

R2.03 : Tests unitaires & Débogueur

*Les tests permettent de vérifier et valider le comportement de l'application.
Le déboguer permet de trouver plus facilement les erreurs
et de les corriger plus rapidement, sûrement et proprement.
Ces deux outils (tests et débogueur) permettent donc d'assurer la qualité de code.*

Arnaud LEMAIRE
@Lilobase

When part of your code doesn't have tests
Traduire le Tweet



Partie 1 Retour vers les tests unitaires



1. Récupération du projet testspartice sur Github

Sur le remote :

- ☐ Rendez-vous à l'adresse suivante : <https://github.com/iblasquez/testspartice>
- ☐ Copiez l'adresse **https** de ce dépôt.

En local en ligne de commande :

- ☐ Ouvrez **Git Bash** et placez-vous dans le **workspace javabut1**
- ☐ Clonez le dépôt Github **testspartice** via la **ligne de commande git clone** suivie de l'adresse **https**
- ☐ Depuis Git Bash, rendez-vous dans le répertoire **testspartice** (**cd testspartice**) puis déconnectez ce dépôt local du remote Github en utilisant la commande : **git remote rm origin**

En local dans l'IDE :

- ☐ Lancez Eclipse avec le workspace **javabut1**
- ☐ Importez un **projet Maven existant**, puis sélectionnez le projet **testspartice** de manière à bien voir le **pom.xml** dans le cadre **Projects**.

2. Familiariser-vous avec ce projet existant ...

- ☐ Commencez par jeter un **petit coup d'œil dans la vue Package Explorer** pour évaluer l'importance du projet et se familiariser avec le nom des classes et donc la terminologie du contexte métier 😊
- ☐ Consultez l'**historique des versions** (**Team→Show In History**)
- ☐ **Promenez-vous dans le code**, juste en tant qu'observateur, pour vous l'approprier...
- ☐ **Lancez les jeux d'essais et tests** (s'il y en a...)
- ☐ Lancez **SonarLint** et recherchez d'éventuels **//TODO** (**window → Show view → Tasks**)

Vous l'avez sûrement compris en vous familiarisant avec ce projet, votre travail d'aujourd'hui va consister à écrire des tests unitaires pour couvrir le code des classes existantes de ce projet qui pour le moment n'ont ni jeux d'essais, ni tests automatisés ... 😊

Vous pouvez directement travailler dans la branche **master**,
puisque d'une part vous travaillez tout seul sur ce projet
et d'autre part, vous allez écrire chaque classe de tests l'une après l'autre et commiterez dès lors qu'une classe de tests couvrira le code de la classe de production correspondante.
Notons qu'il est préférable au cours d'un développement
d'écrire les tests unitaires en même temps que le code de production
pour vérifier au fur et à mesure les comportements implémentés.

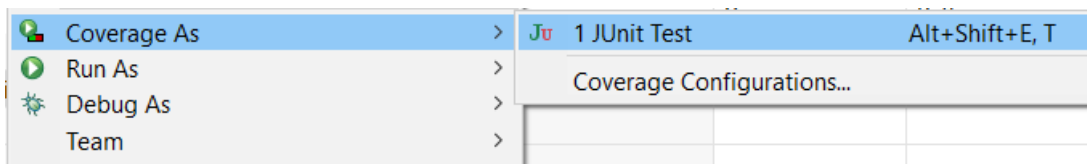
3. Première tâche : Tests unitaires sur la classe **StringUtils**

❑ Couvrir **src/main/StringUtils** en écrivant des tests unitaires dans **src/test/StringUtilsTest**

- La classe de tests **StringUtilsTest** a déjà été créée.
- A vous de jouer pour **couvrir toutes les lignes de code** des méthodes de la classe de production **StringUtils**
 - Commencez par écrire un test sur la méthode **isPalindrome** et regarder la couverture de code...
Qu'en pensez-vous ?
 - N'oubliez pas de tester des exemples pairs et des exemples impairs 😊



Rappel : Pour lancer la couverture de code par les tests, placez-vous sur une classe de tests, puis à l'aide d'un clic droit, sélectionnez **Coverage As→ JUnit Test**



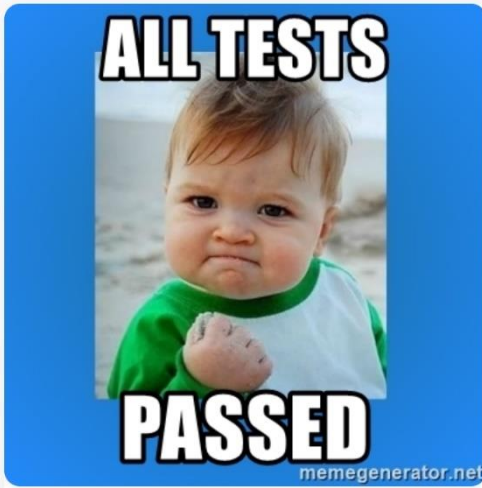
❑ Commitez la classe de test **StringUtilsTest** avec un message du type : **add StringUtilsTest**

Optionnel : Si vous avez fini le TP en avance ou si vous avez envie de coder un peu plus chez vous 😊
Avec le code actuel, les exemples suivants ne sont pas des palindromes :

- "no lemon, no melon"
- "Madam, I'm Adam."
- "I did, did I?"
- "Eva, can I see bees in a cave?"
- "Don't nod"

En effet, la méthode **isPalindrome** ne traite ni les espaces, ni les majuscules, ni la ponctuation.
Voici donc 3 améliorations possibles à apporter à la méthode **isPalindrome** en vérifiant l'implémentation avec les tests correspondants bien sûr 😊 !

4. Deuxième tâche : Tests unitaires sur la classe **MathUtils**



❑ Couvrir `src/main/MathUtils` en écrivant des tests unitaires dans `src/test/MathUtilsTest`

→ Commencez par créer la classe de tests **MathUtilsTest** (au bon endroit)

→ A vous de jouer pour couvrir toute la méthode **isDivisibleBy** de la classe de **MathUtils**

❑ Commitez la classe de test **MathUtilsTest** avec un message du type : `add MathUtilsTest`

5. Troisième tâche : Tests unitaires sur la classe **DateUtils**

❑ Pour vous aider à couvrir le code de la classe `src/main/DateUtils`, un squelette de la classe `src/test/DateUtilsTest` a été écrit avec les méthodes de test suivantes ...

```
should_be_a_PreJulianYear_before_1582_not_a_leap_year
should_be_not_a_PreJulianYear_after_1582 ⇒ tester seulement les années 1582 et 1852
should_be_a_leap_year_when_divisible_by_400
should_be_a_leap_year_when_divisible_by_4_but_not_by_100
should_not_be_a_leap_year_when_divisible_by_100_but_not_by_400
should_not_be_a_leap_year
```

... à compléter avec les exemples d'années suivantes

1163, 1500, 1515, 1582, 1600, 1700, 1789, 1800, 1852, 1900, 1955, 1968, 1985, 1992, 2000, 2001, 2100, 2016, 2020, 2022, 2042, 2048, 2200, 2300, 2400, 2500, 2600

Pour l'instant, répartissez correctement toutes ces années dans les méthodes de tests , écrivez autant d'assertions que d'années et faites passer les tests AU VERT !!!

❑ Commitez la classe de test **DateUtilsTest** avec un message du type :
`add examples in DateUtilsTest`

Leap Year
(2012, 2016, 2020)

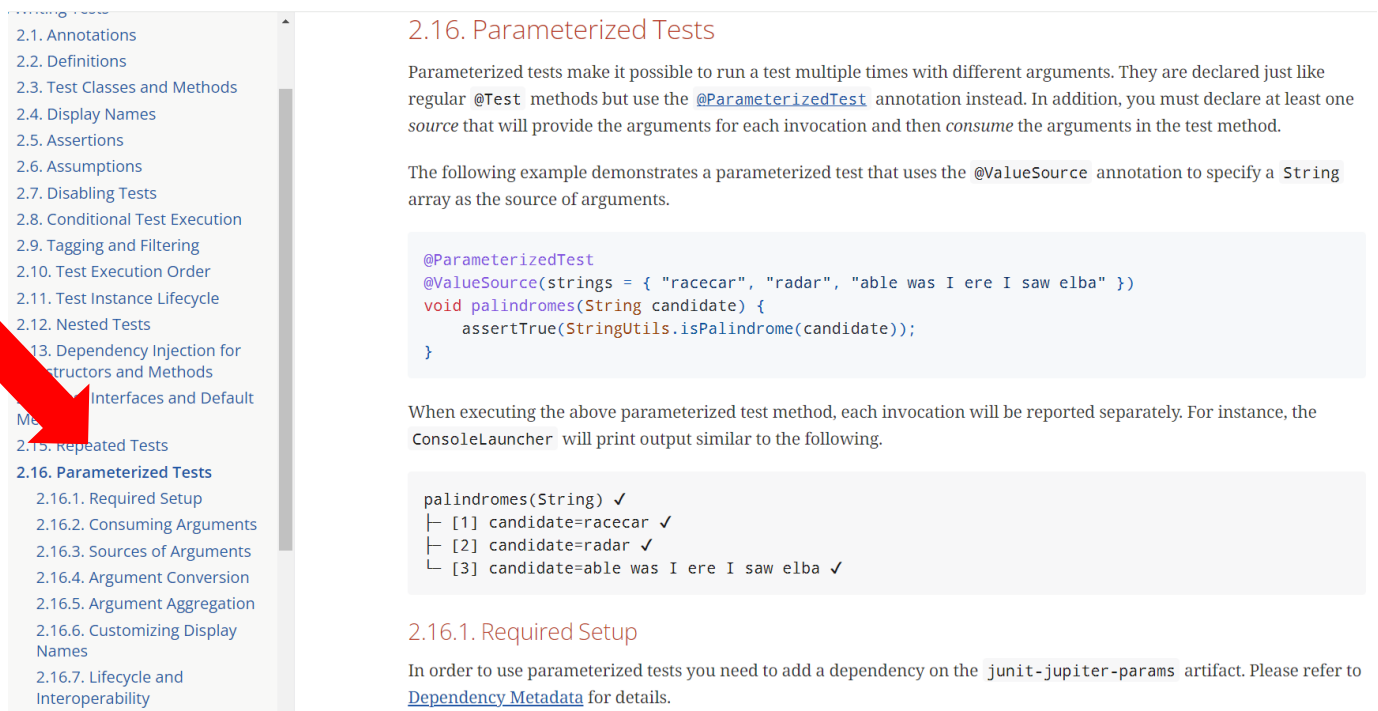


6. Premier pas avec les tests paramétrés

Que ce soit pour la méthode `isPalindrome` ou `isLeapYear`, les tests écrits sont (un peu) répétitifs à écrire. Y aurait-il une manière plus simple de jouer « un tableau d'exemples » avec un seul `assert` ? Oui !!! Les frameworks de tests unitaires permettent cela et c'est ce qu'on appelle des **tests paramétrés**. Pour savoir, comment mettre en œuvre les tests paramétrés avec JUnit, rien de tel qu'un petit tour dans la documentation.

❑ a. Toujours commencer par jeter un petit coup d'œil dans la documentation 😊

Rendez-vous donc sur le site : <https://junit.org/junit5/> et cliquez sur **User Guide**
Dépliez le menu **2. Writing Test** pour voir apparaître la rubrique **2.16 Parametrized Tests**



2.16. Parameterized Tests

Parameterized tests make it possible to run a test multiple times with different arguments. They are declared just like regular `@Test` methods but use the `@ParameterizedTest` annotation instead. In addition, you must declare at least one *source* that will provide the arguments for each invocation and then *consume* the arguments in the test method.

The following example demonstrates a parameterized test that uses the `@ValueSource` annotation to specify a `String` array as the source of arguments.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```

When executing the above parameterized test method, each invocation will be reported separately. For instance, the `ConsoleLauncher` will print output similar to the following.

```
palindromes(String) ✓
├─ [1] candidate=racecar ✓
├─ [2] candidate=radar ✓
└─ [3] candidate=able was I ere I saw elba ✓
```

2.16.1. Required Setup

In order to use parameterized tests you need to add a dependency on the `junit-jupiter-params` artifact. Please refer to [Dependency Metadata](#) for details.

Parcourir rapidement cette rubrique pour vous familiariser avec le notion de **tests paramétrés sous JUnit** et voir un peu ce que nous propose les options qui nous sont offertes.

Pour la suite, nous nous focaliserons uniquement sur quelques parties de cette rubrique...

❑ b. Création d'une branche `experiment-parametrized-tests`

Pour pouvoir jouer tranquillement avec les tests paramétrés sans perdre les fichiers de tests initiaux, créez à partir du dernier commit une branche que vous appellerez **experiment-parametrized-tests** et placez-vous sur cette branche.

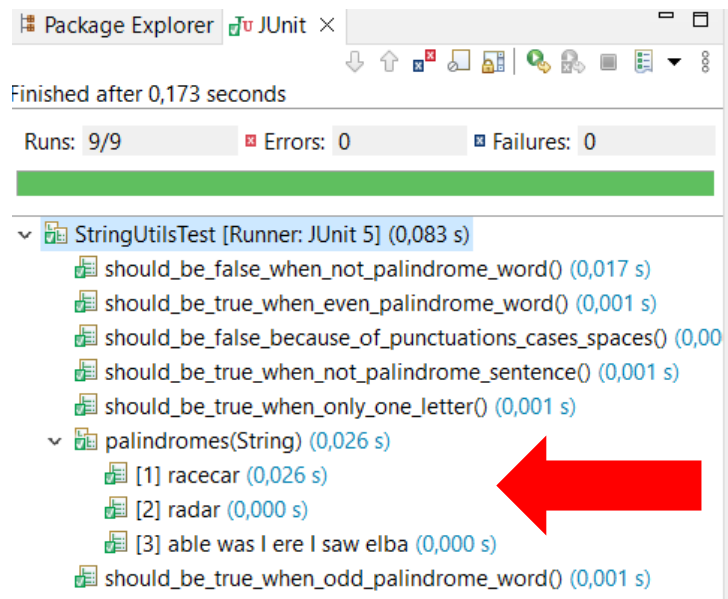
```
> testspartice [testspartice experiment-parametrized-tests]
  > src/main/java
    > testspartice
      > DateUtils.java
```

❑ c. Une première approche des tests paramétrés sur les palindromes.

⇒ Exécutez un code exemple fonctionnel implémentant un test paramétré...

Le premier exemple de la rubrique **2.16 Parametrized Tests** illustre la mise en œuvre d'un test paramétré sur une méthode `isPalindrome` d'une classe `StringUtils`.

Quelle chance ! Vous pouvez directement **implémenter** par un copier/coller ce code dans votre classe `StringUtilsTest`, corriger les **erreurs de compilations** (c-a-d ajouter les *bons imports*), et **exécuter** ce fichier test avec **Run As → JUnit Test** de manière à visualiser dans la **vue JUnit** un affichage similaire au `ConsoleLauncher` présenté dans de la documentation :



⇒ Ecrire un test paramétré à partir du code exemple ...

- Implémentez maintenant dans la classe **StringUtilsTest** une méthode de test `no_palindromes` sous forme de tests paramétrés.
- Exécutez et lancez cette méthode pour vérifier sa « bonne » implémentation

⇒ Répartir tous vos exemples présents dans ce fichier tests dans une des deux méthodes :
palindromes, no_palindromes

⇒ Supprimez toutes les méthodes de tests précédentes de manière à ne garder que ces deux méthodes et donc uniquement deux `assert` avec tous les exemples précédents présents dans l'annotation `@ValueSource` de l'une ou l'autre des deux méthodes 😊.

⇒ **Exécutez** ce fichier avec **Run As** → **JUnit Test** de manière à visualiser dans la **vue JUnit** tous les tests AU VERT !!!

⇒ **Commitez** avec un message du type : `add parametrized tests for palindromes`

❑ c. Et maintenant paramétrisons les tests années bissextiles ...

⇒ Dans la classe `DateUtilsTests`, transformez chaque méthode de test en test paramétré ! ... pour au final, garder tous les exemples de dates, mais ne voir apparaître plus que 7 `assert` dans le code de ce fichier 😊

⇒ **Exécutez** ce fichier avec **Run As** → **JUnit Test** de manière à visualiser dans la **vue JUnit** tous les tests AU VERT !!!

❑ c. Terminons par paramétrer les tests de `MathUtilsTests`

⇒ Un peu de documentation avant de se lancer

La méthode à tester `isDivisibleby` de `MathsUtils` possède 2 paramètres d'entrée.

L'annotation `@ValueSource` ne peut être utilisée que dans le cas d'une méthode à 1 seul paramètre d'entrée.

L'annotation `@CsvSource` doit être utilisée si la méthode à tester à plusieurs paramètres d'entrée.

Consultez la sous rubrique `@CsvSource` de **2.16.3 Sources of Arguments** pour tout savoir sur l'utilisation de cette annotation

(<https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources-CsvSource>)

⇒ Dans la classe `MathUtilsTests`, transformez chaque méthode de test en test paramétré !... pour au final, garder tous les exemples, mais ne voir apparaître plus que 2 `assert` dans le code de ce fichier 😊.

Implémentez :

- Un test paramétré avec un simple `@CsvSource`
- Un autre test paramétré avec `@CsvSource(useHeadersInDisplayName = true ...`

⇒ **Exécutez** le fichier de tests avec **Run As** → **JUnit Test** de manière à visualiser dans la **vue JUnit** tous les tests AU VERT et dépliez les méthodes de tests pour voir comment le paramétrage de `@CsvSource` peut modifier l'affichage dans la vue JUnit.

⇒ **Commitez** en une fois les deux dernières classes de tests avec un message du type :
`add parametrized tests for leap year`

7. Rendre l'affichage de la vue JUnit plus convivial avec l'annotation @DisplayName

⇒ un peu de documentation sur l'annotation @DisplayName

- Rendez-vous dans la rubrique **2.16.6 Customizing Display Names** pour découvrir quelques exemples qui permettent d'améliorer l'affichage de rendu des tests unitaires dans la vue JUnit

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-display-names>

- En effet, le framework JUnit5 propose l'annotation **@DisplayName**. Vous pouvez en savoir plus sur les possibilités offertes par cette annotation en allant dans la rubrique **2.4 DisplayName**

<https://junit.org/junit5/docs/current/user-guide/#writing-tests-display-names>

Nous pouvons remarquer entre autres :

- ➔ que sans l'annotation @DisplayName, c'est le nom de la méthode qui s'affiche dans la vue JUnit, c'est pour cela qu'on demande d'avoir un nom de méthode de test très explicite.
- ➔ qu'avec l'annotation @DisplayName, c'est le paramètre de cette annotation qui sera affiché dans la vue JUnit. Le nom de la méthode de test peut donc être plus courte, tout en restant explicite sur l'intention de la méthode de test (c-a-d ce qu'elle est en train de tester !)

⇒ Jouez avec l'annotation @DisplayName

A partir des exemples et de la documentation des rubriques **2.4 DisplayName** et **2.16.6 Customizing Display Names** ajoutez des annotations @DisplayName dans vos méthodes de tests pour vous familiariser avec les services offerts par cette annotation (Rappel : une annotation Java est en fait une classe Java).

⇒ Commitez vos modifications avec un messages du type : **add DisplayName annotation**

8. Tests classiques vs Tests paramétrés

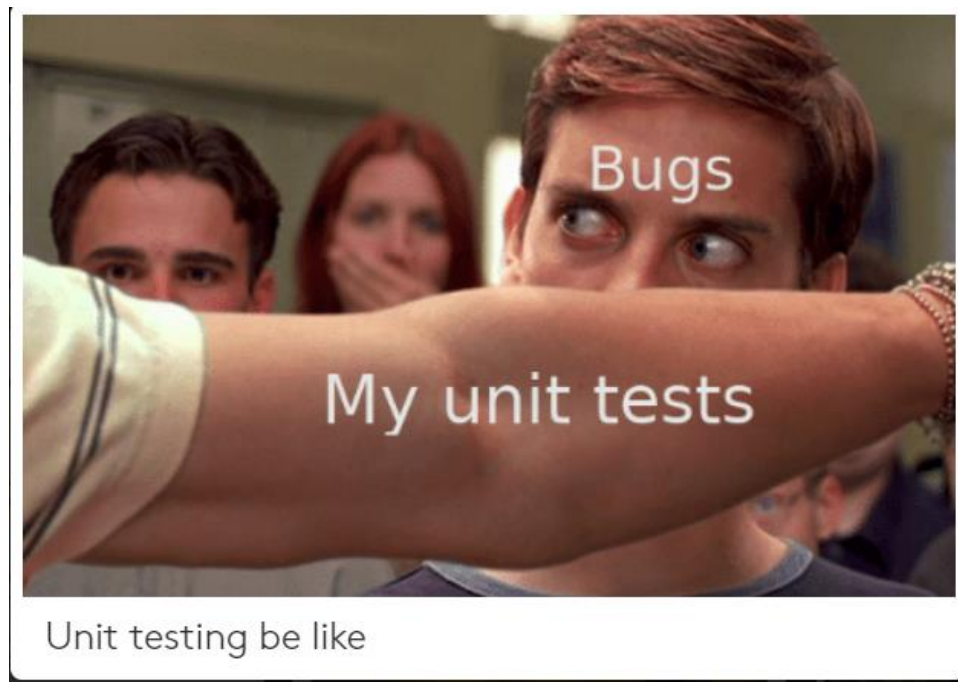
Affichez par exemple dans l'éditeur de code la classe **StringUtilsTest**.

En *switchant* de la branche **master** à la branche **experiment-parametrized-test** (et vice versa grâce à **Team → Switch to**), visualisez la différence d'implémentation entre les tests classiques et les tests paramétrés.

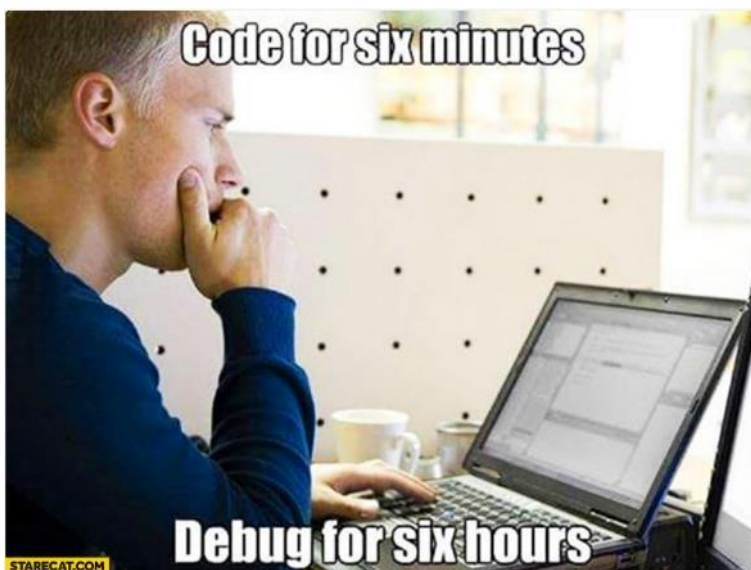
Les méthodes proposées dans cet exercice se prêtent bien aux tests paramétrés, mais ce n'est pas toujours le cas

Nous avons un peu exagéré ici sur le nombre d'exemples (notamment pour tester les années potentiellement bissextiles...), mais c'est un travail pédagogique 😊

Le processus de test consiste à exécuter un programme dans l'intention de détecter des erreurs.
(Myers, Sandler, 2004)



Les tests permettent donc de détecter les erreurs, mais une fois détectées, il faut corriger ces erreurs. Et ce n'est pas toujours évident... C'est pour cela qu'il est parfois utile au moment de la correction des erreurs de faire appel à un débogueur, plutôt que de parsemer des `System.out.println` un peu partout dans le code 😊



Écrire des print pour débbuguer, c'est la vie qu'on a décidé d'mener 🤪



Partie 2 A la découverte du débbuger

Quand mon alert("prouit") s'affiche pendant la démo client

effello, commit du 4 Fév 2022



Lorsque j'ai besoin de me rafraîchir la mémoire sur des notions Java, j'utilise régulièrement deux sites :

<https://www.baeldung.com/> et <https://www.vogella.com/tutorials/>

Pour découvrir comment utiliser le débbuger sous Eclipse, je vous propose de réaliser deux tutoriels : un sur chaque site.

Tutoriel n°1 : <https://www.baeldung.com/eclipse-debugging>

Tutoriel n°2 : <https://www.vogella.com/tutorials/EclipseDebugging/article.html>

... le tout en ayant le site de JM Doudoux sous la main...

<https://www.jmdoudoux.fr/java/dejae/chap008.htm>

Quand je suis en mode pas à pas et que je tombe sur le bug que j'essaie de corriger

Les Joies du Code, commit du 21 Mar 2022



Pour voir le gifs animés :

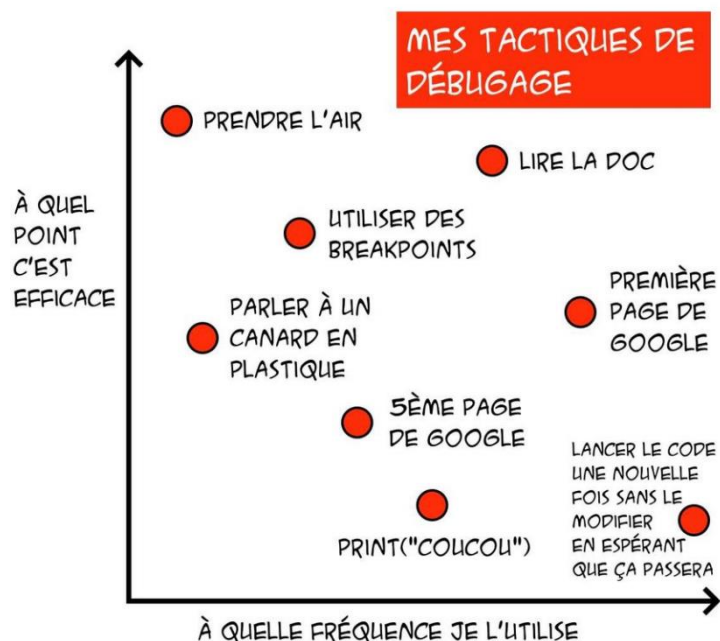
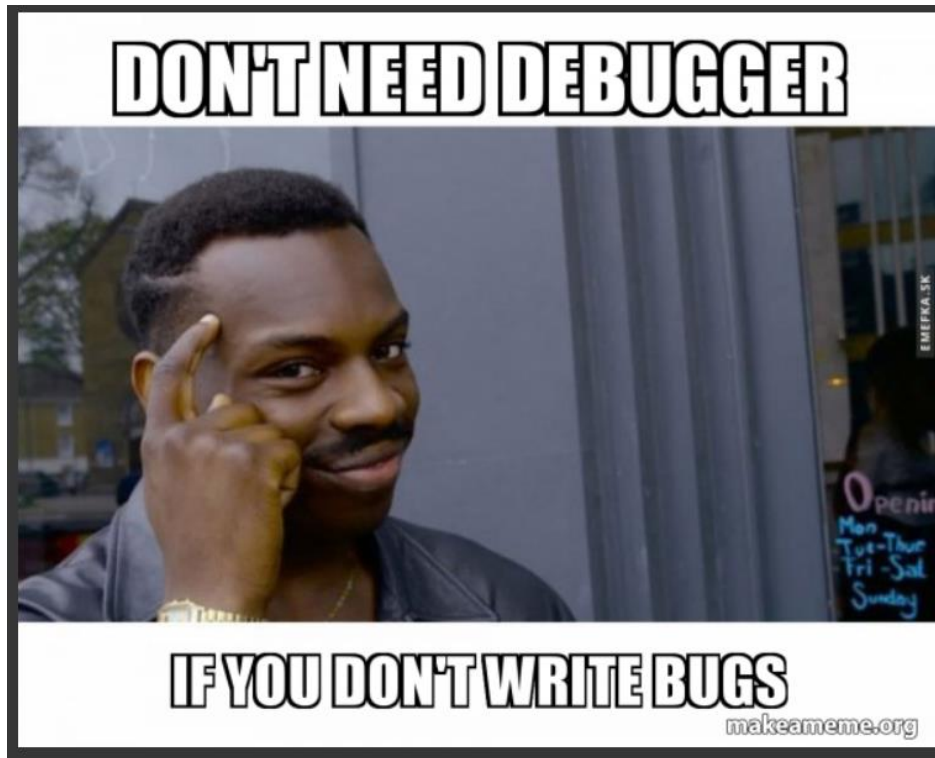
<https://unil.im/debugpas> et

<https://unil.im/breakpoint>

Quand j'ai positionné mon breakpoint au mauvais endroit

Les Joies du Code, commit du 8 Mar 2022





<https://twitter.com/lesjoiesducode/status/1533007808009994241>