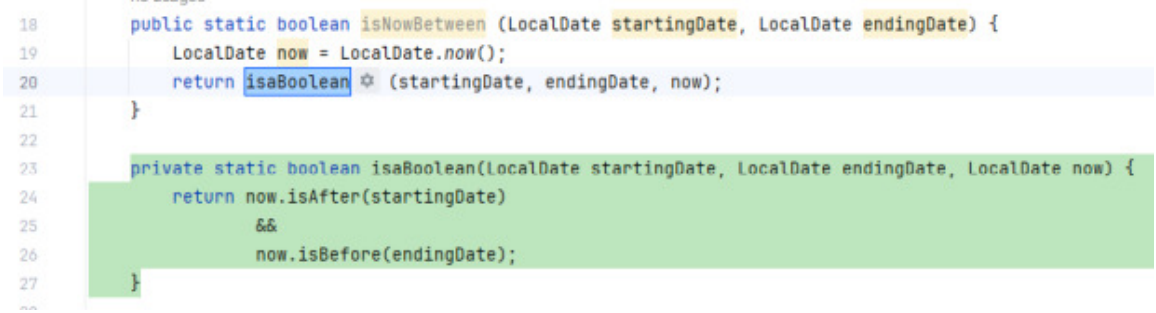
- Sélection de la portion de code qui formera le corps de la méthode à extraire
- Clic-droit ou **Ctrl+Alt+M**
- Déclencher l'option ***Refactor > Introduce Method***
- Saisir le nom, sa visibilité et ses paramètres
- Valider avec *Entrée*

Exemple - TAF : Dans la méthode isNowBetween(), créer une méthode vérifiant si une date est comprise entre 2 dates. Il faut extraire une méthode à partir de l'expression surlignée ci-dessous.

Version initiale :

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

Etapes :

<table>
<tr>
<td></td>
<td> 
</td>
</tr>
<tr>
<td>Une méthode privée est automatiquement créée, un nom par défaut lui est donné (modifiable), L'expression est remplacée par l'appel de la méthode</td>
<td> 
</td>
</tr>
</table>

Version finale :

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

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

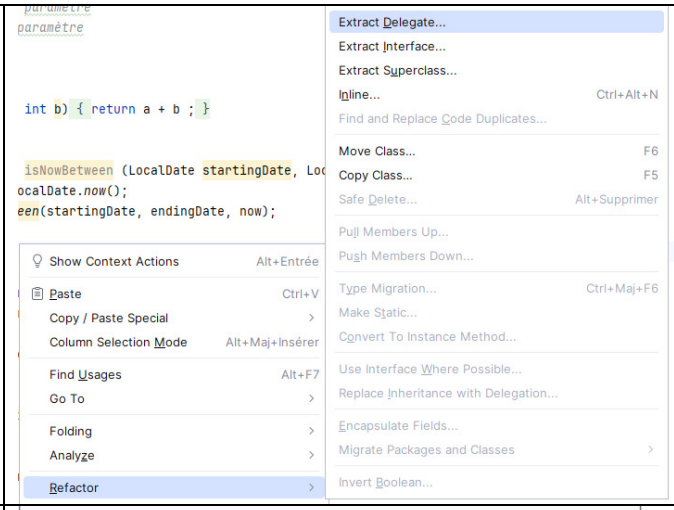
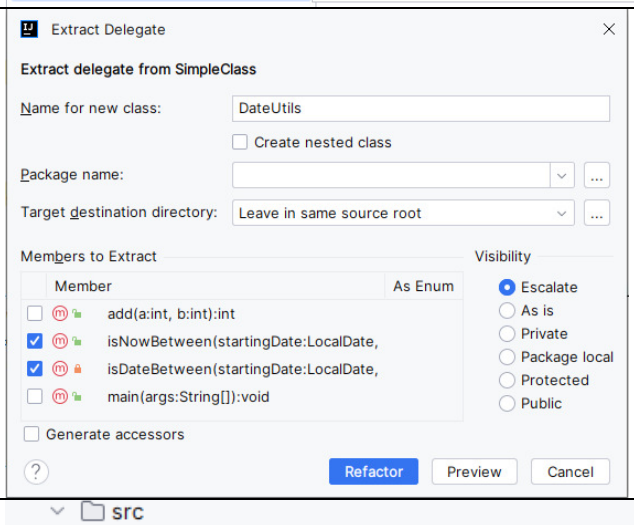
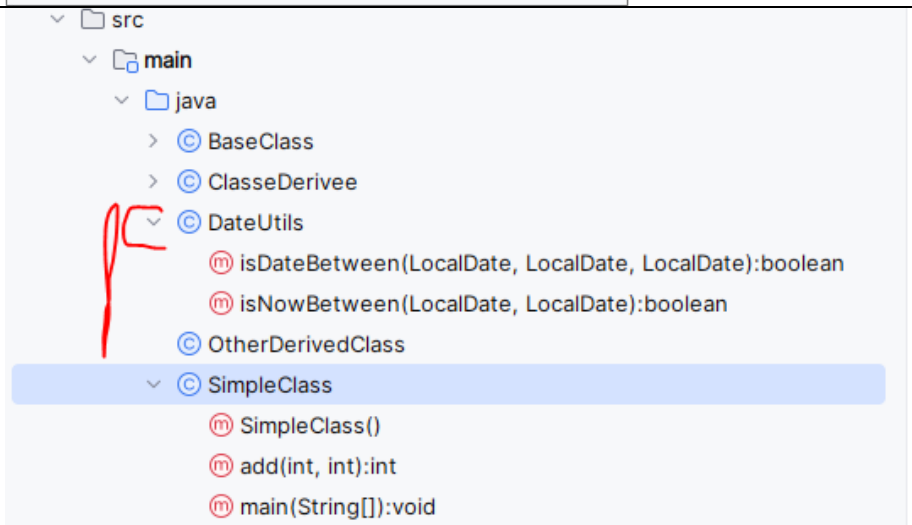
c. Extraction de classe

Exemple - TAF : Placer les méthodes de la classe SimpleClass liées aux dates dans une classe spécifique à la gestion des dates : **DateUtils**.

Étapes

- Clic-droit dans le corps de la classe contenant les éléments que l'on souhaite déplacer.
- Déclencher l'option ***Refactor > Extract Delegate***
- Saisir le les informations de la classe : nom, package, élements à déléguer et leur visibilité
- Valider avec *Entrée*

Illustration

Refactor > Extract delegate	
Saisie des informations liées à la classe à créer	
La nvlle classe est créée avec les 2 méthodes extraites. L'ancienne classe est modifiée	

Version finale de la classe SimpleClass :

```
public class SimpleClass {

    public SimpleClass() {
    }
    /**
     * Additionne a et b
     * @param a - premier paramètre
     * @param b - second paramètre
     * @return      */
    public int add(int a, int b) {
        return a + b ;
    }

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

        System.out.println("Hello world");
    }
}
```

… et la classe DateUtils.

Remarque : l'option **Escalate**, signifie : **la visibilité pourra être modifiée afin de garantir que les appels en cours vers des éléments délégués sont toujours Ok**.

Dans notre cas, la visibilité des méthodes n'a pas changé, car il n'y a pas eu besoin de le faire (leur usage ne l'a pas exigé). On peut cependant choisir une option spécifique de visibilité ;

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

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

3. Inlining = Inverse de Extraction

Consiste à remplacer un élément par le code dont il est fait :
- remplacer une variable par l'expression qui la constitue
- remplacer une méthode par son corps

a. Inline - élément

Exemple1 - TAF : Dans la méthode isNowBetween(), déléguer la variable now créée précédemment.

Étapes

- Clic-droit sur l'élément (la défiition, une référence à un élément)
- Déclencher l'option ***Refactor > Inline Variable*** **(Ctrl+Alt+N)**
- Choisir l'option souhaitée (insérer toutes les références et supprimer la variable, insérer toutes les références mais conserver la variable, insérer uniquement la référence sélectionnée et conserver la variable)
- Valider avec *Entrée*

Version finale :

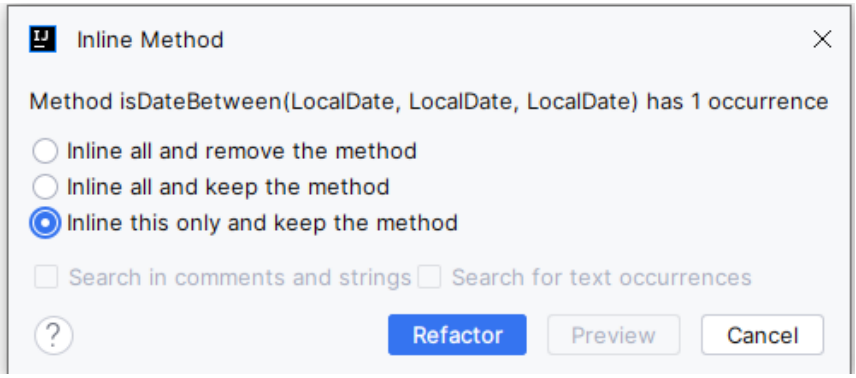
```
public static boolean isNowBetween(LocalDate startingDate, LocalDate endingDate) {
    return isDateBetween(startingDate, endingDate, LocalDate.now());
}
```

b. Inline - méthode

Exemple2 - TAF : Dans la méthode isNowBetween(), remplacer l'appel de la méthode isDateBetween() par son corps, sans toutefois supprimer la méthode isNowBetween () de la classe.

Étapes

- Clic-droit sur l'élément (la défiition, une référence à un élément)
- Déclencher l'option ***Refactor > Inline Method***
- Choisir l'option souhaitée (insérer toutes les références et supprimer la variable, insérer toutes les références mais conserver la variable, insérer uniquement la référence sélectionnée et conserver la variable)



- Taper sur *Entrée*

Version finale de la classe DateUtils :
Où l'appel à la méthode la variable now a été remplacée par son contenu dans la méthode isNowBetween(), mais la méthode `isDateBetween()` () n'a pas été supprimée.

```
public class DateUtils {

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

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

4. Déplacement d'éléments

Nous avons déjà vu comment déplacer des méthodes d'une classe, vers une nouvelle classe que l'on crée (§2.c -Extraction de classe).

a. Déplacement de méthode STATIQUE vers une classe existante

Contexte : Supposons que la classe SimpleClass possède une méthode supplémentaire isDateOutsider()

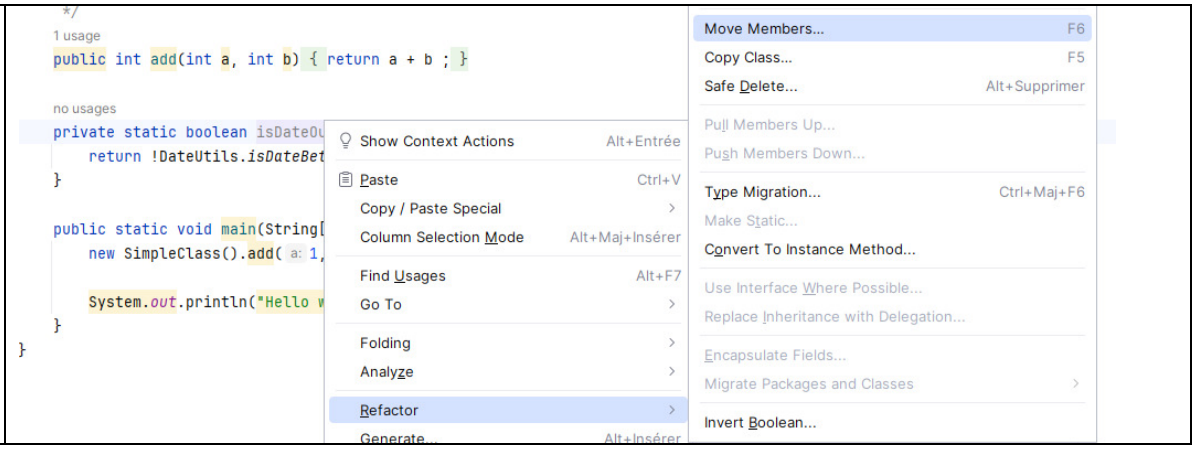
```
public SimpleClass() {
    }

    /**
     * Additionne a et b
     * @param a - premier paramètre
     * @param b - second paramètre
     * @return
     */
public int add(int a, int b) {
        return a + b ;
    }

private void isDateOutsider(LocalDate date, LocalDate startingDate, LocalDate
endingDate) {
        return !DateUtils.isDateBetween(date, startingDate, endingDate);
    }

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

        System.out.println("Hello world");
    }
}
```

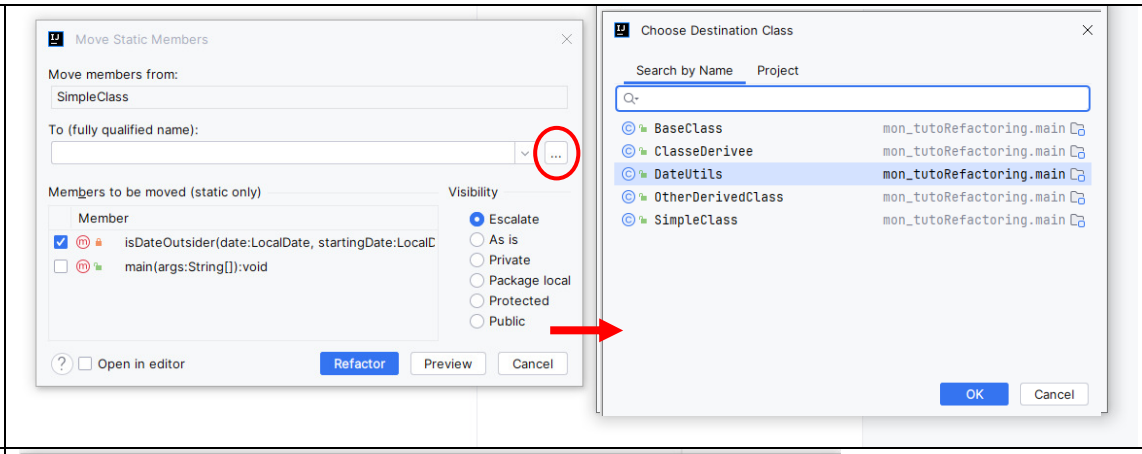
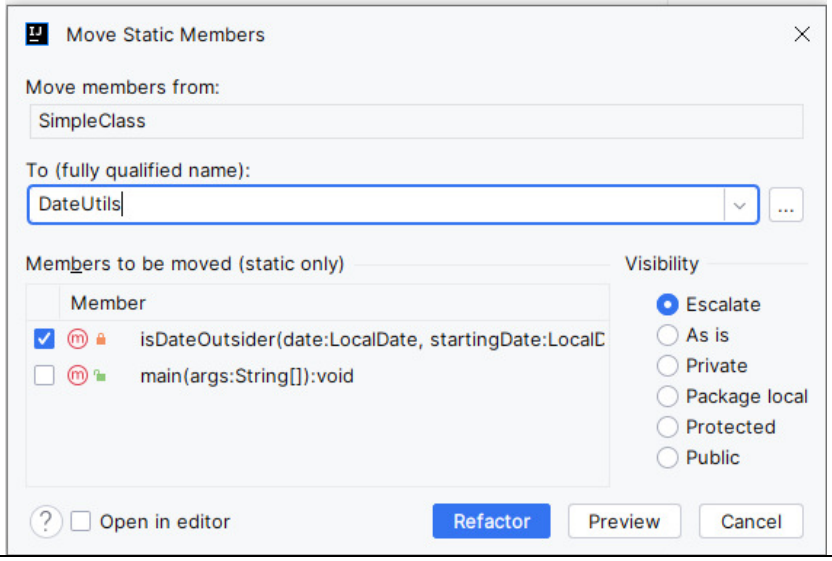
Exemple - TAF : Déplacer cette méthode dans la classe **DateUtils**.

Étapes

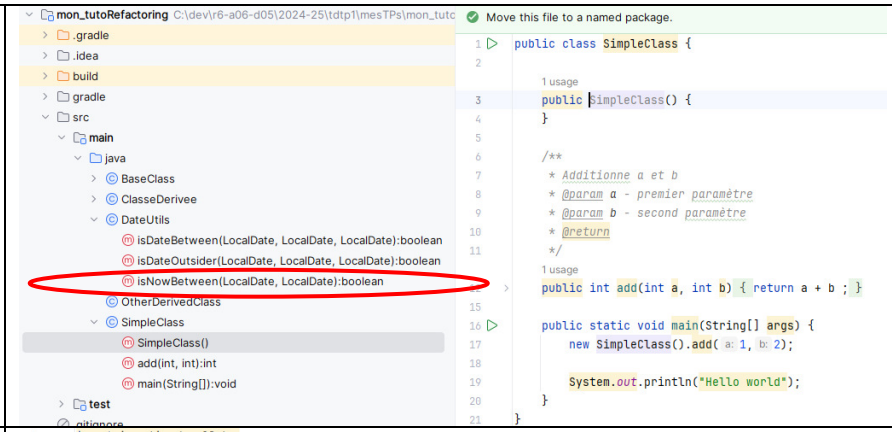
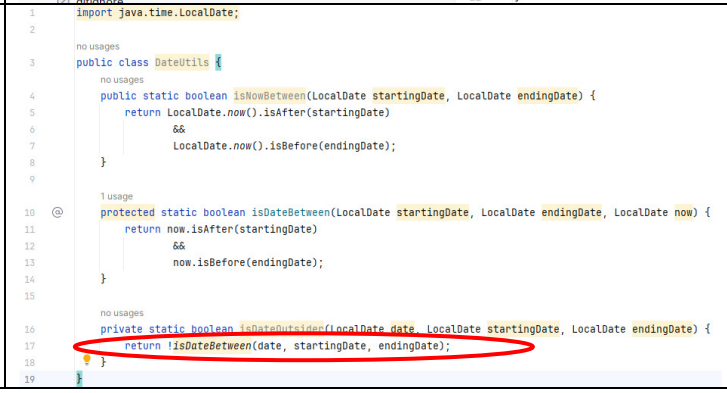
- Clic-droit sur l'élément (la définition, une référence à un élément)
- Déclencher l'option ***Refactor >Move Static Members***
- Choisir l'option souhaitée : le nom)
- Valider avec *Entrée*

Illustration

Refactor > Move Members	

Choisir la classe de destination et valider	
Valider	

Version finale :

La méthode a été transférée vers la classe DateUtils.	
Dans la méthode isDateOutside(), la référence à la classe DateUtils a disparu car elle n'est plus nécessaire.	

Rappel : l'exemple concerne le déplacement d'une méthode statique.

b. Déplacement de méthode d'INSTANCE vers une classe existante

Lorsque le déplacement concerne 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 que le programmeur ait le droit de la modifier.
Les options de la fenêtre Déplacer une méthode d'instance sont les suivantes :

- Select an Instance Parameter — On y choisit la classe vers laquelle la méthode sera déplacée
- Visibility —Sélection de la visibilité de la méthode dans sa nouvelle classe d'accueil.
- Select a name for the parameter - Choix du nom de l'instance de la classe d'origine.

La méthode sera alors déplacée et toutes ses instances seront mises à jour pour refléter la nouvelle classe.

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

5. Changement de signature d'une méthode

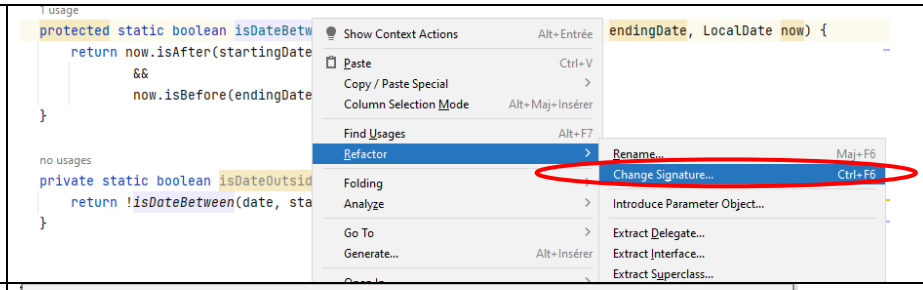
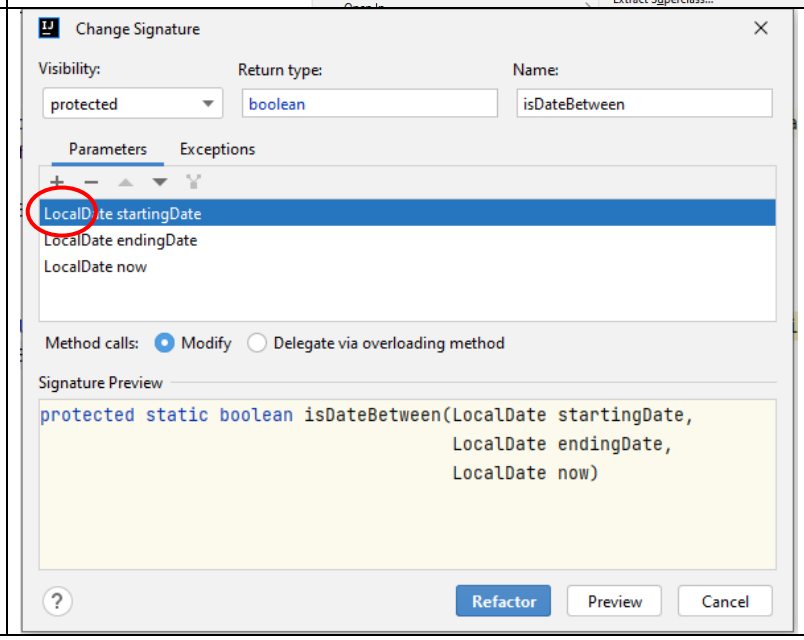
Il s'agit de modifier n'importe quel composant de la signature d'une méthode.

Exemple - TAF : Modifier l'implémentation de la méthode isDateBetween() pour préciser si les paramètres fournis (les dates limites) sont inclusifs ou exclusifs.
Pour ce faire, un paramètre booléen sera ajouté pour apporter cette précision :

Étapes

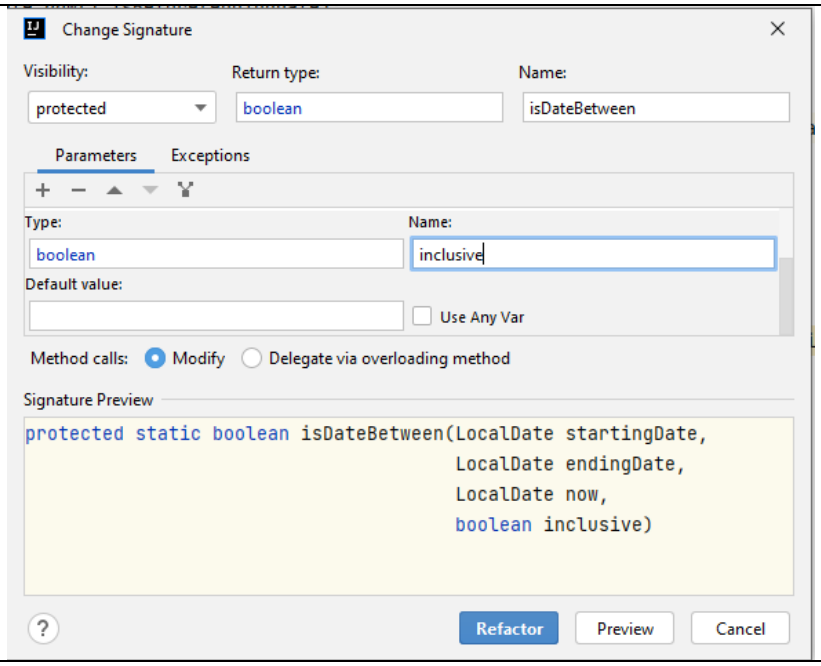
- Clic-droit sur l'élément (la définition, une référence à un élément)
- Déclencher l'option ***Refactor >Change signature***
- Apporter les modifications souhaitées à la signature de la méthode
- Valider avec *Entrée*

Illustration

Sélection méthode puis ***Refactor >Change signature***	
Ajout d'un paramètre	

Renseigner le paramètre : nom, type, éventuellement valeur par défaut
La nouvelle signature est prévisualisée.

Remarque - Option **Delegate via overloading method :**
En cochant cette option, on aurait créé une autre méthode (par surcharge) avec le paramètre souhaité, au lieu de modifier la méthode actuelle.



Reste à faire :

1) Modifier le corps de la méthode pour prendre en compte le nouveau paramètre
2) Recompiler le code pour identifier l'emplacement de chaque appel de cette méthode afin de le corriger pour y ajouter un nouveau paramètre true/false

Version finale

```
protected static boolean isDateBetween(LocalDate startingDate, LocalDate endingDate, LocalDate
now, boolean inclusive) {

    LocalDate startingDay;  // sera le paramètre startingDate ou le lendemain du paramètre
    LocalDate endingDay;    // sera le paramètre endingDate ou le la veille du paramètre

    if (inclusive) {
        startingDay = startingDate.minusDays(1);    // la veille
        endingDay = startingDate.plusDays(1);          // le lendemain
    }
    else {
        startingDay = startingDate;
        endingDay = startingDate;
    }

    return now.isAfter(startingDay)
                &&
                now.isBefore(endingDay);
    /* Pour info :
    *     date1.isBefore(date1) retourne FAUX   // Before strict
    *     date1.isAfter(date1) retourne FAUX    // After strict
    *  Donc, pour que l'évaluation soit inclusive,
    *  il faut calculer After(laVeille du paramètre staringDate) et Before(le lendemain du
paramètre endingDate) */
}
```

6. Déplacement d'éléments dans un graphe d'héritage

Le déplacement peut se faire :
- d'une classe héritée vers la classe mère
- de la classe mère vers une classe héritée

a. Déplacement vers le haut : de classe dérivée vers classe mère

Exemple d'illustration

Considérons 3 classes : BaseClass et 2 classes dérivées : ClasseDerivee et OtherDerivedClass

```
public class BaseClass {
    public BaseClass() {
    }
}

public class ClasseDerivee extends BaseClass {

    private int member;

    public static void main(String[] args) {
        ClasseDerivee subject = new ClasseDerivee();
        System.out.println("Doublement 21. Résultat = " + subject.doubleValue(21));
    }

    /**
     * @param number  - le nombre à doubler
     * @return  - 2*number
     */
    private int doubleValue(int number) {
        return number + number;
    }
}

public class OtherDerivedClass extends BaseClass {
}
```

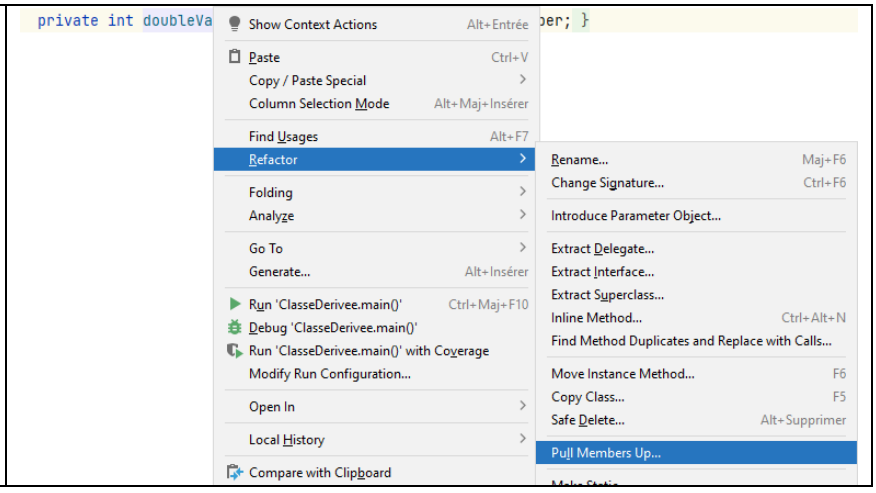
TAF – Extraire la méthode doubleValue() (de ClasseDerivee) dans la classe mère (= extraire la méthode de la classe fille et la placer dans la classe mère)

Étapes

- Clic-droit sur le membre de la classe fille à extraire
- Déclencher l'option ***Refactor > Pull Member Up***
- Renseigner les options supplémentaires
- Valider avec *Refactor*

Illustration

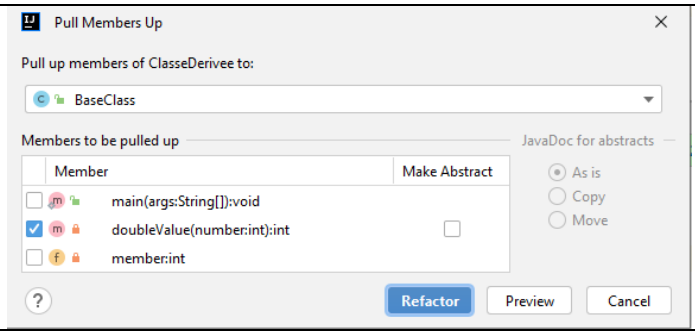
Pas de raccourci clavier disponible pour cette action.



Options supplémentaires :

- Possibilité de sélectionner d'autres membres pour les extraire en même temps que la méthode initialement choisie

- Seules les méthodes ont une case « Make abstract ».



Version finale

```
Move this file to a named package.                                SonarLint: Show rule d
2 usages  2 inheritors
public class BaseClass {
    2 usages
    public BaseClass() {
    }

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

README.md | ClasseDerivee.java | DateUtils.java | SimpleClass.java

```
Move this file to a named package.                                SonarLint: Show rule d
2 usages
public class ClasseDerivee extends BaseClass{
    no usages
    private int member;
    no usages
    public static void main(String[] args) {
        ClasseDerivee subject = new ClasseDerivee();
        System.out.println("Doublement 21. Résultat = " + subject.doubleValue( number: 21));
    }
}
```

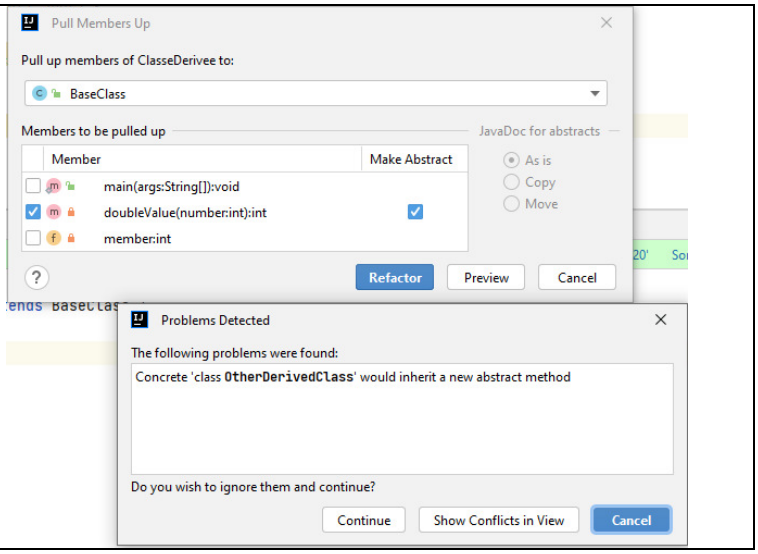
OtherDerivedClass.java

```
Move this file to a named package.                                SonarLint: Show rule d
no usages
public class OtherDerivedClass extends BaseClass {
}
```

- La méthode doubleValue() a migré vers la classe mère
- sa visibilité passe de **privée** à **protégée** afin que la classe dérivée puisse toujours l'utiliser.

Remarque : Options supplémentaires

Si elle est cochée, la méthode située dans la classe mère sera **abstraite**. La méthode concrète restera dans la classe dérivée, accompagnée de l'annotation *@Override*. **Il manquera alors l'implémentation aux autres classes dérivées.**



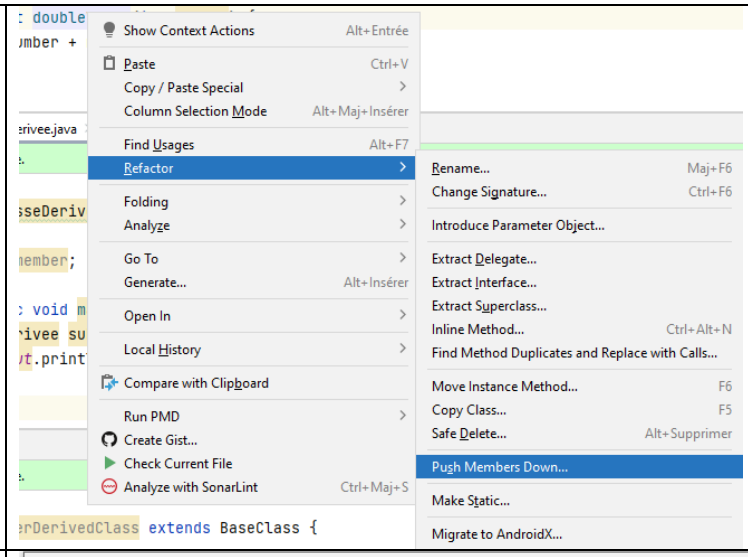
b. Déplacement vers le bas : de classe mère vers classe dérivée

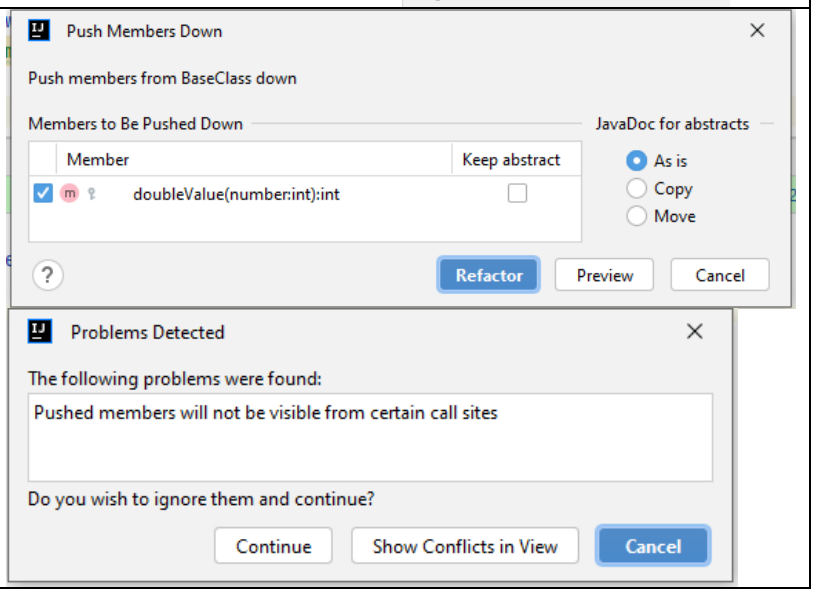
C'est l'action opposée de ***Pull Membres Up***.

Étapes

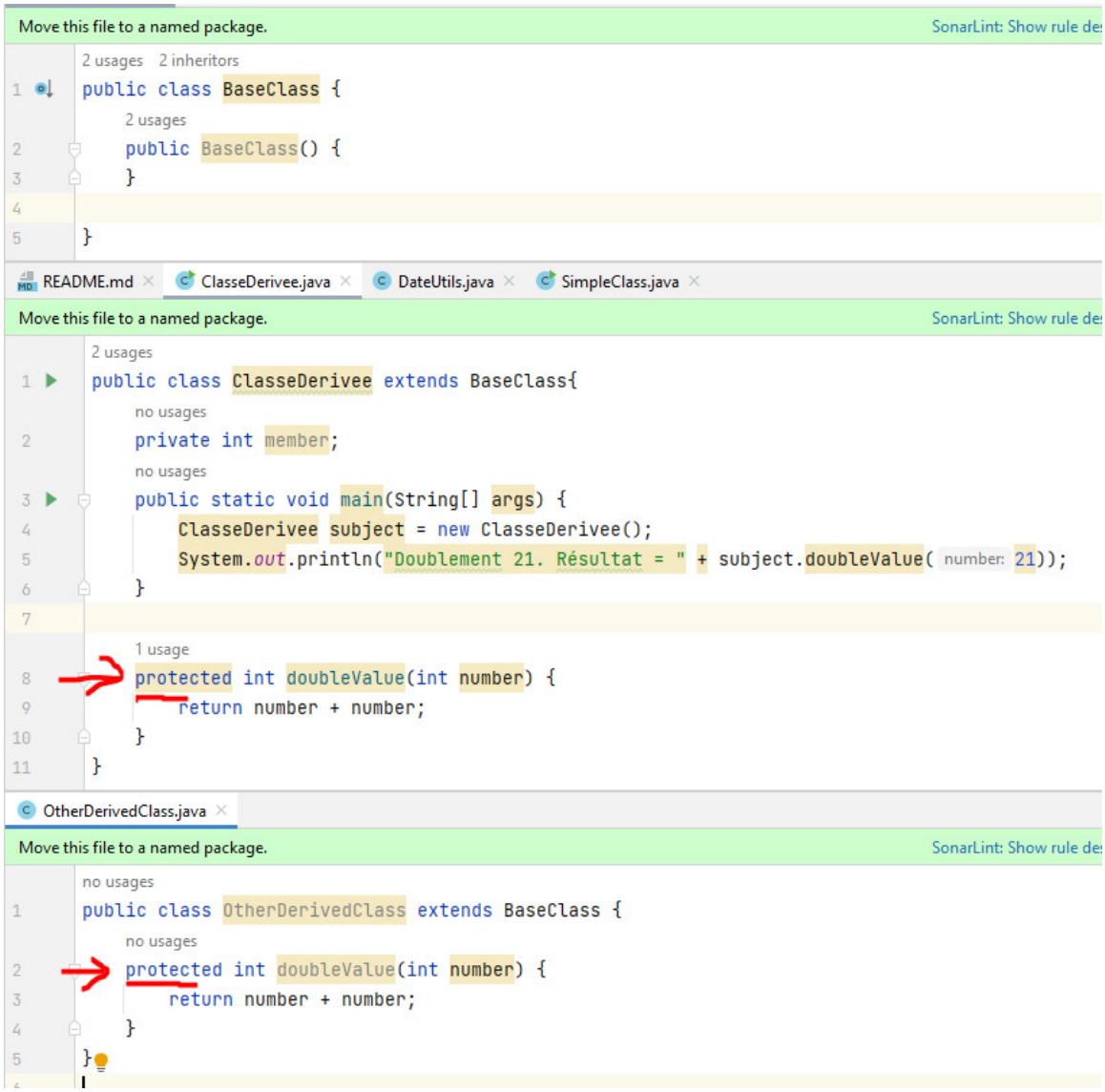
- Clic-droit sur le membre de la classe mère à extraire
- Déclencher l'option ***Refactor > Pull Member Down***
- Renseigner les options supplémentaires
- Valider avec *Refactor*

Illustration

<table>
<tr>
<td>Pas de raccourci clavier disponible pour cette action.</td>
<td> 
</td>
</tr>
<tr>
<td>Les membres sélectionnés seront poussés dans toutes les classes dérivées.

Avertissement
IntelliJ informe que les membres poussés ne seront plus visibles depuis certaines classes (qui étaient en relation avec la classe mère.)</td>
<td> 
</td>
</tr>
</table>

Version Finale



```
Move this file to a named package.                                SonarLint: Show rule des
2 usages  2 inheritors
public class BaseClass {
    2 usages
    public BaseClass() {
    }

}
```

README.md | ClasseDerivee.java | DateUtils.java | SimpleClass.java

```
Move this file to a named package.                                SonarLint: Show rule des
2 usages
public class ClasseDerivee extends BaseClass{
    no usages
    private int member;
    no usages
    public static void main(String[] args) {
        ClasseDerivee subject = new ClasseDerivee();
        System.out.println("Doublement 21. Résultat = " + subject.doubleValue( number: 21));
    }

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

OtherDerivedClass.java

```
Move this file to a named package.                                SonarLint: Show rule des
no usages
public class OtherDerivedClass extends BaseClass {
    no usages
    protected int doubleValue(int number) {
        return number + number;
    }
}
```

- La méthode est descendue dans toutes les classes dérivées.
- Elle a conservé la visibilité ***protected*** qu'elle a acquise dans la classe mère.

Remarque : Options supplémentaires

- Possibilité de déplacer d'autres méthodes que elle initialement choisie - Si la case « Keep abstract » est cochée, l'option laissera une méthode abstraite dans la classe de base. Dans ce cas, des implémentations seront placées dans chaque classe dérivée avec la mention ***@Override***