

Premiers pas en Java à l'aide de l'IDE Eclipse

Cette année, vous utiliserez l'environnement **Eclipse** pour implémenter vos projets en java.
Eclipse est un IDE Open Source (IDE : Environnement de Développement Intégré (EDI))

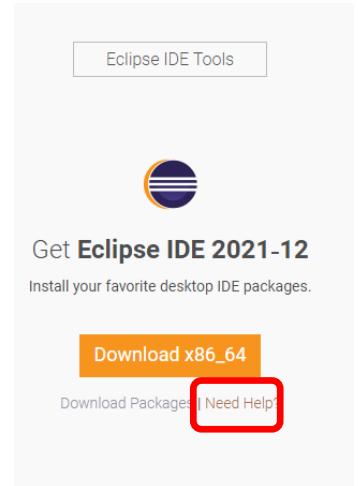
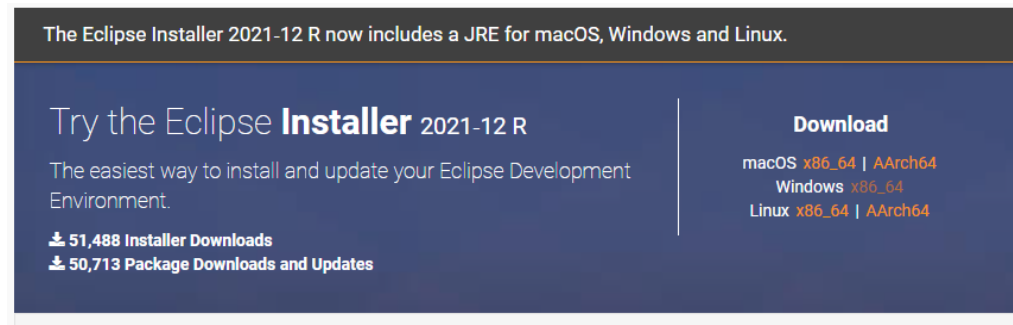
1. Installation d'Eclipse

Pour installer Eclipse sur votre portable, rendez-vous sur le site : <https://www.eclipse.org/>
Cliquez sur le bouton orange **Download** en haut à droite :



Cliquez sur **Need Help ?** Vous permettra ensuite d'ouvrir un tutoriel pour vous expliquer comment installer Eclipse.

Suivant le système d'exploitation présent sur votre machine, choisissez le bon installateur qui peut être téléchargé depuis le haut de cette page :



Une fois téléchargé cliquez ensuite sur l'installateur et suivez le tutoriel **Need Help ?**
(<https://www.eclipse.org/downloads/packages/installer>) :

- En choisissant comme package : **Eclipse IDE for Java Developers**
- Pour l'**installation folder**, je vous conseille de créer un répertoire Eclipse au préalable de manière à ce que votre chemin ressemble à :

Installation Folder

C:\Users\Isabelle\eclipse\java-2021-12



Il ne vous reste plus qu'à lancer Eclipse en cliquant sur **LAUNCH !**

2. Lancement d'Eclipse

Au lancement, Eclipse vous demande de renseigner le chemin de votre dossier de travail (**Workspace**) où seront rangés par défaut vos projets.

Tapez le chemin où vous souhaitez ranger vos votre TP, puis appuyez sur le bouton **Launch**.

Remarque :

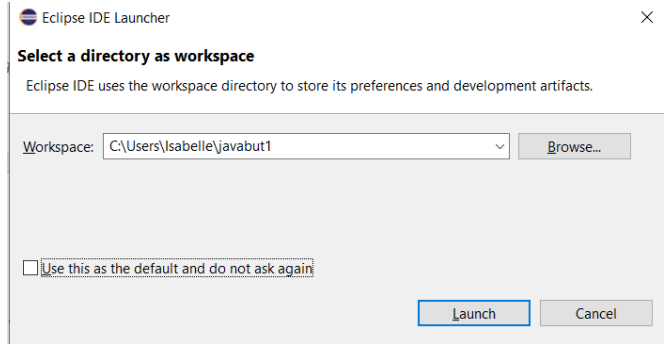
Dans un **workspace**, vous pourrez créer plusieurs projets java, mais rien ne vous empêche de créer autant de **workspaces** que vous le souhaitez.

Une bonne pratique serait d'ailleurs que vous créiez un **workspace** dédié à la **Sae** le moment venu 😊

Une page d'accueil présentant les fonctionnalités d'Eclipse est alors affichée.

Cliquez sur la croix pour fermer l'onglet **Welcome**

Vous voilà arrivés dans votre IDE préféré 😊



3. Création d'un nouveau projet Java (projet simple)

Pour créer un **projet**, choisissez

File → New

→ Java Project

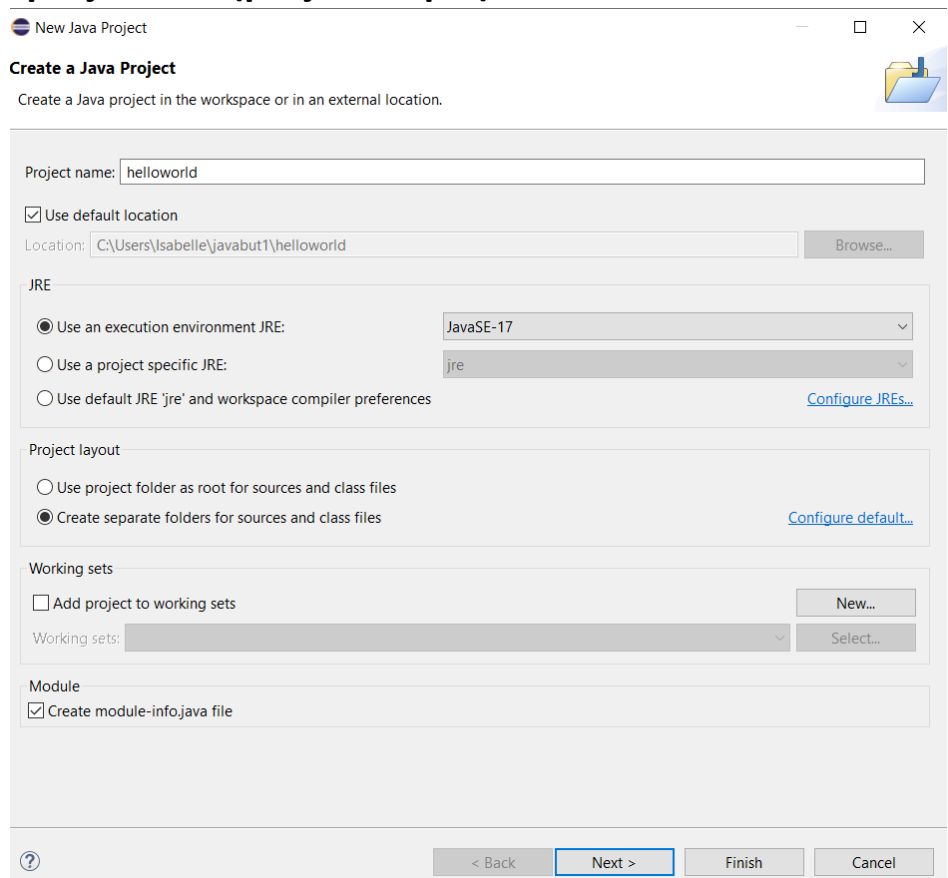
L'assistant *New Java Project* permet de choisir le nom du projet, le dossier dans lequel il sera enregistré, la version du JDK avec lequel il est compatible ainsi que les sous-dossiers où seront rangés les fichiers source `.java` et les fichiers `.class`.

Pour cette première approche, vous avez juste besoin de remplir le **nom** du projet : **helloworld** (il est de coutume en informatique de toujours commencer l'apprentissage d'un nouveau langage par l'écriture d'un petit programme qui affiche les mots *hello world* 😊)

Laissez les autres options cochées.

Cliquez sur **Next**.

L'assistant permet alors de sélectionner les sous-projets et les bibliothèques nécessaires au projet. Ne rien changer et cliquez sur **Finish**.



Pour finir, cliquez sur **Don't create** quand l'IDE vous demandera si vous souhaitez un module.

4. Premier programme Java ⇒ afficher un message

En Java tout est objet, donc un simple affichage (dans **main**) doit se faire dans une classe 😊

❑ Créer la classe :

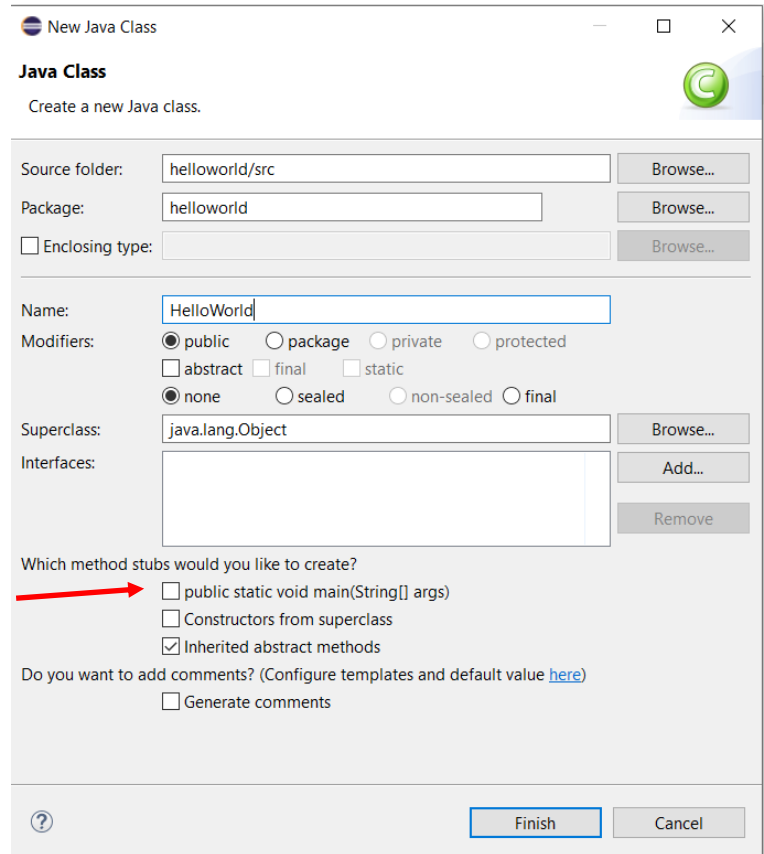
Pour créer une **classe**, choisissez **File → New → Class**

(Remarque : Si l'élément **Class** n'apparaît pas, sélectionnez **Other...** puis **Class** dans la liste qui s'affiche)

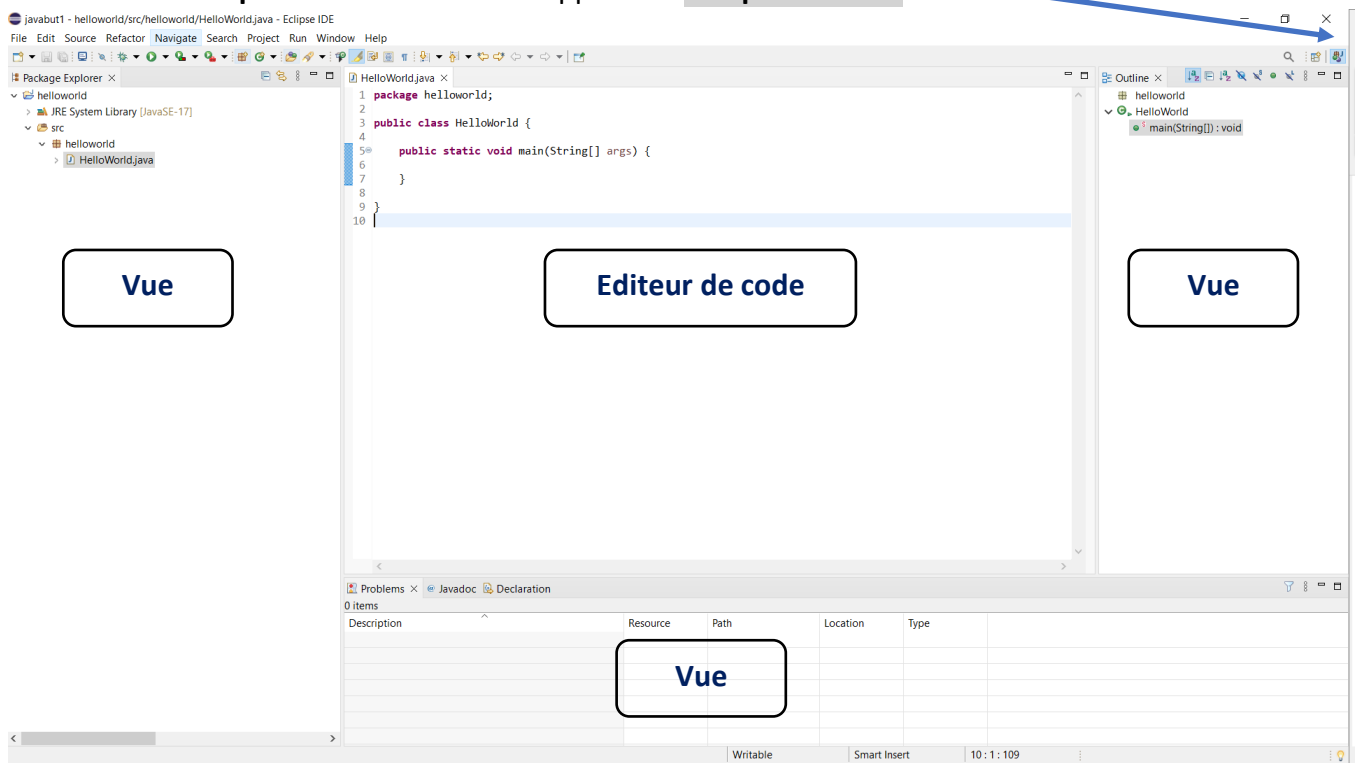
L'assistant *New Java Class* permet de renseigner l'identificateur de la nouvelle classe, son paquetage, sa super-classe et diverses options comme l'ajout d'une méthode **main**, l'implémentation automatique des méthodes abstraites, ...

Pour cette première approche, on remplit juste le nom de la classe en respectant les conventions de nommage de java (première lettre en majuscule et *CamelCase*) : **HelloWorld** (pas besoin de l'extension **.java**).

On peut éventuellement cocher l'option **public static void main (String[] args)** pour éviter d'avoir à l'écrire par la suite si cette classe est supposée disposer d'un **main**, ce qui sera le cas pour ce premier programme 😊
Cliquez sur **Finish**.



Vous obtenez l'espace de travail suivant appelée la **Perspective Java**



Au centre se trouve l'**éditeur de code** qui contient le fichier juste créé : **HelloWorld.java**.

Les autres fenêtres représentent le(s) projet(s) selon un certain point de vue (**Package Explorer**, **Outline**, ...) . Une telle fenêtre est appelée **une «Vue»**.

La partie gauche de l'espace de travail expose la vue **Package Explorer**.

La vue **Package Explorer** permet d'avoir une vision d'ensemble d'un projet, de le développer pour visualiser ces différents éléments et de naviguer entre les différents projets du workspace. Cette vue permet d'accéder rapidement à un fichier de code : il suffit alors de double-cliquer sur ce fichier pour l'afficher dans l'éditeur java.

La partie droite de l'espace de travail expose la vue **Outline** qui propose une vision hiérarchique et structurée du contenu du fichier ouvert dans la fenêtre d'édition : cette vue permettra d'accéder facilement et directement aux différents éléments codés dans le fichier (attributs, méthodes).

La partie inférieure de l'espace de travail expose les vues **Problems**, **Javadoc**, **Declaration**. C'est également dans cette partie qu'apparaîtra la vue **Console** qui exposera le résultat de l'exécution du programme sur la console de sortie.

Afin de gérer au mieux l'espace visuel à l'écran, Eclipse propose un système d'onglets permettant de basculer d'une vue à l'autre. Il est possible de fermer une vue en cliquant sur la croix associée à la vue.

Il est possible d'ajouter de nombreuses autres vues. Le **choix d'une vue** s'effectue à partir de :

Window → Show View

En plus du concept de vue, Eclipse introduit également un concept de **Perspective**.

La copie d'écran de la page précédente (qui doit correspondre à votre plan de travail à l'écran) correspond à la perspective Java. Elle est choisie *automatiquement* par Eclipse dès que le développeur crée un projet Java et permet de visualiser les packages et les classes du projet et leur contenu.

Si le développeur souhaite déboguer un programme, c'est la perspective **Debug** qui devra être choisie par Eclipse.

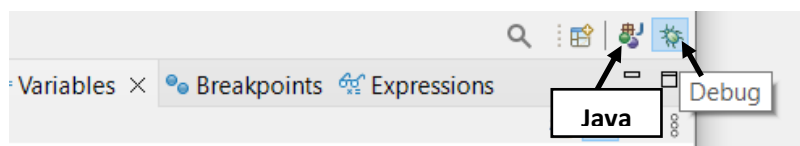
Eclipse propose différentes perspectives sur un même projet. Le choix d'une perspective s'effectue de la manière suivante : **Window → Perspective**

En choisissant **Open Perspectives** on obtient une liste des perspectives possibles.

Choisir par exemple, la perspective **Debug** et cliquez sur **OK**.

Nous reviendrons sur cette perspective un peu plus tard dans le module 😊

Il est possible de passer d'une perspective à l'autre en cliquant sur les **onglets Resource** et **Java** en haut à droite du plan de travail.



A noter : Si, au cours de votre développement, vous fermez des onglets ou en ouvrez d'autres, vous pouvez retrouver à tout moment la perspective Java initiale à partir de :

Window → Perspective -> Reset Perspective

❑ Implémenter la classe (écriture du code java) :

Implémenter dans la méthode `main` de la classe **HelloWorld**, un simple affichage en utilisant l'instruction `System.out.println` qui permet d'afficher un message sur la console :

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
  
        System.out.println("Hello world !");  
  
    }  
}
```

N'oubliez pas de sauvegarder votre fichier :

- Soit par **File** → **Save**
- Soit par le raccourci clavier **CTRL+S** qui va vite devenir votre meilleur ami 😊

A noter : Si le fichier est modifié et n'est pas sauvegardé, une ***** apparaît devant le nom du fichier. L'* disparaît dès lors que le fichier est sauvegardé.

❑ Compiler la classe (en byte code ⇒ HelloWorld.class) :

Un fichier Java est **compilé automatiquement** au moment où vous l'enregistrez si l'option **Build Automatically** du menu **Project** est bien cochée. Vérifiez que ce soit bien le cas sous votre Eclipse et ne touchez à rien 😊

N'oubliez pas d'enregistrer fréquemment vos fichiers pour compiler régulièrement !

Remarque : Vous pouvez choisir à certains moments de compiler *manuellement*. Pour cela, il faudra décocher **Build Automatically** et utiliser les éléments **Build** du menu **Project**.

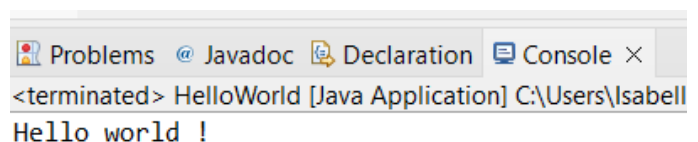
❑ Exécuter l'application depuis l'IDE :

A noter : Avant d'exécuter, assurez-vous bien que tous les fichiers de votre projet ont bien été sauvegardés, c-a-d qu'aucune ***** ne doit apparaître devant le nom d'un fichier !

Pour **exécuter l'application** depuis l'IDE, sélectionner à partir de la barre des menus :

Run → Run as → Java Application

Le code s'exécute alors dans **la vue console** qui apparaît dans la partie inférieure de l'espace de travail.



Vous pouvez aussi vous placer dans l'éditeur sur le fichier **.java** qui contient un **main** et cliquer sur le bouton droit de la souris et sélectionner directement : **Run as → Java Application**

A noter : une application pourra être exécutée (lancée par **Run**) uniquement si une méthode **main** existe dans une des classes du projet (si plusieurs **main** existent, il faudra bien sûr choisir celui à lancer)

La méthode **main** étant le point d'entrée pour l'exécution d'une application 😊

Si on tente de faire exécuter un programme qui contient du code non-sauvegardé java demandera de sauvegarder avant d'exécuter le code.

❑ Se familiariser avec l'arborescence du projet :

Savez-vous exactement où se trouve le code de votre projet sur votre machine ? Vous rappelez-vous du chemin pour y parvenir ?

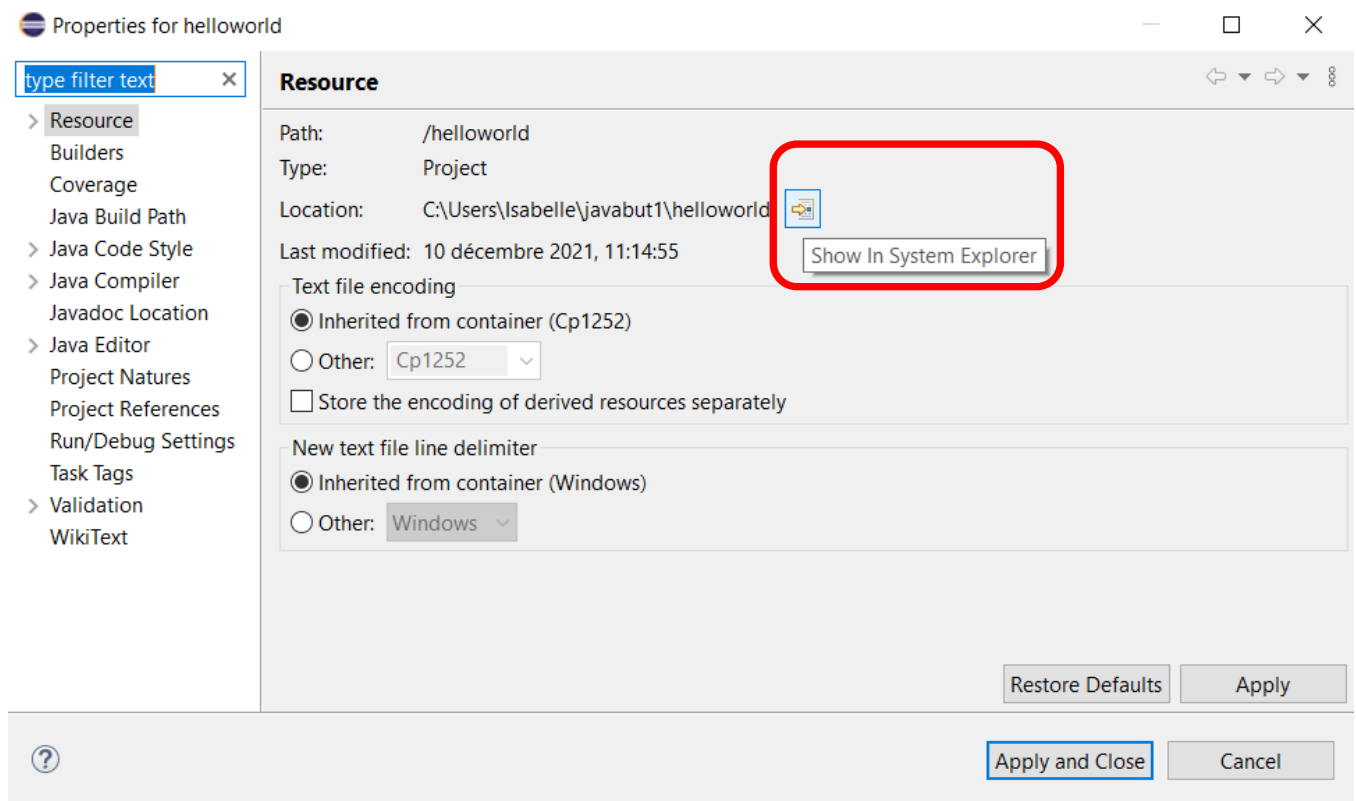
L'IDE peut rapidement vous y emmener, pour cela il suffit de connaître l'astuce suivante :

Dans la vue Package Explorer, placez-vous sur la racine du projet **helloworld**

Faites un clic droit et cliquez sur **Properties** :

Assurez-vous, comme dans le copie d'écran ci-dessous, d'avoir bien sélectionné

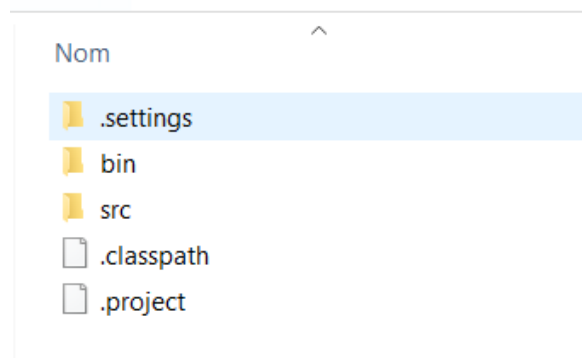
Resource, puis cliquez sur le bouton **Show In System Explorer** à la fin de la ligne **Location** qui indique la localisation (c.-à-d. le chemin) de votre projet sur votre disque.



Cliquer sur le bouton **Show In System Explorer** permet d'ouvrir directement l'explorateur de fichiers au *bon* emplacement du projet, ce qui s'avère très utile dans la pratique 😊.

Vous arrivez donc dans le **workspace javabut1** où vous voyez :

- Un répertoire **.metadata** qui est propre à l'IDE Eclipse
- Le répertoire **helloworld** de votre projet que vous allez ouvrir en double cliquant dessus afin de visualiser l'architecture d'un simple projet Java sous Eclipse au travers de l'arborescence suivante :
 - Le répertoire **.settings** et le fichier **.project** sont propre à la création du projet java sous Eclipse.
 - et fichier **.classpath** spécifie les fichiers source Java et les fichiers de ressources d'un projet pris en compte par le générateur Java et spécifie comment rechercher des types en dehors du projet. Le



générateur Java compile les fichiers source Java dans le dossier de sortie et y copie également les ressources.

- **Le répertoire src contient les fichiers sources : fichier(s) .java** contenant le **code implémenté**. Dépliez-le afin de retrouver le fichier **HelloWorld.java** qui se trouve, par défaut, dans un paquetage (répertoire) ayant le même nom que le projet **helloworld** : c'est ce qu'indique la première instruction du fichier **HelloWorld.java** :
`package helloworld;`
- **Le répertoire bin contient le code compilé : fichier(s) .class** contenant le **byte code** (**code compilé** par l'IDE) qui sera ensuite exécuté par la JVM.
Dépliez également ce répertoire afin de trouver le fichier **HelloWorld.class**
Remarque : si vous êtes curieux et vous voulez voir à quoi ressemble le byte code, vous pouvez double cliquer sur ce fichier et demander à ce qu'il soit ouvert avec Eclipse...
N'oubliez pas ensuite de fermer le fichier **HelloWorld.class** sous Eclipse avant de continuer 😊

Retournez dans l'IDE Eclipse pour continuer !

5. Amélioration du projet et premiers pas dans la javadoc

⇒ pour réussir à ajouter un peu d'interactivité
en permettant la lecture de données entrées (saisies) au clavier

Pour créer un programme interactif acceptant les entrées d'un utilisateur, vous pouvez utiliser `System.in` qui fait référence au **périphérique d'entrée standard** (généralement le **clavier**).

L'objet `System.in` n'est pas aussi flexible que `System.out` : il est conçu pour ne lire que des octets. C'est pourquoi Java propose une classe nommée `Scanner` qui rend `System.in` plus flexible. Pour associer un objet de type **Scanner** à un périphérique d'entrée standard (**System.in**), il faut utiliser l'instruction suivante :

```
Scanner scanner = new Scanner(System.in);
```

❑ Où trouver la documentation de l'API standard Java (la fameuse « **javadoc** ») ?

Pour récupérer dans votre programme, une entrée saisie à partir du clavier, il vous faudra donc appeler une méthode de la classe `Scanner`.

Pour savoir quelles méthodes sont proposées par la classe `Scanner`, rendez-vous dans la **javadoc** :

- La javadoc de la version 17 de java se trouve à l'adresse suivante :

<https://docs.oracle.com/en/java/javase/17/>

- La javadoc de la version 8 se trouvait à l'adresse suivante :

<https://docs.oracle.com/javase/8/docs/api/index.html>

Lorsque vous développez en Java, vous vous référez très souvent à la javadoc (pour ne pas ré-inventer la roue et utiliser des bibliothèques déjà existantes !!!)

Dorénavant, au début de chaque TP, vous penserez à ouvrir la **javadoc correspondante à la version de Java sur laquelle vous travaillez.**

Pour cela, rendez-vous <https://docs.oracle.com/en/java/javase/> choisissez la version de Java, puis API Documentation pour retrouver la javadoc

❑ Se familiariser avec la *javadoc* en consultant classe **Scanner** :

La *javadoc* de la classe **Scanner** peut, par exemple pour la version 17, être consultée à l'adresse suivante : <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Scanner.html>
Astuce : en tapant dans votre moteur de recherche préféré « **javadoc scanner** », vous allez rapidement trouver un lien qui vous mènera vers la page *javadoc* de scanner d'une version Java 😊

Rendez-vous sur la page de la *javadoc* de la classe **Scanner** (version 17 de Java ou autre).
Toutes les pages *javadoc* suivent le même plan (l'IDE peut nous aider à générer de la *javadoc* de nos propres classes si on le souhaite, on verra cela plus tard).
Pour vous familiariser avec cette documentation, déroulez la *javadoc* de **Scanner** en vous référant aux explications suivantes :

La documentation commence par indiquer le paquetage (**package**) dans lequel se trouve cette classe. Concentrez-vous uniquement sur la notion de **package** : on n'abordera pas la notion de module dans ce cours d'introduction à la programmation objet à l'aide de Java 😊

```
Module java.base
Package java.util ←
Class Scanner
  java.lang.Object
    java.util.Scanner

All Implemented Interfaces:
  Closeable, AutoCloseable, Iterator<String>
```

En Java, il est d'usage de regrouper les classes en **paquetages (packages)**, afin de mieux **structurer l'application** et favoriser la **cohérence du code**. Cette structuration a trois intérêts :

- ➔ permettre la modularité, séparation plus claire des différentes parties de l'application ;
- ➔ éviter les conflits de noms (en fournissant un espace de nommage) ;
- ➔ limiter la visibilité des classes.

La hiérarchie des **paquetages** est directement liée aux **répertoires** contenant les fichiers source.

Dès qu'une classe fait partie d'un paquetage, elle change de nom et porte en préfixe le nom du paquetage. Ainsi par exemple, pour utiliser la classe **Scanner**, il faut :

- ➔ soit utiliser son nom complet (on dit aussi *nom qualifié*) : **java.util.Scanner**
- ➔ soit ajouter en début de fichier (après une éventuelle instruction package) une instruction d'importation de la classe : `import java.util.Scanner;`
Si cette ligne est présente au début du fichier, alors seul le nom court (**Scanner**) peut être utilisée par la suite dans tout le fichier. En pratique, on utilise le nom court et l'IDE vous sera d'une grande aide pour ajouter les `import`, comme nous le verrons par la suite 😊

Continuez de parcourir la javadoc de la classe **Scanner**, vous y découvrirez :

- **des exemples de code** qui illustrent comment utiliser un objet de type **Scanner**.
- **une liste de constructeurs** dans la rubrique **Constructor Summary** pour vous permettre d'instancier votre objet de type Scanner en fonction de vos besoins (ci-dessous ne sont présentés qu'une partie des constructeurs que propose la javadoc en ligne...)

Constructor Summary

Constructors	
Constructor	Description
<code>Scanner(File source)</code>	Constructs a new <code>Scanner</code> that produces values scanned from the specified file.
<code>Scanner(File source, String charsetName)</code>	Constructs a new <code>Scanner</code> that produces values scanned from the specified file.
<code>Scanner(File source, Charset charset)</code>	Constructs a new <code>Scanner</code> that produces values scanned from the specified file.
<code>Scanner(InputStream source)</code>	Constructs a new <code>Scanner</code> that produces values scanned from the specified input stream.
<code>Scanner(InputStream source, String charsetName)</code>	Constructs a new <code>Scanner</code> that produces values scanned from the specified input stream.
<code>Scanner(InputStream source, Charset charset)</code>	Constructs a new <code>Scanner</code> that produces values scanned from the specified input stream.

- **une liste de méthodes** dans la rubrique **Method Summary** pour vous permettre d'utiliser votre Scanner à bon escient en fonction de vos besoins:

Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method	Description
void	<code>close()</code>	Closes this scanner.
Pattern	<code>delimiter()</code>	Returns the <code>Pattern</code> this <code>Scanner</code> is currently using to match delimiters.
<code>Stream<MatchResult></code>	<code>findAll(String patString)</code>	Returns a stream of match results that match the provided pattern string.
<code>Stream<MatchResult></code>	<code>findAll(Pattern pattern)</code>	Returns a stream of match results from this scanner.
String	<code>findInLine(String pattern)</code>	Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.

... et bien d'autres dont un certain nombre de méthodes qui commencent par **nextXXX** 😊

OVERVIEW MODULE PACKAGE CLASS USE TREE PREVIEW NEW DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH:

String	next()	Finds and returns the next complete token from this scanner.
String	next(String pattern)	Returns the next token if it matches the pattern constructed from the specified string.
String	next(Pattern pattern)	Returns the next token if it matches the specified pattern.
BigDecimal	nextBigDecimal()	Scans the next token of the input as a <code>BigDecimal</code> .
BigInteger	nextBigInteger()	Scans the next token of the input as a <code>BigInteger</code> .
BigInteger	nextBigInteger(int radix)	Scans the next token of the input as a <code>BigInteger</code> .
boolean	nextBoolean()	Scans the next token of the input into a boolean value and returns that value.
byte	nextByte()	Scans the next token of the input as a <code>byte</code> .
byte	nextByte(int radix)	Scans the next token of the input as a <code>byte</code> .
double	nextDouble()	Scans the next token of the input as a <code>double</code> .
float	nextFloat()	Scans the next token of the input as a <code>float</code> .
int	nextInt()	Scans the next token of the input as an <code>int</code> .
int	nextInt(int radix)	Scans the next token of the input as an <code>int</code> .
String	nextLine()	Advances this scanner past the current line and returns the input that was skipped.
long	nextLong()	Scans the next token of the input as a <code>long</code> .
long	nextLong(int radix)	Scans the next token of the input as a <code>long</code> .

Que ce soit dans la rubrique **Constructor Summary** ou **Method Summary**, vous pouvez **double-cliquer** sur un nom de constructeur ou un nom de méthode pour avoir plus de détails sur ces derniers. Cette action vous amènera directement au *bon* endroit dans les rubriques suivantes **Constructor Details** ou **Method Details** c.-à-d. au *bon* emplacement, celui qui vous en dira plus sur l'opération double-cliquée précédemment.

- ➔ Double-cliquez par exemple sur le constructeur que nous allons utiliser pour récupérer les entrées saisies par l'utilisateur depuis la console...

Scanner (InputStream source)	Constructs a new Scanner that produces values scanned from the specified input stream.
---	--

.... vous permettra d'obtenir plus de détails sur ce constructeur

Scanner	
<pre>public Scanner(InputStream source)</pre>	
Constructs a new Scanner that produces values scanned from the specified input stream. Bytes from the stream are converted into characters using the underlying platform's default charset.	
Parameters:	
source - An input stream to be scanned	

- ➔ Double-cliquez ensuite sur la méthode **nextLine()** que nous allons utiliser pour récupérer les entrées saisies par l'utilisateur sous forme de chaîne de caractères (**String**)

String	nextLine()	Advances this scanner past the current line and returns the input that was skipped.
---------------	-------------------	---

.... vous permettra d'obtenir plus de détails sur cette méthode

nextLine	
<pre>public String nextLine()</pre>	
Advances this scanner past the current line and returns the input that was skipped. This method returns the rest of the current line, excluding any line separator at the end. The position is set to the beginning of the next line.	
Since this method continues to search through the input looking for a line separator, it may buffer all of the input searching for the line to skip if no line separators are present.	
Returns:	
the line that was skipped	
Throws:	
<code>NoSuchElementException</code> - if no line was found	
<code>IllegalStateException</code> - if this scanner is closed	

Remarque : La méthode **nextLine()** permet donc de récupérer la ligne entière saisie par l'utilisateur jusqu'à ce qu'il appuie sur la touche **Entrée**.

- ➔ Double-cliquez enfin sur la méthode **nextInt()** que nous allons utiliser pour récupérer les entrées saisies sous forme d'entier (**int**)

int	nextInt()	Scans the next token of the input as an int.
------------	------------------	--

.... vous permettra d'obtenir plus de détails sur cette méthode

nextInt	
<pre>public int nextInt()</pre>	
Scans the next token of the input as an int.	
An invocation of this method of the form <code>nextInt()</code> behaves in exactly the same way as the invocation <code>nextInt(radix)</code> , where <code>radix</code> is the default radix of this scanner.	
Returns:	
the int scanned from the input	
Throws:	
<code>InputMismatchException</code> - if the next token does not match the <i>Integer</i> regular expression, or is out of range	
<code>NoSuchElementException</code> - if input is exhausted	
<code>IllegalStateException</code> - if this scanner is closed	

Remarque : La classe `Scanner` ne contient pas de méthode **nextChar()**. Pour extraire un seul caractère du clavier, vous pouvez utiliser la méthode **nextLine()**, puis la méthode **charAt()**.

❑ Utiliser la classe Scanner à bon escient (grâce à la javadoc 😊)

1. Déclaration et Instanciation d'un objet de type Scanner afin d'interagir avec le clavier :

En tout début de votre méthode `main`, ajoutez l'instruction suivante qui permet de déclarer et d'instancier un objet de type `Scanner`.

```
Scanner keyboard = new Scanner(System.in);
```

Comme cet objet va nous permettre de lire les saisies entrées au clavier (`System.in`), nous avons choisi de le nommer `keyboard` pour bien montrer son intention 😊

Une fois, cette ligne de code tapée, l'IDE met une croix rouge dans la marge et souligne en rouge **Scanner**. Cette croix rouge nous indique que notre programme ne compilera pas et qu'il faut traiter à cette ligne une **erreur de compilation**.

L'IDE va plus loin que de simplement nous signaler une erreur de compilation, il nous fait des suggestions pour corriger cette erreur de compilation (en fonction de ce que vous êtes en train d'implémenter à vous de déduire si elles ont pertinentes ou pas 😊)

C'est en cliquant sur la croix rouge que nous pouvons visualiser les suggestions de l'IDE pour corriger cette erreur de compilation comme le montre la copie d'écran ci-dessous :

- ➔ Dans le cadre blanc, des suggestions sont proposés
- ➔ Le cadre jaune évolue en fonction de la suggestion choisie dans le cadre blanc et donne un



aperçu de ce que deviendrait le code actuel si cette suggestion était choisie.

Comme nous l'avons expliqué précédemment, pour pouvoir utiliser la classe `Scanner` de **java.util**, il faut faire un **import java.util.Scanner**; juste après l'instruction `package` comme l'indique la première suggestion.

Double-cliquez dans le cadre blanc sur la première suggestion **Import 'Scanner' (java.util)** afin que l'IDE insère automatiquement le code de l'**import** dans votre fichier.

Sauvegardez votre fichier : vous constatez que la croix rouge a disparu, mais que cette fois-ci un triangle jaune est apparu. 🟡

2. Lecture de données en provenance de la console en utilisant la *bonne* méthode nextXXX :

Complétez votre classe **HelloWorld** de manière à implémenter le code suivant qui vous permettra de tester les méthodes `nextLine()` et `nextInt()`.

```
import java.util.Scanner;

public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello world !");

        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter your firstName :");
        String firstName = keyboard.nextLine();

        int groupNumber;
        do {
            System.out.print("Enter you group number (1,2 or 3 :)");
            groupNumber = keyboard.nextInt();
        } while (groupNumber < 1 || groupNumber > 3);

        System.out.print("Welcome "+ firstName + " from G" +
            groupNumber + " to the Object-Oriented Development class!");

    }

}
```

➔ Sauvegardez ce programme et exécutez-le !

Même s'il y a un warning, sur la ligne de l'instanciation du Scanner (flux utilisé pour la lecture des données depuis la console `System.in`), ce warning n'empêche pas l'exécution du programme 😊

➔ Pour supprimer ce warning, il faut fermer le Scanner (le Scanner est en fait un flux de lecture de données qui s'ouvre lors de l'instanciation et qu'il est de bon ton de fermer lorsque ce dernier n'est plus utilisé)

Ajoutez donc maintenant comme dernière ligne de votre méthode `main` l'instruction suivante :

```
keyboard.close();
```

Sauvegardez et le warning de la première ligne devrait alors disparaître.

Compilez et exécutez !

Dorénavant, vous adopterez cette bonne pratique : un flux de lecture de données de type Scanner doit être fermé lorsqu'il n'est plus utilisé 😊

➔ Facultatif : En consultant la javadoc et les nombreuses méthodes **nextXXX**, vous pouvez vous amuser à lire (saisir) de nouvelles données depuis la console, puis à les écrire (afficher) sur la console.

A retenir

Bien veillez à ce que le nom de la classe commence avec le mot clé **public** et ait le même nom que le fichier **.java**

Bien respecter les conventions de nommage de java
(pour une classe : première lettre en majuscule et CamelCase)

Avec l'IDE Eclipse :



indique une **erreur de compilation**.

Avec une erreur de compilation, le code ne peut pas être compilé et donc **le projet ne peut pas être exécuté !**



indique une **warning**.

Un code avec des warnings peut être compilé et donc **le projet peut être exécuté !**
En mentionnant un warning, l'IDE vous met en garde que quelque chose d'anormal pourrait arriver à l'exécution.

Ainsi pour éviter tout bug intempestif à l'exécution, **une bonne pratique consiste à écrire et compiler un code sans warning avant de lancer l'exécution.**

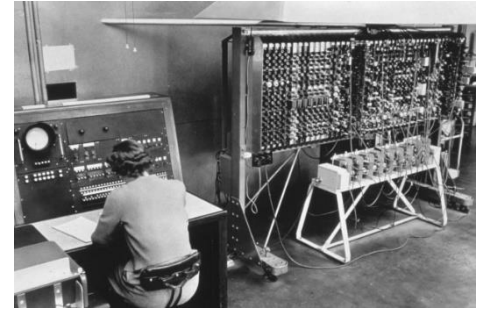
Avoir la *javadoc* à portée de main :

<https://docs.oracle.com/en/java/javase>

6. Programmation Orienté Objets => pas d'objet sans classe !

Les débuts de l'informatique : la classe !

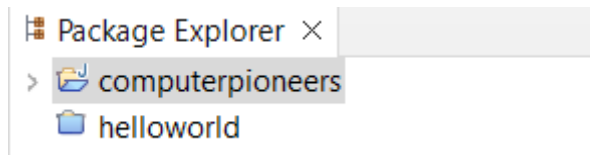
Rappel du cahier des charges (vu en TD) : Imaginez maintenant que le département Informatique soit votre nouveau client et que pour les JPO, il vous demande de créer un jeu (une sorte de memory par exemple) qui permettrait d'associer à quelques personnages célèbres de l'histoire de l'informatique la machine sur laquelle ils ont travaillé.



Commencez par **créer un nouveau projet** que vous appellerez **computerpioneers** (**File** → **New** → **Java Project** et **Don't create** quand l'IDE vous demandera si vous souhaitez un **module**). Par convention nous écrivons les noms de projet en minuscule.

Pour travailler sereinement, uniquement sur ce projet, placez-vous dans la **vue Package Explorer** sur le projet fraîchement créé **computerpioneers** et via à un clic droit sélectionnez **Close Unrelated Project** puis OK.

Cette action permet de fermer tous les projets présents dans le workspace ainsi que tous leurs fichiers associés : seul le projet à partir duquel cette action a été lancée reste ouvert (icône projet ouvert).



Notez que pour ouvrir à nouveau un projet fermé, il suffit juste de double-cliquez dessus.

❑ **Focus sur la classe Device :**

1. Implémentation (écriture du code) :

Dans ce projet, créez une classe **ComputerPioneer** et une classe **Device**.

Implémentez cette classe de manière qu'elle respecte le *design* (conception) du diagramme de classe ci-contre.

Device
-name: String -inventionYear: Integer
+Device(name: String, inventionYear: Integer)

2. Test

Créez une classe **Main** et lors que la création de cette classe cochez la création de la méthode **main**. Implémentez les deux jeux d'essais suivants :

- **Instanciation** d'un objet de **Device** nommé **babbageMachine** dont l'appel de la méthode **toString()** sur cet objet permettra de procéder via un **System.out.println** à l'**affichage** console suivant :

Babbage Analytical Machine was invented in 1837.

A noter : La méthode **toString()** (qui comme son nom l'indique renvoie un **String**) devra bien sûr être ajoutée à la classe **Device** 😊

- **Instanciation** d'un objet de **Device** nommé **turingEngine** dont l'appel de la méthode **toString()** sur cet objet permettra de procéder via un **System.out.println** à l'**affichage** console suivant :

Turing Engine was invented in 1936.

3. Zoom sur la méthode toString()

Enlevez le **toString** de l'affichage de l'objet **turingMachine** de manière à obtenir l'instruction suivante : `System.out.println(turingEngine);`

Exécutez à nouveau votre projet. Quelle différence voyez-vous avec ou sans **toString()** ?

.....

Revenez dans la classe **Device**. Aviez-vous remarqué le petit triangle vert que vous signalez le compilateur à côté de la méthode **toString** ?



```
public String toString() {
```

Si vous cliquez une fois sur ce triangle, vous allez vous retrouver dans la classe **Object**.

Ne touchez à rien dans le code de cette classe !!! Mais rappelez-vous que :

En java, toute classe hérite de la classe Object

Les méthodes equals, hashCode et toString sont déjà définies dans la classe Object

Le **toString** que nous avons écrit dans la classe **Device** a la **même signature** que le **toString** déjà écrit dans la classe **Object**, on dit donc que nous avons **redéfini le toString dans la classe Device** avec la valeur des attributs propres au **Device**.

La méthode **toString()** de la classe **Object** renvoie la référence de l'objet.

Les méthodes **toString()** que nous redéfinissons dans nos classes sont habituellement utilisée pour renvoyer l'état d'un l'objet (rappel : l'état d'un objet correspond à la valeur de ces attributs).

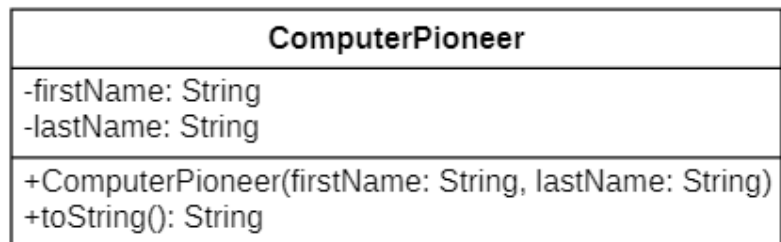
Redéfinir correspond en anglais au mot technique **override** (annule et remplace), c'est pourquoi si vous voulez écrire du code *plus propre*, vous pouvez ajouter l'annotation **@Override** au-dessus de la signature de cette méthode. Nous reviendrons sur ces notions lorsque nous traiterons les concepts d'héritage et de polymorphisme.

```
@Override  
public String toString() {
```


❑ Focus sur la classe ComputerPioneer :

4. Implémentation :

Implémentez maintenant la classe ComputerPioneer qui respecte le *design* ci-contre



5. Test

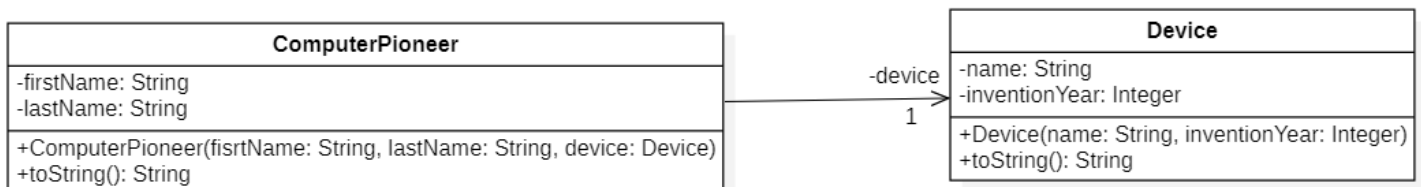
Implémentez dans la méthode main, deux nouveaux jeux d'essais relatifs à la classe ComputerPioneer pour un affichage attendu en mode console :

Ada Lovelace is a pioneer in Computer Science.
Alan Turing is a pioneer in Computer Science.

❑ **Relation entre les classes métier ComputerPioneer et Device sachant que un pionnier travaille sur un et un seul appareil**

6. Implémentation

Complétez votre code de manière à ce qu'il respecte le *design* ci-dessous



7. Test

Réécrire entièrement la méthode main de manière à obtenir les deux jeux d'essais suivants :

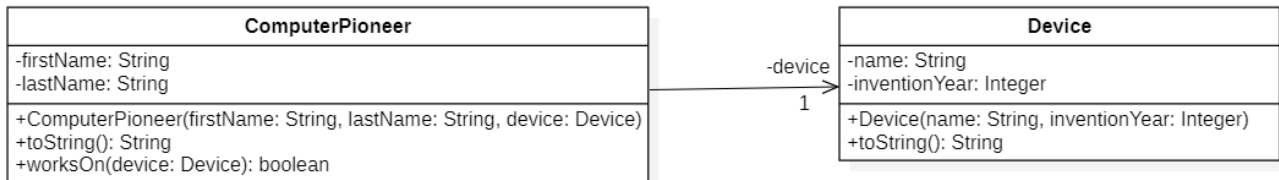
The Babbage Analytical Machine was invented in 1837. Ada Lovelace is a pioneer in Computer Science who worked on it.

The Turing Engine was invented in 1936. Alan Turing is a pioneer in Computer Science who worked on it.

❑ **Ajouter du comportement à la classe ComputerPioneer :**
Être capable de dire si un pionnier travaille sur un device donné

8. Implémentation et Test

Complétez votre code de manière à ce qu'il respecte le *design* ci-dessous
c.-à-d. que devez implémenter un **nouveau service (méthode) worksOn** dans la classe
ComputerPioneer qui prendra un Device en paramètre et dont le comportement consistera à dire si
le *pioneer* travaille sur un certain *device*



Bien sûr l'implémentation de worksOn dépend de ce que vous entendez par :
le *pioneer* travaille sur un certain *device* en question.

Dans un premier temps, complétez votre méthode main avec les deux jeux d'essais suivants :

```
public static void main(String[] args) {

    //... code déjà écrit précédemment

    System.out.println("Test case 3 ");
    System.out.println("-----");
    System.out.println(adaLovelace.worksOn(babbageMachine));
    System.out.println(adaLovelace.worksOn(turingEngine));
    System.out.println(alanTuring.worksOn(babbageMachine));
    System.out.println(alanTuring.worksOn(turingEngine));
    System.out.println("-----");

    System.out.println("Test case 4 ");
    System.out.println("-----");
    Device babbage = new Device ("Babbage Analytical Machine",1837);
    Device turing = new Device ("Turing Engine",1936);
    System.out.println(adaLovelace.worksOn(babbage));
    System.out.println(adaLovelace.worksOn(turing));
    System.out.println(alanTuring.worksOn(babbage));
    System.out.println(alanTuring.worksOn(turing));
    System.out.println("-----");

}
```

8.a Implémentez ensuite la méthode **worksOn** dans la classe **ComputerPioneer** pour faire sorte d'obtenir l'affichage suivant à la console :

```
-----  
Test case 3  
-----  
true  
false  
false  
true  
-----  
Test case 4  
-----  
false  
false  
false  
false  
-----
```

⇒ Comment avez-vous implémenté la méthode **worksOn** ?

.....
.....

Autrement dit la méthode **worksOn** compare la de deux objets de type Device (celui utilisé lors de l'instanciation de l'objet ComputerPioneer et celui passé en paramètre de la méthode **worksOn**)

8.b Implémentez ensuite la méthode **worksOn** dans la classe **ComputerPioneer** pour faire sorte d'obtenir l'affichage suivant à la console :

```
-----  
Test case 3  
-----  
true  
false  
false  
true  
-----  
Test case 4  
-----  
true  
false  
false  
true  
-----
```

⇒ Comment avez-vous implémenté la méthode **worksOn** ?

.....
.....

Autrement dit la méthode **worksOn** compare de deux objets device (celui utilisé lors de l'instanciation de l'objet ComputerPioneer et celui passé en paramètre de la méthode **worksOn**)

A retenir

Une bonne pratique de conception pour le diagramme de classes consiste

à ne **représenter que les opérations (services)**
qui apportent un comportement métier pertinent
(comme `worksOn` par exemple)

c.-à-d. qu'habituellement, pour ne pas alourdir la lecture des diagrammes de classes,
on ne représente pas sur le diagramme de classes :

ni les constructeurs, ni les getteurs/setteurs,

ni méthodes `toString`, `equals` et `hashCode` :

ces méthodes pouvant être **générées automatiquement via l'IDE :**

par exemple sous Eclipse avec le Menu **Source** qui propose
une série de sous-menus **Generate...**

Une bonne pratique d'implémentation Java consiste pour chaque nouvelle classe métier

à **redéfinir pour les 3 méthodes de la classe `Object` :**

`toString`, `hashCode`, `equals`

(pour le moment via une génération automatique de l'IDE,

ou comme on le verra plus tard via `lombok`

ou en utilisant pertinemment la nouvelle notion *record* disponible depuis la version Java 14)

Pour implémenter une égalité en Java,

il est indispensable de bien comprendre la **différence** entre :

<code>==</code>	Comparaison des références de deux objets
<code>equals</code>	Comparaison du contenu de deux objets (état)

7. Reverse-engineering (rétro-conception) d'un projet existant

Pour faire de la **rétro-conception** (*reverse engineering*) sur un projet Eclipse (c-a-d transformer automatiquement le code du projet en diagramme de classe en repassant), il est nécessaire d'installer un plug-in à notre IDE Eclipse.

Le plug-in le plus facile à utiliser pour la rétro-conception est **Object Aid UML Explorer**.

1. Choix et installation d'un plug-in sous Eclipse

Il y a deux manières d'installer un plug-in sous Eclipse :

- soit à du menu **Help** → **Install New Software...**
- soit à du menu **Help** → **Eclipse Market**

Quoi qu'il en soit, retenez bien que si vous ajoutez un plug-in à votre IDE, **vous devez impérativement redémarrer votre IDE après l'installation du plug-in pour que ce dernier puisse être utilisé !!!**

⇒ **Rendez-vous sur le site** <https://www.objectaid.com> .

Si vous voyez le logo ci-contre, suivez les instructions de la rubrique d'installation qui vous permettra d'installer ce plug-in via le menu **Install New Software...**



⇒ **Si malheureusement le site d'Object Aid est vide** suivez les instructions suivantes qui vous permettront d'installer un autre plug-in **UML Lab Modeling** via l'**Eclipse Marketplace**

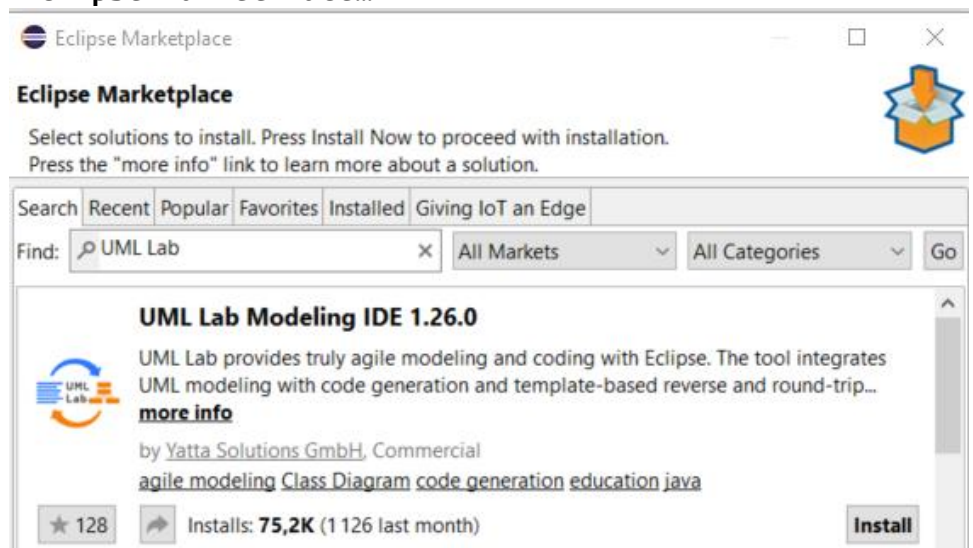
Ce plug-in est plus complet que le précédent, mais pas de panique, nous n'utiliserons pour le moment que le reverse sur le diagramme de classe 😊



2. Installation du plug-in UML Lab Modeling via l'Eclipse Marketplace (à faire si ObjectAid n'a pas pu être installé)

Rendez-vous dans **Help** → **Eclipse MarketPlace...**

Dans la barre de recherche, tapez **UML Lab** et cliquez sur **Go** pour rechercher le plug-in **UML Lab Modeling**.



Cliquez ensuite sur le bouton **Install**

Sur l'écran suivant, vérifiez que tout est coché et cliquez sur **Confirm**

Acceptez les termes de la licence pour pouvoir cliquer sur **Finish**.

.... Le plug-in est en train de s'installer, il faut attendre un peu
(vous pouvez suivre la barre de progression de l'installation en bas à droite de votre IDE)
jusqu'à ce que vous voyez un message qui vous demande de redémarrer l'IDE

Cliquez sur **Restart Now** pour pouvoir que l'IDE puisse redémarrer et prendre en compte ce plug-in.

Fermez l'onglet **Welcome** si ce dernier s'affiche après le redémarrage de l'IDE.

Et vérifiez que vous vous trouvez toujours sur la perspective Java
(en haut à droite de votre IDE)



3. Utilisation d'un plug-in de rétro-conception :

⇒ Si vous avez pu installer **ObjectAid**, rendez-vous sur le tutoriel suivant

https://github.com/iblasquez/tuto_ModelisationUML/blob/master/ObjectAid/README.md

qui vous expliquera comment l'utiliser : après avoir créé un fichier prêt à recevoir le diagramme de classe qui aura l'extension *.ucls*, il suffira juste de faire du drag and drop des classes depuis la vue Package Explorer (drag) dans ce fichier (drop) et le diagramme de classes se générera tout seul 😊



⇒ Si vous avez installé **UML Lab Modeling**, suivez les instructions suivantes 😊

Nous allons suivre les tutoriels officiels.

Rendez-vous sur le site <https://www.uml-lab.com/en/uml-lab/>

Choisissez la rubrique UML Lab, puis Tutorials, pour vous retrouver sur la page :

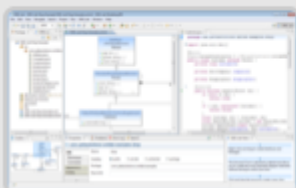
<https://www.uml-lab.com/en/uml-lab/tutorials/>



Seul le premier tutoriel nous intéresse pour le moment, cliquez donc sur :

From Source Code to UML

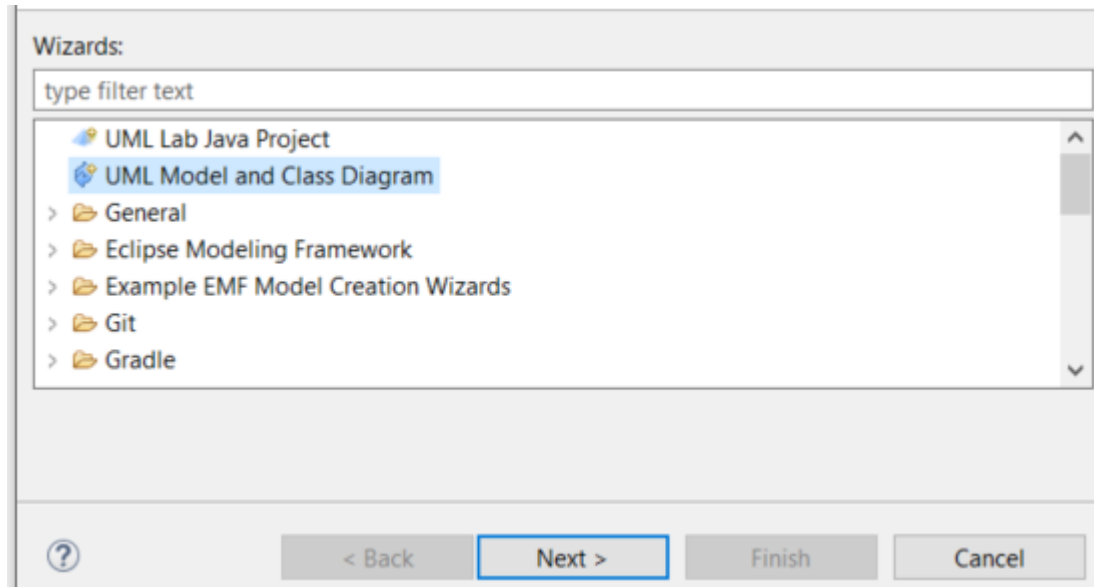
FROM SOURCE CODE TO UML



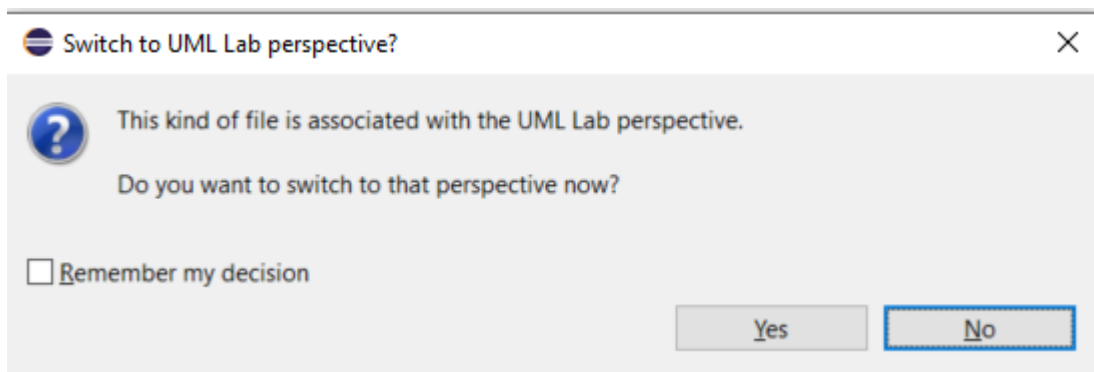
Creating a UML model from your existing source code is really easy with UML Lab. We call this Reverse Engineering. Find out just how easy it is in this tutorial.

Nous allons suivre les étapes de ce tutoriel :

- Placez-vous dans le vue Package Explorer sur le projet à partir duquel vous voulez faire la rétro-conception (**computerpioneers** pour nous)
- A l'aide d'un clic droit, choisissez **New → UML Model and Class Diagram**
Si vous ne voyez pas directement UML Model and Class Diagram, passez par :
New → Other... → UML Model and Class Diagram et cliquez sur **Next**

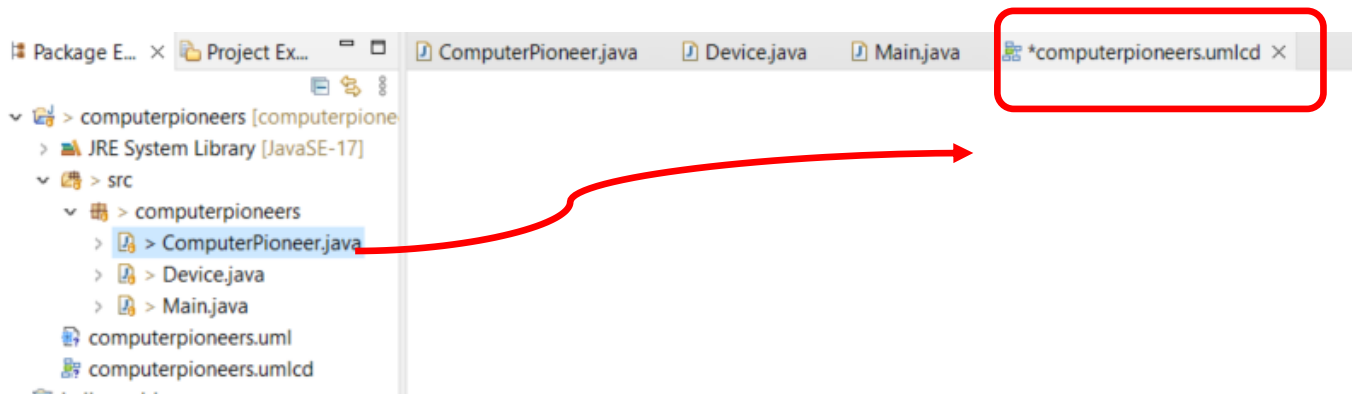


- Sur l'écran suivant vous devez visualiser le projet : **computerpioneers**
Et avoir un champs **File Name** pré-rempli avec **computerpioneers.uml**
(nom du projet + extension .uml) Cliquez sur **Finish**.
- **Restez dans la perspective Java**, ne passez pas dans la perspective UML Lab Perspective, autrement dit, répondez **No** à la question Suivante :

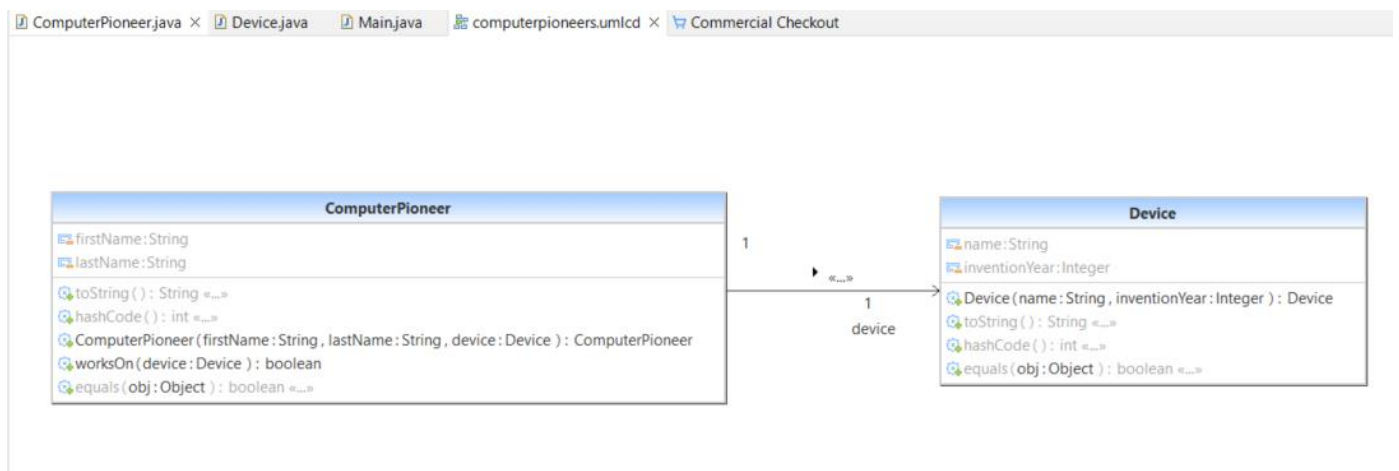


Rester dans la perspective Java, vous évitera de fausses manipulations pour l'instant.
En effet, quand vous ferez des modifications dans votre code, votre diagramme de classes sera modifié en conséquence, mais si vous modifiez votre diagramme de classes, votre code ne sera pas impacté 😊

- Normalement un fichier avec une extension **.umcd** s'est ouvert dans votre éditeur de code. Il ne vous reste plus qu'à **drag & drop** (glisser & déposer) une à une les classes de la vue Package Explorer vers le fichier **computerpioneers.umlcd** pour que le diagramme de classes rétro-conçu du projet apparaissent



.... pour obtenir :



- Lorsque vous **sauvegardez le fichier .umlcd pour la première fois**, choisissez :

Never Generate Code on Save

ce qui vous évitera de mauvaises surprises 😊

- Attention, vous avez noté en bas du fichier .umlcd, le message suivant :

Trial version (1 day left)

Sign in to use your extended trial or subscribe for a license

Ce plug-in est soumis à licence, vous ne pourrez l'utiliser qu'un jour ☹

Tips : Mais heureusement, avec votre statut étudiant, vous pouvez très souvent bénéficier d'une licence dite « academic » ou « education » en vous inscrivant avec l'adresse mail de l'université et/ou un scan de votre carte étudiant , et c'est le cas pour UML Lab

Modeling 😊

4. Activer une licence academic

Rendez-vous à l'adresse suivante <https://www.yatta.de/products/uml-lab/en/academic/> pour prendre connaissance du **programme académique** que propose yatta pour UML Lab Modeling pour les étudiants. Cliquez sur **Register now** et suivez les instructions.

Vous pouvez choisir de vous connecter par votre compte Github ou Facebook ou Google LinkedIn (1)

Votre adresse universitaire vous sera demandée dans un formulaire par la suite 😊

Une fois que vous avez demandé votre licence,

Vous pouvez revenir sur Eclipse et cliquer sur **Sign up** et vous connecter avec le compte choisi en (1).

For students

Single user license for academic purposes. No commercial usage allowed (not for profit). Requires registration and confirmation.

 Single-user license

ou

 Registration with student ID

 Extendable after six months

[Register now](#)

A noter, l'obtention d'une licence académique peut parfois prendre plusieurs jours....

FAQ

Who can participate in the Academic Program?

+

Do I have to register?

+

How long does it take to verify my registration?

×

After you sign up to one of the Academic Program options, we need to verify your application and academic status. This may take up to a day, but if you have been waiting longer than 24 hours, then shoot us an email at academic@yatta.de.

Autre bon plan :



JetBrains propose des IDE dans de nombreux langages très puissants et très appréciés des développeurs professionnels.

L'année prochaine, vous pourrez, par exemple si vous le souhaitez, utiliser **IntelliJ** à la place d'Eclipse, lorsque nous aborderons des concepts avancés d'orienté objets en java.

Comme pour Lab UML Modeling, il est possible d'obtenir une licence gratuite JetBrains grâce à votre statut étudiant (licence à renouveler tous les ans) en suivant le lien suivant :

<https://www.jetbrains.com/fr-fr/community/education/#students>