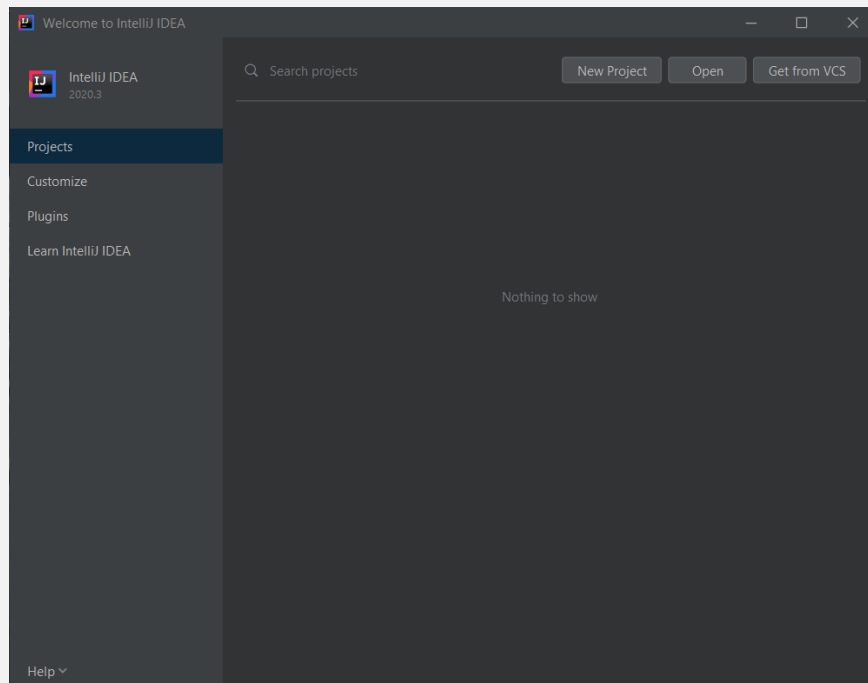


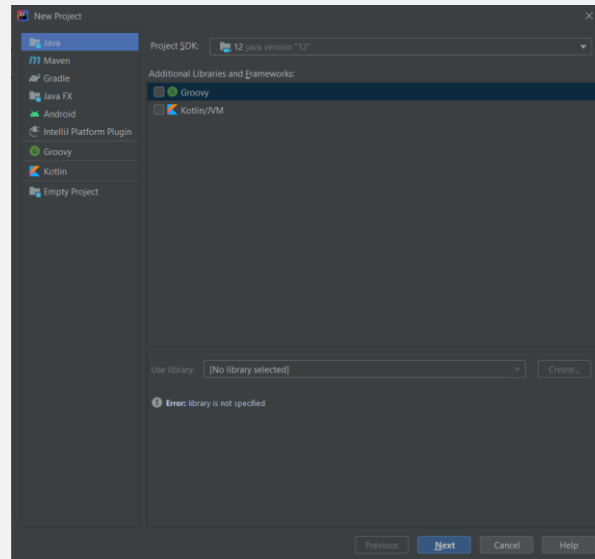
Annexe 1C – « Hello World »

Il est de bon aloi de réaliser un petit programme affichant « Hello World » quand on aborde un nouveau langage. Nous allons tout de même aller un peu plus loin pour donner une idée d'un projet Java réalisé avec IntelliJ IDEA.

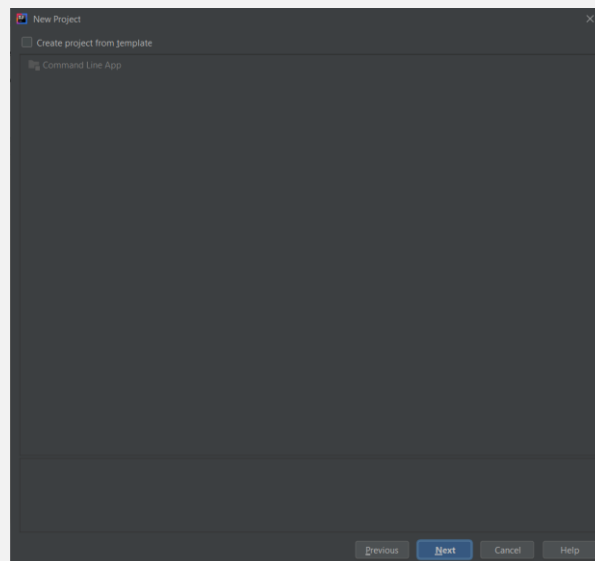
1. Démarrez IntelliJ et cliquez sur New Project



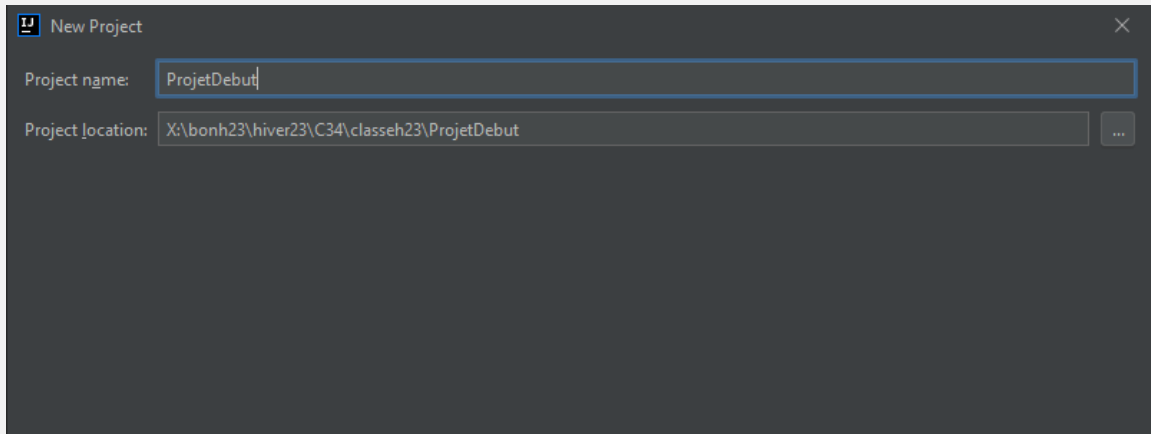
2. On travaille en effet avec le JDK 12 ... faire Next →



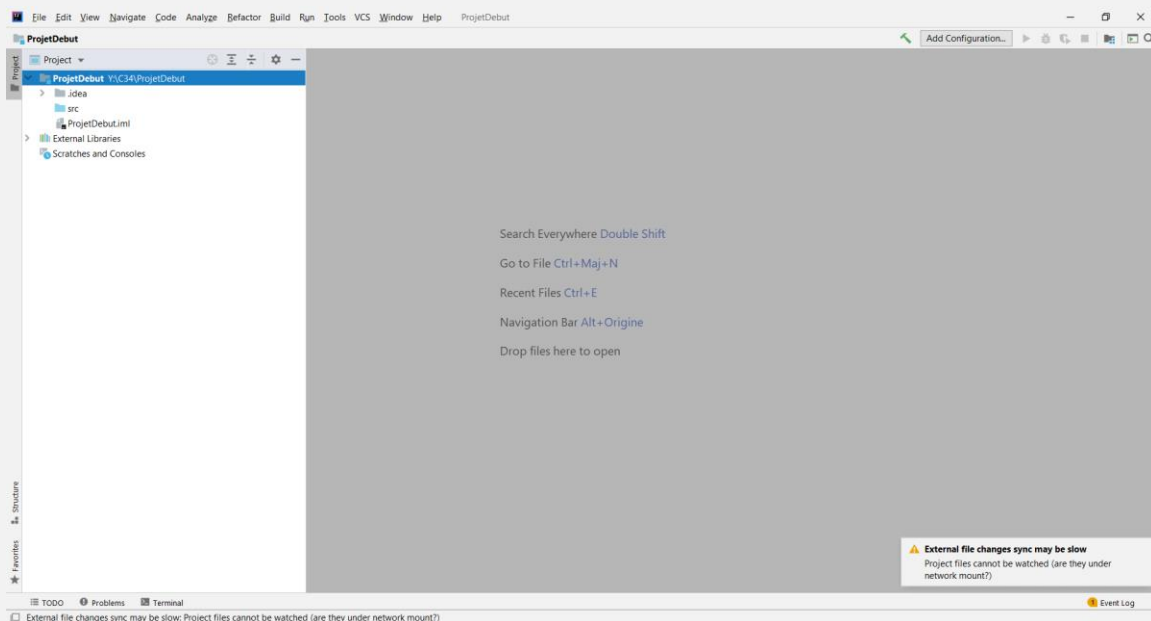
3. Faire Next à nouveau prendre la version 12



4. Donner un nom à votre projet et donner un emplacement dans votre X avec un dossier réservé pour chacun de vos projets :



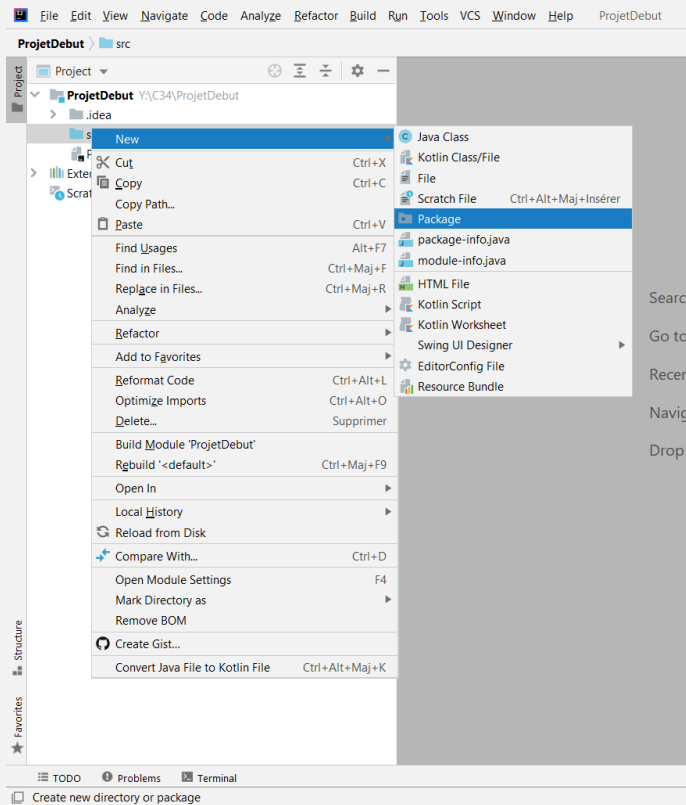
Faire Finish →



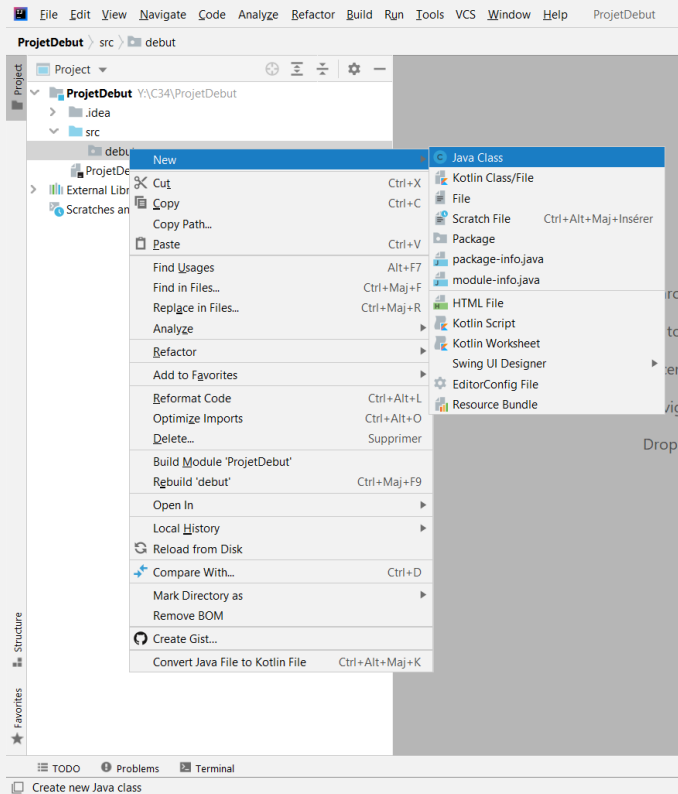
Le dossier src est particulièrement important, c'est lui qui va contenir nos fichiers source (.java) représentant les modèles pour de futurs objets à construire.

Nous allons faire un modèle pour des objets Ordinateur dans le contexte d'un cégep qui doit répertorier tous les ordinateurs faisant partie de son réseau. Les éléments qu'on va faire seront revus par la suite, c'est juste pour coder un peu !

On va commencer par créer un package. Un package représente un ensemble de classes qui ont un lien entre elles :



Sur le dossier src, utilisez le menu contextuel (bouton de droite) pour faire New → Package et donner comme nom de package debut



Sur le dossier du package, créez une nouvelle classe Ordinateur en faisant New → Java Class

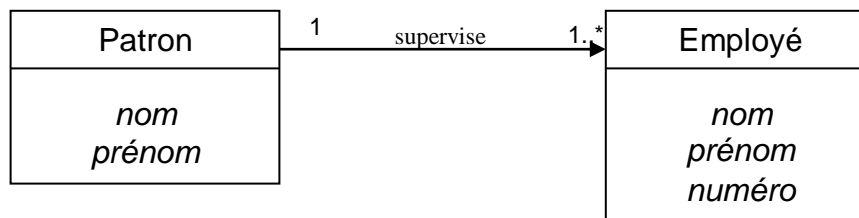
- Discutons des variables d'instance pouvant représenter l'état d'un Ordinateur
- Se créer une seconde classe, TestOrdinateur, qui permettra d'exécuter le programme à l'aide d'une méthode d'exécution main. On crée un objet Ordinateur et on affiche son code.
- On modifie son local et on affiche le numéro de local modifié
- Bouton de droite → Run pour exécuter

ANNEXE 2 - Modèles conceptuels

1. À l'aide des concepts suivants : Moteur, Cylindre, LitreEssence, Pompe, PompeDiesel, PompeSuperSansPlomb, Voiture **et** ReservoirEssence, établissez un modèle conceptuel simple représentant la consommation d'essence d'une voiture.

2. À partir des concepts suivants : Produit, Magasin, CaisseEnregistreuse, LigneDeTransaction, Transaction **et** Paiement concevez un modèle conceptuel du système de la caisse enregistreuse permettant de produire une facture d'épicerie.

Le modèle conceptuel doit contenir les concepts ci-haut mentionnés, des associations entre les concepts, des attributs (données, variables d'instance) et les multiplicités rattachées à tout cela:



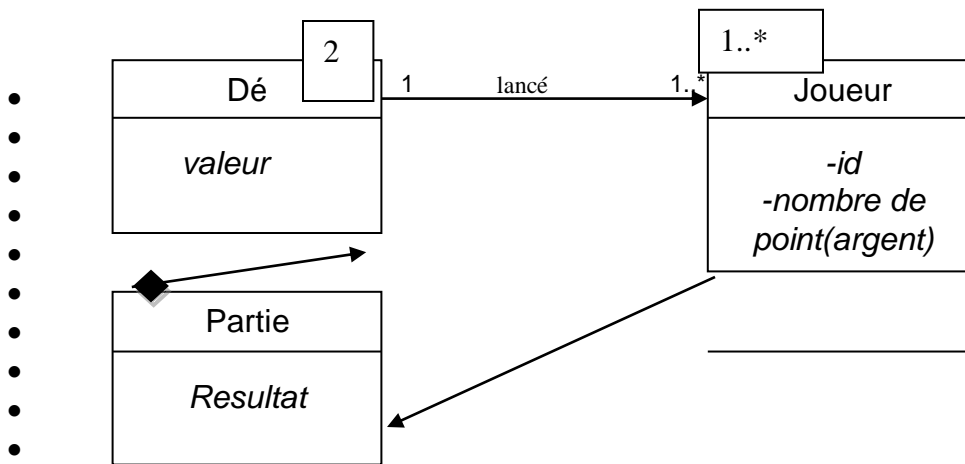
Les attributs que vous avez à inclure dans les concepts doivent permettre l'impression d'un reçu de caisse comme celui-ci:

Magasin Allial			
6212, Vaudry, Montréal, (514) 723-4545			
534334	Biscuits Tradition	4	2.99
366473	Chips Regulier	2	1,49
Total			14,94
Recu			20,00
A Rendre			5,06
Nbre d articles		6	
14:46	Code Magasin 464773-8894		
AU REVOIR			

Il s'agit d'un modèle conceptuel donc :

- pas de type pour les données (*int*, *double*, etc.).
- pas de méthodes pour faire les calculs, seulement les attributs nécessaires afin de pouvoir les réaliser.

Exemple avec un dé : Si on lance deux dé on gagne si la somme des d dés est 7 ou 11



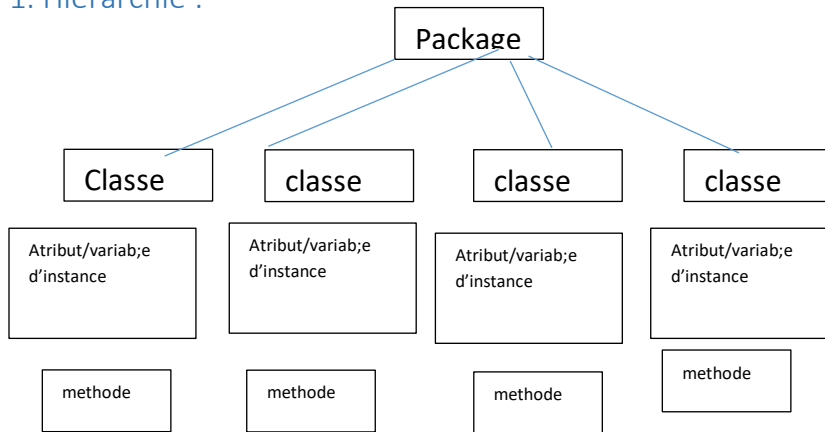
◆ = composition

→ Association

→ Fleche blanche specialisation

Annexe 3 – Éléments de programmation Java

1. Hiérarchie :



Package : regroupement de classe ayant un but commun

Lorsqu'on a besoin d'une classe faisant partie d'un package :

- `import java.util.*` → toutes les classes du package
- `Import java.util.Calendar` → une classe en particulier

*** `java.lang` : Package par défaut comprenant les classes de base, pas besoin de l'important

Classe :

- Sert de modèle aux objets
- Exécutable si elle contient une méthode `main`
- Une classe est en fait un type défini par le programmeur, par opposition à des type précédents prédéfini : `int`, `double`, `char`, `long`, `short`, `real`, `float`

*** Le nom d'une classe commence toujours par une Cliquez ou appuyez ici pour entrer du texte.

Méthode :

- Créée pour accomplir une tâche spécifique (afficher des résultats, demander d'entrer son nom, etc.)
- Permet la réutilisation du code
- Choix du nom très important (facilite sa réutilisation et la compréhension du code)

***** Le nom d'une classe commence toujours par une minuscule**

2. Exemple : Compagnie de taxis

3. Signature / en-tête d'une méthode :

ex. public static void ecrire (String texte)

A D B C

A – modificateurs d'accès → voir Annexe 3B

Public : accès a la méthode partout (en autant que vous avez fait le bon import)

Protected : accès à la méthode :

- Dans les packaged d'origine
- Dans les sous-classe d'un package différent(en autant que bon import)

« Rien » : accès a la méthode dans le package d'origine seulement

Private : accès à la méthode dans la classe d'origine seulement

ANNEXE 3B - Exercice sur les modificateurs d'accès

Voici un ensemble de packages (paquets), de classes et de méthodes pouvant servir à la classification d'employés. Déterminer si les appels aux méthodes peuvent être faits étant donné leur modificateur d'accès.

```
package module1;

public class Employee
{
    private String nom;
    int age;

    public String getNom ()
    {
        ...
    }

    protected void setNom ( ... )
    {
        ...
    }

    void affiche ()
    {
        ...
    }
}
```

////////////////////////////////////

```
package module1;

class Cadre extends Employee
{
...
void affiche ()
    {
        getNom ( );
        setNom ( aString );
    }
}
```

- $$\begin{array}{l} 1. \text{ vrai} \\ 2. \text{ vrai} \end{array}$$

3. _____ Faux _____
4. _____ Vrai _____

////////////////////////////////////

5. _____ vrai _____
6. _____ faux _____

////////////////////////////////////

7. _____ faux _____
8. _____ faux _____

Dans la vraie vie...

les variables d'instance (celles dont la valeur pouvant varier d'un objet à un autre qui sont des instances de la classe) ont toujours comme modificateur d'accès `PRIVATE`

Pourquoi ? Par le principes d'encapsulation des données

IMPORTANT : Quel que soit le modificateur d'accès, les imports nécessaires doivent toujours être faits pour avoir accès à une donnée ou à une méthode.

Annexe 4 – Notes de cours – les constructeurs

A) Méthode constructeur

- Assigne des paramètres à des variables d'instance à l'intérieur de la classe
- Son appel permet la création d'objets à l'extérieur de la classe
- A toujours le même nom que la classe dont il fait partie
- N'a pas de type de retour (même pas void)

EX :

```
Public class EtudiantCegep{
    Private String matricule;
    Private int note;
}
```

```
Public etudiantCegep(matricule1, note1){
    matricule= matricule1;
    note = note1;
}
```

// a l'exterieur de la classe

```
EtudiantCegep e1 =new EtudiantCegep(«1662413 »,79);
```

EX2 :

```
Public etudiantCegep(matricule, note){
    matricule= matricule;
    note = note;
}
```

```
EtudiantCegep e1 =new EtudiantCegep(«1662413 »,79);
```

Problème : Les variables d'instance ne seront pas initialisées, car le paramètre a le même nom

Solutions :

1. On donne des noms différents pour le paramètre et pour la variable
2. Utiliser le mot clé « this » pour distinguer la variable du paramètre

```
Public class EtudiantCegep{  
    Private String matricule;  
    Private int note;  
}
```

```
Public etudiantCegep(matricule1, note1){  
    this.matricule= matricule1;  
    this.note = note1;  
}
```

// à l'extérieur de la classe

```
EtudiantCegep e1 =new EtudiantCegep(«1662413 »,79);
```

B) Surcharge de méthodes :

Habituellement, une classe a plusieurs méthodes « constructeur » qui représentent différentes situations de création d'objets

Solution : surcharge de méthode : plusieurs méthodes peuvent avoir le même nom à condition que soit :

- Le nombre
- Le type
- L'ordre

des paramètres varient d'une méthode à une autre.

EX : On tient à modéliser pour des fins d'inventaire, différents modèles de réfrigérateurs.

La classe Refrigerateur a 4 variables d'instance :

- nomModele
- prix
- capacite (pieds cubes)
- distributeurAGlace (oui ou non)

```
public class Refrigerateur
{
    private String nomModele;
    private double prix;
    private double capacite;
    private boolean aDistributeurAGlace;
```

A-constructeur par défaut sans paramètre

Cliquez ou appuyez ici pour entrer du texte.

B- constructeur pour créer n'importe quel réfrigérateur

Cliquez ou appuyez ici pour entrer du texte.

C-80% des réfrigérateurs reçus sont des Frigidaire , 16 pieds cubes, à 900\$ sans distributeurs à glace

Cliquez ou appuyez ici pour entrer du texte.

D- constructeur n'initialisant pas le prix , il sera déterminé plus tard dans le programme

Cliquez ou appuyez ici pour entrer du texte.

Annexe 5 – Exercice sur les constructeurs



Soit le Gym Gymorrison, ouvert depuis peu. Il offre à ses futurs membres différentes options d'abonnements :

- L'abonnement de base permet un mois (30 jours) d'accès au gymnase. Il coûte 41,25\$.

- Un abonnement " à la carte " est également disponible. Il permet au membre d'acheter un

nombre variable de jours d'accès au gym. Le prix de l'abonnement est de 8,25\$ par jour d'accès.

- Finalement, des abonnements VIP sont possibles (sessions avec entraîneur privé, sessions de Crossfit, etc.) Le nombre de jours d'accès et le prix de l'abonnement varient selon la situation et la période de l'année.

Modélisons donc des objets `AbonnementGym` représentant un abonnement à ce centre d'entraînement. Un objet `AbonnementGym` est caractérisé par 3 attributs représentant son état : le prix de l'abonnement, la durée de celui-ci (en jrs) et le nom du membre auquel est associé l'abonnement en question.

À FAIRE :

1. Créer un projet (`ProjetAnnexe5`) avec un package (`projetannexe5`) et une classe (`AbonnementGym`). La classe `AbonnementGym` contiendra donc les données/attributs nécessaires ainsi que **3 constructeurs** permettant d'initialiser les objets `AbonnementGym` en utilisant le constructeur relatif aux trois cas les plus communs décrits ci-haut.

2. Par la suite, vous allez créer une autre classe `TestConstructeur` dans le même projet et le même package permettant de voir si nos méthodes constructeur fonctionnent bien.

Cette classe ne sert pas à modéliser des objets : elle ne contient donc qu'une méthode `main` que vous pouvez générer à partir de la création de la nouvelle classe en écrivant `main` et en sélectionnant la méthode suggérée

3. À l'intérieur de la méthode `main`, créez 5 objets `AbonnementGym` utilisant un ou l'autre des constructeurs.

4. À l'aide de la méthode `System.out.println`, faites afficher le prix de chaque abonnement. Est-ce possible ? Cliquez ou appuyez ici pour entrer du texte.

Pourquoi?non, parce que la variable `prix` est `private` dans la classe `gym`

Solution : On va coder des méthode d'accès (`get`) qui permettront d'accéder à la variable à l'extérieur de la classe au besoin

Autre méthode : méthode de mutation (`set`)

Pour modifier la variable d'instance à l'extérieur de sa classe-moèle

Annexe 6-Les constructeurs / modélisation

Dans cet exercice, je vous demande de modéliser une classe `CompteBancaire` dans un nouveau projet. Voici les exigences de base :

- Les données d'un objet `CompteBancaire` seront : le nom du propriétaire, le solde au compte et le numéro de celui-ci.
- Le numéro du compte est toujours composé de la première lettre du nom du propriétaire, suivi d'un tiret, suivi du nombre représentant le nombre de comptes créés à ce jour pour l'ensemble de la banque. Par exemple, si le premier compte à avoir été créé était un compte appartenant à Adrien Lachance, son numéro de compte serait : A-1 (pensez à une variable statique)
- Créer des méthodes représentant l'action de déposer ou de retirer une somme dans un `CompteBancaire` donné

Dans une autre classe `Test` comprenant une méthode `main` , simuler dans la méthode `main` les opérations suivantes :

- Un compte est créé pour Flavien Larrivée avec un solde initial de 100 \$
 - Un compte est créé pour Denise Lachance avec un solde initial de 36.78\$
 - Faites afficher le nom du propriétaire du premier compte créé
 - Denise retire 20 \$ de son compte
 - Faites afficher le numéro du compte de M. Larrivée
 - Un compte est créé pour Martial Maurice avec un solde initial de 40\$
 - Faites afficher le numéro de compte de Martial Maurice
 - Faites afficher le solde du compte de Mme Lachance
-
- Variable static : Une variable est static si une seule copie de sa valeur est nécessaire pour l'ensemble de la classe
 - Autre modificateurs :static final

- Animal de compagnie
 - A1 Race :furet nom : fido
 - A2 race :chat nom fido
 - A3 race python nom leo

Allocation dynamique de mémoire

- C'est une variable static si on n'a pas besoin de stocker la valeur dans un objet
- C'est de l'allocation static de mémoire

Public classe animal {

Private string race;

Private string nom;

Public static int nbAniamux = 0;

Public animal (String race, string nom){

This.race = race;

This.nom = nom;

nbAnimeaux ++;

}

Methode static : methode dont le resultat est independant de l'etat(donc des variables d'instance) de l'objet

Ex :

Calcuculerpretbourse()

- N'est pas ne methode static, car elle depent du revenue des parents, etc.

Emplacementdelelevedurantlexamen()

- Est static, car le resultat est independant des variables d'instance

Classe final

Public final classe Mcdo

ANNEXE 7 - les expressions régulières

L'utilisation d'expressions régulières est une manière de comparer des chaînes de caractères à l'aide de correspondances entre ces chaînes et des chaînes-modèle.

L'expression régulière est en fait une String / modèle suivant les conventions suivantes :

Modèles (pour un caractère ou un nombre)	
[abc]	a, b, ou c
[^abc]	Tous les caractères sauf a, b, ou c (négation)
[a-zA-Z]	a à z, ou A à Z, (ensemble)
[a-d[m-p]]	a à d, ou m à p: [a-dm-p] (union)
[a-z&&[def]]	d, e, ou f (intersection)
[a-z&&[^bc]]	a à z, sauf b et c: [ad-z] (soustraction)
[a-z&&[^m-p]]	a à z, mais pas m à p: [a-lq-z] (soustraction)

Modèles généraux (pour un caractère)	
.	Tout caractère n'importe quoi
\d	Un chiffre: [0-9]
\D	Tout sauf un chiffre: [^0-9]
\s	Un caractère blanc: [\t\n\x0B\f\r]
\S	Tout sauf un caractère blanc: [^\s]
\w	Un caractère (lettre ou chiffre): [a-zA-Z_0-9]
\W	Tout sauf un caractère (lettre ou chiffre): [^\w]

Multiplicateurs	
X?	X : 0 ou 1 fois seulement
X*	X: 0 ou plusieurs fois
X+	X: 1 ou plusieurs fois
X{n}	X: exactement n fois
X{n,}	X: au moins n fois
X{n,m}	X: au moins n fois et au plus m fois

***** attention en Java, \ est un caractère d'échappement. Pour utiliser les modèles de caractères, on doit donc utiliser \\.**

On constitue donc une String composée des différents symboles ci-dessus et on compare la chaîne à vérifier (entrée par l'utilisateur) avec le modèle à l'aide de la méthode matches

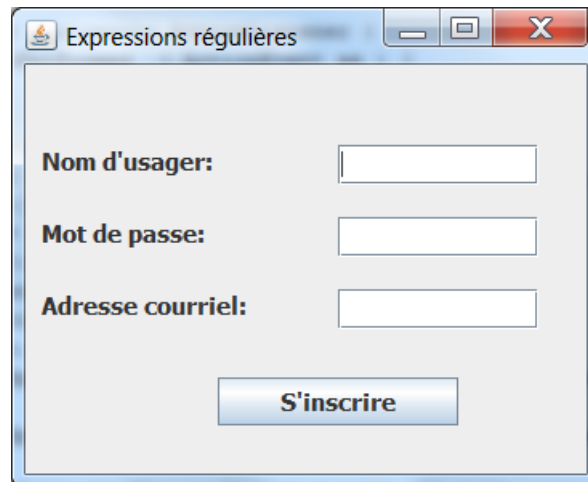
EX.

```
String modele = "..\\d" ;  
System.out.println ("nana".matches(modele)); → false
```

```
String modele = "..\\d" ;  
System.out.println ("na9".matches(modele)); → true
```

Annexe 7C – exercice de programmation avec les expressions régulières

Soit un logiciel installé sur les stations d'un département des finances d'un cégep et qui permet d'accumuler des statistiques sur l'utilisation des ordinateurs. Au début du logiciel, on demande à l'utilisateur d'entrer son nom d'utilisateur, son mot de passe et de fournir une adresse courriel afin de pouvoir être mis au courant de futures offres promotionnelles.

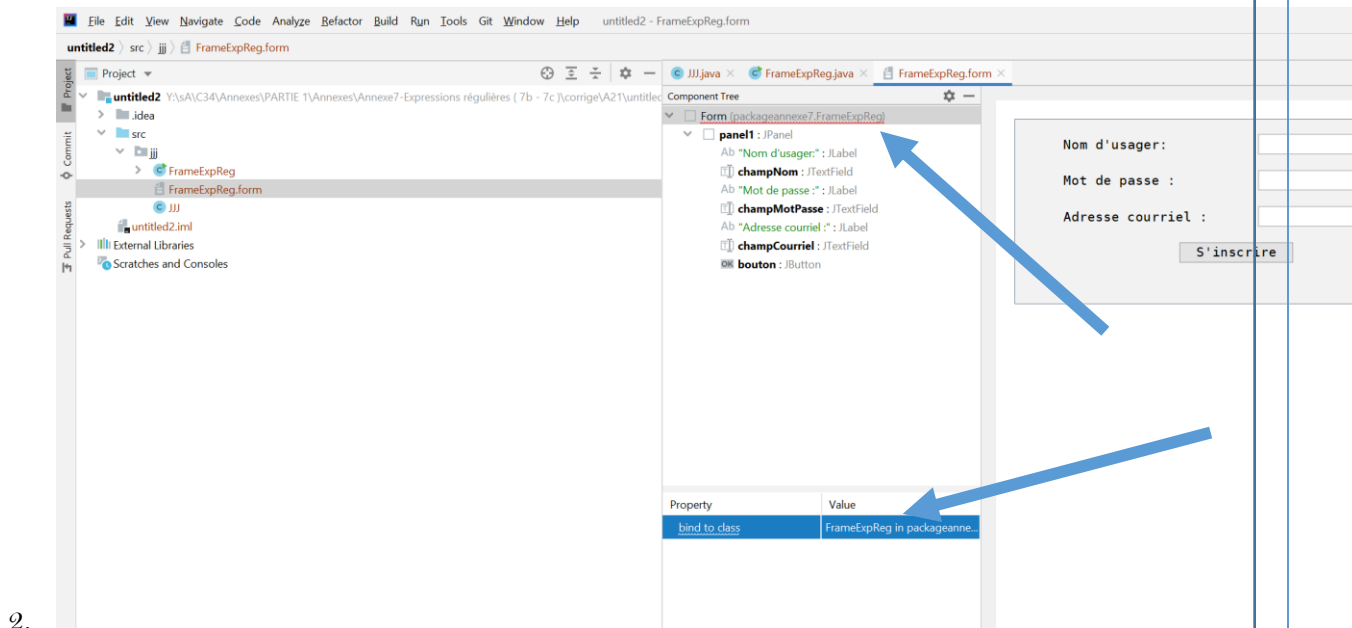


1. Pour ce faire, on doit d'abord créer une classe `Enregistrement`, qui représentera des objets ayant trois variables d'instance correspondant aux trois champs ci-haut.
2. Coder un constructeur permettant d'initialiser les 3 variables à l'aide de trois paramètres ainsi que trois méthodes d'accès.
3. Coder maintenant une classe `Contrôle` qui fera des vérifications sur les objets `Enregistrement`. Dans cette classe, codez 3 méthodes prenant tjrs comme paramètre un objet `Enregistrement` afin de pouvoir le vérifier
 - a. Faites une méthode permettant de vérifier le nom d'utilisateur de l'`Enregistrement` en question. Le nom d'utilisateur doit ressembler à la forme : `R2LabonEr` où `R1`, `R2` ou `R3` représente le type d'employé, suivi du nom (5 lettres) et du prénom (2 lettres)
 - b. Faites une méthode permettant de vérifier le courriel de l'objet `Enregistrement`, soit qu'il aie un arobas (`@`) à l'intérieur
 - c. Faites une méthode permettant de vérifier le mot de passe de l'objet `Enregistrement`; le mot de passe doit avoir 8 caractères ou plus, comprenant au moins un chiffre et au moins une lettre majuscule.

4. Intégrer la fenêtre `FrameTestExpReg` dans votre projet. Modifier le package en conséquence puis tester vos classes à l'aide de cette interface graphique.

Pour intégrer la forme / interface graphique à votre projet (IntelliJ)

1. Glissez et déposez les deux fichiers (`.java` et `.form`) dans le package de votre projet dans IntelliJ. Faire Refactor.



Dans le fichier `.form`, sélectionner `Form` et **modifier la valeur par votre nom de package à la place du mien (`packageAnnexe7`) en cliquant dessus**

3. Devrait pouvoir faire rouler l'interface avec Run. Peut être nécessaire de faire Menu Build → Rebuild project pour recompiler le tout.

Annexe 7B - les expressions régulières (exercices de base + Pattern, Matcher)

1. Vrai ou Faux ?

A) `String un = "allo";`
`un.matches("\\d{4}");`
faux

B) `String deux = " dede";`
`deux.matches("\\s{1,}[a-e]{2,5}");`
vrai

C) `String trois = "343---";`
`trois.matches("...\\w\\w\\w");`
faux

D) `String quatre = "2222";`
`quatre.matches("22");`
faux

E) `String cinq = "sage";`
`cinq.matches("\\s.{3}");`
false

F) `String six = "éric";`
`six.matches("\\w{4}");`
false a cause du e accent

G) `String six = "éric";`
`System.out.println (six.matches("[\\wé]{4}"));`
vrai

H) Une expression régulière permettant de représenter tout matricule du collège ?
`\\d{7}`

I) Une expression régulière pouvant représenter n'importe quel nombre entier positif?

`[1-9][0-9]{0,}` ou `[1-9][0-9]*`

positif, négatif ou nul ? `[-]? Ou -? Ou -?[1-9]\\d*|0`

- J) Une expression régulière pouvant contenir une lettre majuscule suivie de deux astérisques ? `[A-Z]*{2}`

2. Classes Pattern, Matcher

La méthode `matches` de la classe `String` imite en tous points celle de la classe `Pattern` (package `java.util.regex`). En effet, on peut également créer des expressions régulières en créant un objet `Pattern`. Cela permet d'accéder à de nouvelles méthodes...

Les trois principales classes du package `javax.util.regex` :

- `Pattern` :
 - La classe `Pattern` représente une version compilée d'une expression régulière
 - Elle n'a pas de constructeur; on crée un objet `Pattern` à l'aide de la méthode statique `compile` :

```
Pattern p = Pattern.compile (« \\d{5} »);
```
- `Matcher` :
 - La classe `Matcher` permet d'obtenir un objet qui interprétera le `Pattern` (L'expression régulière) et d'y appliquer des méthodes
 - Comme `Pattern`, on ne peut pas créer un objet `Matcher` avec un constructeur, on doit utiliser la méthode `matcher` de la classe `Pattern` :

```
Matcher m = p.matcher (« 123456666633 »);
```
- méthodes à utiliser sur le `Matcher` : `matches`, `find`, `reset`

```
m.matches()
```

 -> faux car 123456666633 ne correspond pas a 5 chiffres

```
m.find()
```

 -> vrai car tu peux trouver 5 chiffres dans 123456666633

```
m.find()
```

 -> vrai car tu peux trouver 5 chiffres a la suite de l'autre 5 chiffres

```
m.find()
```

 -> false car il ne reste qu'un chiffre

```
m.reset
```

 -> retourne au début

- `PatternSyntaxException`
 - Lancée lorsque la syntaxe de l'expression régulière n'est pas correcte

3. autres méthodes pour travailler avec une chaîne de caractères :

- méthode `split` de la classe `String` :

```
String[] tab= "asdf4pa4osi".split("\\d");  
for ( int i = 0; i < tab.length; i++ )  
    System.out.println(tab[i]);
```

Ça va séparer la chaîne de caractères dans un tableau de `String` selon les chiffres et faire de plus petits `String`

`asdf4pa4osi`, alors `tab[0] = asdf`

`tab[1] = pa`

`tab[2] = osi`

- classe `Scanner` (package `java.util`)
 - permet de « scanner » tout un fichier plutôt que seulement une `String`
 - permet de retourner tout type prédéfini plutôt que seulement des `Strings`

- délimiteur par défaut : un caractère blanc (espace, \r, \n, d'autres) mais on peut utiliser une expression régulière à la place

Annexe 8 - Notes de cours : comparaisons d'objets

1. Égalité entre des valeurs de types prédéfinis :

C'est simple, on emploie comme à l'habitude l'opérateur ==.

```
Ex.: int a = 9;
      int b = 9;
      if ( a == b ) → return true
```

2. Égalité entre 2 objets (autres que String)

- il faut d'abord distinguer si on parle de deux objets ou de deux références à un objet; des références étant des adresses indiquant où se trouvent les variables et les méthodes d'un objet donné.

```
Ex.: Point p1, p2;
      p1 = new Point ( 100, 100 );
      p2 = p1;
```

- dans cet exemple, un seul objet est créé mais il y a deux références à cet objet (p1 et p2) Ainsi, toute modification faite sur cet objet à partir d'une ou l'autre des références aura un impact sur l'objet.

```
Ex.: p1.setX( 200 );
      p1.getX() → 200
      p2.getX() → 200 également puisqu'on parle d'un seul objet
```

- l'opérateur == retournera true uniquement si on compare des références à un même objet

```
Ex.: if (p1==p2) → return true car ce sont deux références à un même objet
```

```
Ex.: Point p3 = new Point ( 15,15 );
      Point p4 = new Point ( 15,15 );
      if ( p3 == p4 ) → return false ( ce ne sont pas deux références à un même
objet. )
```


- Pour comparer deux objets entre eux, il faut vérifier si toutes les variables d'instance sont égales. La méthode `equals` est définie dans certaines classes pour réaliser cette tâche (**classes** `String`, `Color`, `Font`, `Point`, `Calendar`, `Hashtable`, `Vector`, **etc.**)

Ex.: `Point p5 = new Point (20,20);`
`Point p6 = new Point (20,20);`
`if (p5.equals (p6)) → return true .` (On a employé `equals` car on avait affaire à deux objets différents)

- Si le type des objets à comparer ne définit pas la méthode `equals`, on doit la redéfinir (la coder) nous-mêmes. Attention, l'appel de la méthode `equals` sur des objets où elle n'est pas redéfinie fonctionnera mais il s'agira de la méthode `equals` de la classe `Object` (héritage). L'emploi de cette méthode est équivalent à `==` dans ces cas.

Ex.: `Roi r1 = new Roi ("k", "noir");`
`Roi r2 = new Roi ("k", "noir");`

`if (r1.equals(r2)) → return false` car la méthode `equals` n'a pas été redéfinie dans la classe `Roi` ou `Piece`, on est donc en train d'utiliser la méthode `equals` de la classe `Object`.

3. Egalité entre 2 objets (`Strings`)

- Le cas des `Strings` est particulier. Comme on l'a vu, ce type est mi-prédéfini, mi-objet. Le choix d'utiliser `==` ou `equals` pour les comparer réside dans la définition des `Strings`.

Ex.: `String s = "bonjour"; // chaîne littérale`
`String t = new String ("bonjour"); // forme objet`

- Dans le cas des chaînes littérales, on peut utiliser l'opérateur `==` en autant qu'on veuille comparer deux références à des chaînes littérales. On peut utiliser `==` car les chaînes littérales sont automatiquement associées à un même objet à l'intérieur lorsque leurs valeurs sont égales (procédé de l'interning).

Ex.: `String a = "allo";`
`String b = "allo";`
`if (a == b) → return true // 2 chaînes littérales donc associées au même objet`

`String a = "allo";`
`if (a == "allo") → return true // idem`

```
String a = "allo";  
String b = champTexte.getText(); // contenant "allo"  
if ( a==b ) → return false car b n'est pas une chaîne littérale
```

```
String a = "allo";  
String b = new String ("allo");  
if (a==b) → return false car b n'est pas une chaîne littérale
```

- pour comparer deux formes objets ou une forme objet avec une forme littérale, on doit utiliser la méthode equals.

Ex.:

```
String a = "allo";  
String b = new String ("allo");  
if ( a.equals(b) ) → return true
```

- finalement, on peut faire de l'interning sur des formes objets String en appelant la méthode intern()

Ex.:

```
String a = "allo";  
String b = new String ("allo");  
b = b.intern();  
if ( a.equals (b) ) → return true  
if ( a == b ) → return true ( elle est devenue littérale )
```