

R2-01-03 : TP 1

Développement orienté objets & Qualité de développement

ENVIRONNEMENT DE TRAVAIL.....	1
POUR COMMENCER.....	1
EXERCICE 1 : CHAÎNE DE CARACTERES : LA CLASSE STRING	2
EXERCICE 1.1 : PREMIER PAS.....	2
EXERCICE 1.2 : UTILISATION DES METHODES DE LA CLASSE STRING	2
EXERCICE 2 : GESTION DES ETUDIANTS.....	3
EXERCICE 2.1 : CREER DES ETUDIANTS	3
EXERCICE 2.2 : LE PRINCIPE D'ENCAPSULATION.....	5
EXERCICE 2.3 : GENERATION D'UNE DOCUMENTATION	6
EXERCICE 2.4 : AJOUTER DES NOTES.....	7
EXERCICE 2.5 : ASSOCIER DES ETUDIANTS A DES GROUPES	7
EXERCICE 2.6 : DEBOGUEUR	8
EXERCICE 3 : GESTION DE COMPOSANTS GRAPHIQUES 2D	9
EXERCICE 3.1 : POINT	9
EXERCICE 3.1 : CERCLE ET POINT (AGREGATION).....	11
EXERCICE 3.2 : RECTANGLE (COMPOSITION)	13
EXERCICE « FIL ROUGE » : LA BATAILLE DE FAËRUN (ÉTAPE 1)	14

Environnement de travail

Nous utiliserons dans ce module l'environnement de développement intégré (en anglais Integrated Development Environment - IDE) **IntelliJ IDEA**. Il est développé par JetBrains (anciennement « IntelliJ ») et disponible en deux versions, l'une communautaire, open source, sous licence Apache 2 et l'autre propriétaire, protégée par une licence commerciale (en tant qu'étudiant vous pouvez demander une licence pour la version ultimate). Les deux versions supportent les langages de programmation Java, Kotlin, Groovy, Scala et Python. La version ultimate propose d'autres langages comme PHP.

Nous utiliserons également le logiciel de gestion de versions **GIT**. C'est un logiciel libre créé par Linus Torvalds, auteur du noyau Linux, et distribué selon les termes de la licence publique générale GNU version 2. Nous prendrons en main cet outil dans les prochains TP.

Pour commencer

1. Créer un projet Java.
2. Créer un package tp1 dans vos sources (**src**). Ce package contiendra tous les exercices de ce premier TP. Un package est un sorte de dossier permettant de structurer son code.

Exercice 1 : Chaîne de caractères : la classe **String**

Objectifs R2-01 : création d'objets, utilisation de méthodes.

Objectifs R2-03 : rédaction de commentaires, lecture de documents (javadoc).

Exercice 1.1 : premier pas

Dans le package `tp1`, **créer** une classe java `TestString`. Dans cette classe, créer une méthode `Main` (méthode permettant un point d'entrée dans le programme). Astuce sur l'IDE IntelliJ, taper « `main` » et l'IDE devrait vous proposer de générer automatiquement la méthode ci-après.

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

Compléter la procédure principale `main` :

1. Déclarer une variable `entree` de type `Scanner` (vu dans le module R1.01)

JAVADOC Scanner : <https://docs.oracle.com/javase/9/docs/api/java/util/Scanner.html>

2. Déclarer une variable `maChaine` de type `String` (chaîne de caractères) initialisée par lecture sur `entree`
3. Afficher la valeur de `maChaine`

Exercice 1.2 : utilisation des méthodes de la classe **String**

JAVADOC String : <https://docs.oracle.com/javase/9/docs/api/java/lang/String.html>

IMPORTANT : votre code doit être commenté ! Un commentaire doit être mis en premier et mis à jour si nécessaire. Cette demande est à la fois un conseil, une bonne pratique et surtout une obligation ! Votre code sera évalué au cours de ce module.

Compléter la procédure principale `main` en manipulant `maChaine` avec des méthodes de la classe `String` en utilisant la javadoc ci-avant et en prenant exemple de la trace donnée ci-après :

1. Afficher le nombre de caractères dans la chaîne `maChaine`
2. Passer `maChaine` en majuscules et l'afficher
3. Contrôler que deux chaînes sont identiques sans tenir compte de la casse. Pour ce faire, lire une seconde chaîne et afficher un message différent selon que les chaînes sont égales ou non
4. Contrôler que `maChaine` est en minuscules (en affichant un message approprié)
5. Afficher `maChaine` entièrement en majuscules à laquelle on aura enlevé les éventuels blancs au début et à la fin
6. Prouver qu'une chaîne est un palindrome. Utiliser la classe `StringBuilder` pour cela (plutôt qu'une solution « algorithmique », nous privilégions la lecture de la javadoc ci-après dans cet exercice)

JAVADOC StringBuilder : <https://docs.oracle.com/javase/9/docs/api/java/lang/StringBuilder.html>

Exemple de trace :

```
Donner une chaîne de caractères maChaine :  
rotor
```

1. Nombre de caractères de la chaîne de caractères maChaine : 5
 2. La chaîne de caractères maChaine en majuscule : ROTOR
 3. Donner une deuxième chaîne de caractères deuxiemeChaine :
RoToR
- Les deux chaînes sont identiques – sans tenir compte de la casse.
4. La chaîne de caractères maChaine est en minuscule.
 5. La chaîne de caractères maChaine en majuscule sans les 'blancs' en début et fin de chaîne : ROTOR
 6. La chaîne de caractères maChaine est un palindrome.

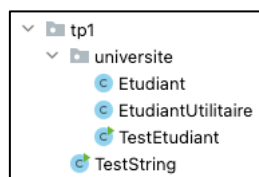
Exercice 2 : Gestion des étudiants

Exercice 2.1 : Créer des étudiants

Objectifs R2-01 : création d'objets, utilisation de méthodes.

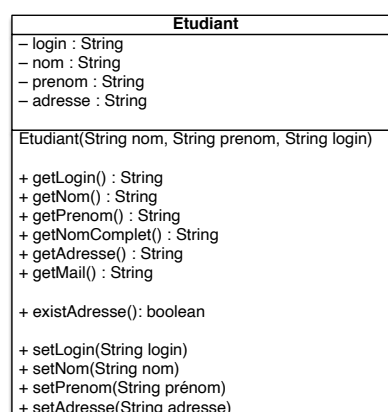
Objectifs R2-03 : création d'une structure dans le source du projet, création d'une classe utilitaire.

Dans le package tp1, **créer** un package universite et dans ce package une classe java **Etudiant**, une classe **TestEtudiant** et une classe **EtudiantUtilitaire**. Les sources de votre projet devraient ressembler à ceci :



Pour le moment, un étudiant a un login, un nom, un prénom et une adresse. **Compléter** la classe **Etudiant** en ajoutant les attributs, constructeurs et méthodes proposés dans le diagramme UML proposé ci-après. Ce diagramme représente la notion d'étudiant.

Dans ce module, vous mettrez en programme les **diagrammes UML** proposés. Dans le module R02.01b, vous apprendrez à les créer.



ATTENTION nous ajoutons des contraintes sur la notion d'étudiant :

- Un étudiant doit toujours avoir un login en minuscule. La méthode **setLogin(String login)** devra modifier le login passé en paramètre en conséquence.
- Un étudiant doit toujours avoir un nom et un prénom avec le premier caractère en majuscule et les autres en minuscule. Exemple : Hervé Blanchon. Pour ce faire, vous

créerez, complétez et utiliserez la méthode `capitalize(String chaîne)` : **String** qui prend en paramètre une chaîne de caractères et qui retourne la même chaîne de caractères avec le premier caractère en majuscule et les autres en minuscule (se référer à la javadoc de **String** vu dans l'exercice précédent). Cette méthode sera à placer dans la classe **EtudiantUtilitaire**.

Pour rappel : signature de la méthode `capitalize` dans la classe **EtudiantUtilitaire** :

```
public static String capitalize(String chaîne) {
```

Exemple d'utilisation de la méthode `capitalize` dans la classe **Etudiant** :

```
public void setNom(String nom) {  
    this.nom = EtudiantUtilitaire.capitalize(nom);  
}
```

Explication de certaines méthodes du diagramme UML :

- **getNomComple()** retourne une chaîne de caractères composée du prénom suivi du nom de l'étudiant
- **getMail()** retourne une chaîne de caractères correspondant à « prenom.nom@etu.univ-grenoble-alpes.fr »
- **existAdresse()** retourne Vrai si l'étudiant a une adresse, Faux sinon. Pour vérifier, si un objet existe vous pouvez vérifier s'il n'est pas **null** – `adresse != null` – et vous pouvez dans le cas d'un **String** vérifier aussi s'il n'est pas vide avec la méthode **isEmpty()**

La classe **TestEtudiant** va permettre de tester notre classe **Etudiant** : création d'objets de type **Etudiant**, modification de ces objets à l'aide de leurs méthodes. **Créer et compléter** la procédure principale `main` pour tester la classe **Etudiant** en prenant exemple de la trace donnée ci-après :

- Créer 2 étudiants à l'aide des constructeurs
- Ajouter une adresse à un étudiant à l'aide de la méthode **setAdresse()**
- Afficher ces étudiants à l'aide d'une méthode **affiche(Etudiant)** que vous créerez dans la classe **EtudiantUtilitaire**
- Vérifier que les contraintes sont respectées, modifier les classes **Etudiant** et **TestEtudiant** en conséquence. Pensez à mettre des valeurs pour les logins, noms et prénoms non conformes à la demande pour voir si votre classe les modifie en conséquence.

Exemple de Test :

```
public static void main(String[] args) {  
  
    // Création des étudiants  
    // IMPORTANT mettre des valeurs qui ne respectent pas les contraintes  
    // pour vérifier leur prise en compte  
    Etudiant etudiant1 = new Etudiant("DUPONTP", "pierre", "DUPONT");  
    Etudiant etudiant2 = new Etudiant("martinf", "francis", "martin");  
  
    // Ajouter une adresse  
    etudiant2.setAdresse("2 Place Doyen Gosse");  
}
```

AIDE : une erreur peut se produire lors de l'exécution du programme. En java, le terminal produira une **stack trace** (plusieurs lignes en rouge sur le terminal de sortie, exemple ci-après). La première ligne donne la cause de l'erreur (exemple TRÈS courant : erreur de type **NullPointerException** – vous avez oublié de créer l'objet que vous souhaitez utiliser). La

suite montre la séquence des appels de méthodes effectués jusqu'au moment où l'erreur d'exécution s'est produite. Le problème vient souvent de la première ligne en partant du haut provenant de votre code. Cliquez dessus et l'IDE vous amènera directement sur le problème.

Exemple d'erreur ci-dessous provenant d'une adresse non initialisée que l'on cherche à afficher.

Login : dupontp

Nom complet : pierre dupont

```
Exception in thread "main" java.lang.NullPointerException
    at tp1.universite.Etudiant.affiche(Etudiant.java:63)
    at tp1.universite.TestEtudiant.main(TestEtudiant.java:11)
```

CONSEIL : Toujours demander à un enseignant la signification d'une erreur si vous ne la comprenez pas !

Exemple de trace :

```
-----
Login : dupontp
Nom complet : Pierre Dupont
Mail : Pierre.Dupont@etu.univ-grenoble-alpes.fr
Adresse : aucune

-----
Login : martinfr
Nom complet : Francis Martin
Mail : Francis.Martin@etu.univ-grenoble-alpes.fr
Adresse : 2 Place Doyen Gosse
```

Exercice 2.2 : Le principe d'encapsulation

Objectifs R2-01 : mettre en place le principe d'encapsulation.

Question : Est-ce que votre classe **Etudiant** respecte les contraintes demandées ? Avez-vous respecté le principe d'encapsulation ?

IMPORTANT principe d'encapsulation : un utilisateur extérieur ne doit pas modifier directement l'état d'un objet (les attributs) et risquer de mettre en péril l'état et le comportement de l'objet.

Pour vérifier cela, dans la classe **TestEtudiant** avez-vous accès directement à l'attribut **login** d'un étudiant ? Par exemple, ajoutez l'instruction suivante :

```
etudiant1.login = "LOGIN_QUI_NE_RESPECTE_PAS_LA_CONTRAINTES";
```

Si c'est le cas, vous ne respectez pas le principe d'encapsulation ! L'état de l'objet `etudiant1` ne sera pas **cohérent avec la demande**.

Vous devez mettre vos attributs en privé dans la classe **Etudiant** :

```
public class Etudiant {
    private String login;
    private String prenom;
    private String nom;
    private String adresse;
```

En tant que développeur, vous devez faire attention à l'état (la cohérence) de vos objets, donc TOUJOURS mettre les attributs en **private**.

CONSEIL : en plus du principe d'encapsulation, nous vous encourageons à utiliser les méthodes setter dans vos constructeurs pour éviter d'avoir à dupliquer du code dans le constructeur ET dans les setters. Exemple :

```
public Etudiant(String login, String prenom, String nom) {
    setLogin(login);
    setPrenom(prenom);
    setNom(nom);
}
```

Rectifier votre code en conséquence !

Exercice 2.3 : Génération d'une documentation

Objectifs R2-03 : création d'une documentation de votre programme javadoc.

Pour générer une documentation vous devez dans un premier temps décrire vos classes et les méthodes publiques intéressantes. Pour ce faire, il suffit de commencer à taper `/**` et entrée pour obtenir le cadre pour votre description. Par exemple :

```
/**
 * La classe Etudiant représente la notion d'étudiant (login, nom, prénom
 * et adresse).
 * ATTENTION des contraintes sur la notion d'étudiant :
 * - un étudiant doit toujours avoir un login en minuscule
 * - un étudiant doit toujours avoir un nom et un prénom avec
 *   le premier caractère en majuscule et les autres en minuscule.
 */
public class Etudiant {
```

```
/**
 * Vérifier l'existence d'un adresse pour l'étudiant
 *
 * @return Vrai si l'étudiant à une adresse, Faux sinon
 */
public boolean existAdresse() {
```

Décrire vos classes puis **générer** la documentation dans un dossier doc qui vous créerez dans votre projet (Tools > Generate JavaDoc...). Exemple de JavaDoc généré :

existAdresse

```
public boolean existAdresse()
```

Vérifier l'existence d'un adresse pour l'étudiant

Returns:

Vrai si l'étudiant à une adresse, Faux sinon

IMPORTANT Vous continuerez à mettre des descriptions sur vos classes et les méthodes intéressantes ET aussi des commentaires. Vous continuerez à générer votre documentation Java. Votre code sera évalué au cours de ce module. C'est la deuxième fois que je mets ce commentaire.

Exercice 2.4 : Ajouter des notes

Objectifs R2-01 : utilisation des tableaux et un peu d'algorithmique.

Ajouter à l'étudiant la possibilité d'avoir des notes (maximum 5) et un calcul de moyenne en fonction de celles-ci. Pour ce faire :

- ajouter un attribut tableau de notes à la classe **Etudiant**. Un étudiant à au plus 5 notes. Utilisez le type double.

```
notes = new double[5];
```

- ajouter un indice en attribut qui va déterminer le nombre de notes ajouté et bien sûr une méthode qui va ajouter une note au tableau de notes (5 maximum) :

```
public void addNote(double note) {
```

- ajouter une méthode **getMoyenne()** qui retourne la moyenne de l'étudiant

Modifier les classes **Etudiant** et **TestEtudiant** pour tester l'ajout des notes en prenant exemple de la trace donnée ci-après :

```
-----
Login : dupontp
Nom complet : Pierre Dupont
Mail : Pierre.Dupont@etu.univ-grenoble-alpes.fr
Groupe : A
Adresse : aucune
Moyenne : aucune note

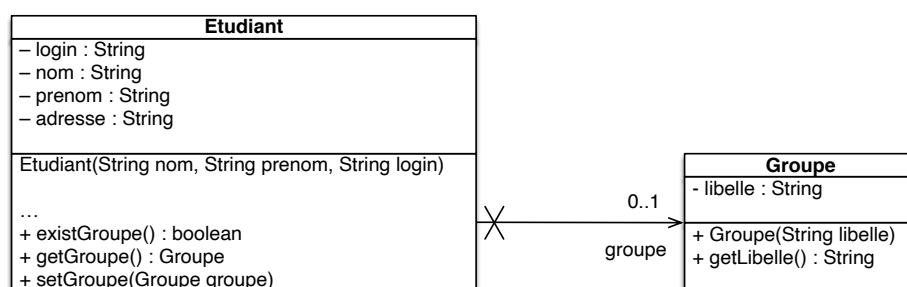
-----

Login : martinf
Nom complet : Francis Martin
Mail : Francis.Martin@etu.univ-grenoble-alpes.fr
Groupe : aucun
Adresse : 2 Place Doyen Gosse
Moyenne : 10.0
3 notes : 8.5 10.0 11.5
```

Exercice 2.5 : Associer des étudiants à des groupes

Objectifs R2-01 : associer des objets entre eux (association).

A ce stade de l'exercice, nous allons ajouter un groupe à un étudiant. Le diagramme UML ci-dessous représente le lien entre les deux classes. C'est une association entre **Etudiant** et **Groupe**. Un étudiant peut être dans 0 ou 1 groupe. Pour ce faire, la classe **Etudiant** aura un attribut groupe de Type **Groupe** (information donnée par le lien). En revanche, la classe **Groupe** ne connaîtra pas les étudiants qui le composent (information donnée par la croix sur le lien). Nous verrons dans les prochaines TP comment faire une association bidirectionnelle.



Créer la classe **Groupe** et **modifier** les classes **Etudiant** et **TestEtudiant**. Pour le test, ajouter le groupe A à l'étudiant Pierre Dupont. La trace attendue est la suivante :

```
-----  
Login : dupontp  
Nom complet : Pierre Dupont  
Groupe : A  
Adresse : aucune  
Moyenne : aucune note  
  
-----  
Login : martin  
Nom complet : Francis Martin  
Groupe : aucun  
Adresse : aucune  
Moyenne : 10.0  
3 notes : 8.5 10.0 11.5
```

Exercice 2.6 : Débogueur

Objectifs R2-03 : premier pas avec un débogueur.

« Un débogueur (ou débogueur, de l'anglais debugger) est un logiciel qui aide un développeur à analyser les bugs d'un programme. Pour cela, il permet d'exécuter le programme pas-à-pas, d'afficher la valeur des variables à tout moment, de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme ... »

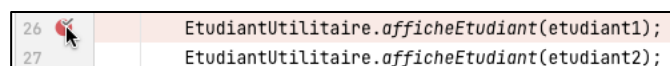
« Le programme à déboguer est exécuté à travers le débogueur et s'exécute normalement. Le débogueur offre alors au programmeur la possibilité de contrôler l'exécution du programme, en lui permettant par divers moyens de stopper (mettre en pause l'exécution du programme) et d'observer par exemple le contenu des différentes variables en mémoire. L'état d'exécution peut alors être observé afin, par exemple, de déterminer la cause d'une défaillance.

Quand l'exécution d'un programme est stoppée, le débogueur affiche la position courante d'exécution dans le code source original »¹

Tester le débogueur.

1. **Première étape** : placer des points d'arrêt (ou breakpoint) qui permettent une mise en pause de l'exécution du programme lorsque celle-ci atteint un point d'arrêt.

Par exemple, placez un point d'arrêt dans la classe **TestEtudiant** sur la méthode d'affichage (double cliqué dans la colonne à côté du numéro de ligne) :

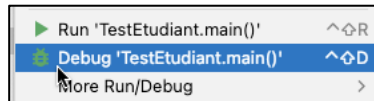


2. **Deuxième étape** : piloter et contrôler l'exécution du programme.

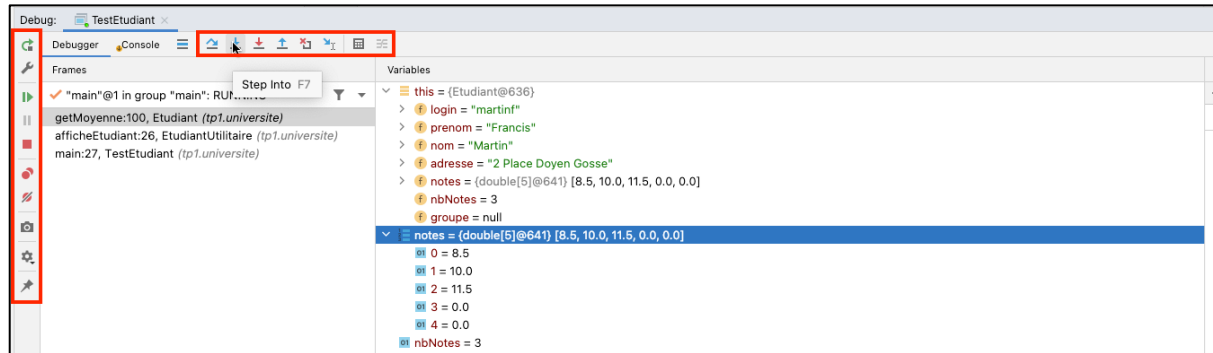
Par exemple, pilotez l'exécution de votre programme en faisant du pas-à-pas ou en entrant dans les méthodes appelées pour aller jusqu'à la méthode **getMoyenne()**.

Pour ce faire, lancer l'exécution en mode debug. Cliquez droit et lancer « Debug 'TestEtudiant.main()' » :






¹ source wikipedia.fr (<http://fr.wikipedia.org/wiki/debogueur>)



A ce moment-là, votre programme s'arrête dès qu'il rencontre un point d'arrêt. Un onglet Debug apparaît alors (voir ci-après). Cet onglet est composé de boutons de contrôle du mode débogage et de panneaux présentant l'état actuel de votre exécution et surtout l'état actuel des objets et des variables locales de la classe courante à l'arrêt de l'exécution.



Les boutons de contrôle (dans les rectangles rouge sur l'image) permettent de continuer l'exécution de votre programme et ainsi de déterminer l'origine d'un problème. Les plus importants :

-  « Resume Program » : Continuer l'exécution jusqu'au prochain breakpoint
-  « Stop Program » : arrêt de l'exécution
-  « Step Over » : pas-à-pas, une instruction est exécutée dans la classe courante ;
-  « Step Into » : entrer dans une méthode ou un constructeur appelé ;
-  « Step Out » : sortir de la méthode ou du constructeur appelé ;

Par rappel, pilotez l'exécution de votre programme en faisant du pas-à-pas ou en entrant dans les méthodes appelées pour aller jusqu'à la méthode **getMoyenne()**.

POURQUOI le faire sur un code qui fonctionne ? Pour prendre l'habitude !

POURQUOI utiliser le débogueur (sous entendu, je peux surement m'en sortir en mettant des **System.out.println()** de partout) ? Le débogueur est un outil très puissant qui permet d'exécuter le programme pas-à-pas et donc de vérifier si votre programme passe où vous le souhaitez et permet également d'afficher les valeurs des variables à tout moment.

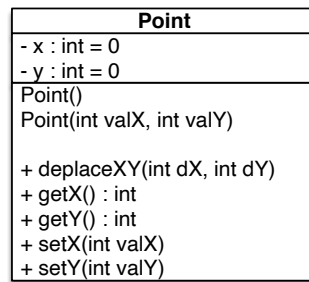
JE NE COMPRENDS RIEN demander à un enseignant de vous faire la démonstration

Exercice 3 : Gestion de composants graphiques 2D

Dans cet exercice, vous allez représenter des éléments graphiques 2D.

Exercice 3.1 : Point

Créer un package **forme** et dans ce package créer la classes **Point** représentée par le diagramme UML ci-dessous, la classe **TestPoint** et la classe **FormeUtilitaire**.



Créer les attributs, constructeurs et méthodes proposées dans le diagramme UML. Respectez le principe d'encapsulation (Pour aider, en UML, - représente un membre **private** et + un membre **public**) et utilisez les setter dans les constructeurs.

ATTENTION nous ajoutons une contrainte sur la notion de point : un point doit toujours avoir un x et un y supérieur ou égal à 0 : créer les méthodes en conséquence ! x et y ne pourront pas prendre de valeur négative, si une valeur négative est donnée, bornez à 0.

Explication de certaines méthodes du diagramme UML :

- **deplaceXY(int dX, int dY)** déplace le point de +dX et +dY. Si le point est en (2,3) et qu'il est déplacé de dX=1 et dY=5, le point sera alors en (3,8).

L'affichage d'un point se fera à l'aide d'une méthode **affichePoint(Point)** que vous créerez dans la classe **FormeUtilitaire**.

Créer et tester votre classe **Point**. La classe **TestPoint** et la trace attendue sont données ci-après.

Aide : le mot clé **this** est une variable de référence. Elle pointe vers l'objet courant de la classe. Elle sera le plus souvent utilisé dans une classe pour désambiguïser l'attribut d'une classe d'un paramètre de méthode, par exemple :

```
public void setX(int x) {
    if (x<0) {
        this.x = 0;
    } else {
        this.x = x; // l'attribut this.x prend la valeur du paramètre x
    }
}
```

Mais le mot clé **this** peut aussi être utilisé pour appeler un autre constructeur, par exemple :

```
public Point() {
    // Appel au constructeur Point(int x, int y)
    this(0,0);
}
```

Classe TestPoint :

```
public class TestPoint {

    public static void main(String[] args) {

        // créer un objet point1 de type Point
        // avec le constructeur par défaut Point()
        Point point1 = new Point();

        // créer un objet point2 de type Point
```

```

// avec le constructeur Point(int valX, int valY)
Point point2 = new Point(-1,4);

// afficher les Points avec la méthode affiche()
// de la classe FormeUtilitaire
System.out.println("-----");
System.out.println("Les points :");
FormeUtilitaire.affichePoint(point1);
FormeUtilitaire.affichePoint(point2);

// déplacer le Point point1 de dX=23 et de dY=-2
// déplacer le Point point2 de dX=-10 et de dY=20
System.out.println("-----");
System.out.println("Les points se déplacent");
point1.deplaceXY(23,-2);
point2.deplaceXY(-10,-6);

// afficher les Points
// avez vous vérifié si x et/ou y étaient négatifs ?
// un point doit toujours avoir un x et un y supérieur ou égal à 0
System.out.println("-----");
System.out.println("Les points :");
FormeUtilitaire.affichePoint(point1);
FormeUtilitaire.affichePoint(point2);
}
}

```

Trace attendue :

```

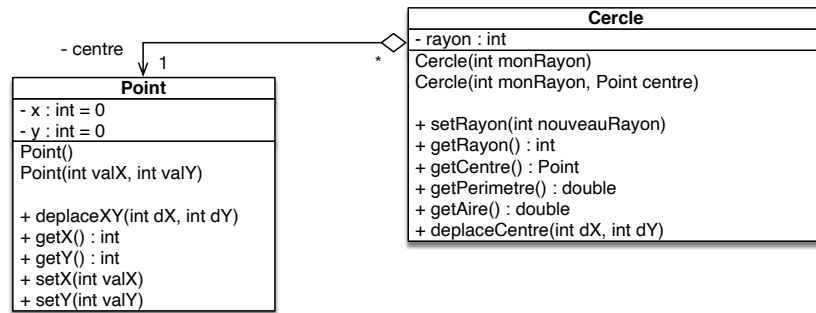
-----
Les points :
Point : x=0, y=0
Point : x=0, y=4
-----
Les points se déplacent
-----
Les points :
Point : x=23, y=0
Point : x=0, y=0

```

Exercice 3.2 : Cercle et point (agrégation)

Objectifs R2-01 : création d'objets, associer des objets entre eux (agrégation)

Dans le package forme, créer les classes **Cercle** représentée par le diagramme UML ci-dessous et **TestCercle**. Que représente le lien entre la classe **Cercle** et le **Point** ? C'est une **agrégation** —◇ : une association non symétrique, qui exprime un couplage fort et une relation de subordination. Pour faire simple, un **Cercle** a pour centre un **Point** et a donc un attribut java de type **Point** pour représenter son centre (en UML, on parle de rôle. Vous apprendrez tout cela en R2.01b). Ce point est passé en paramètre de constructeur et pourra être partagé avec d'autres objets.



Explication de certaines méthodes du diagramme UML :

- **getPerimetre()** et **getAire()** retournent respectivement le périmètre du cercle et son aire.
- **deplaceCentre(int dX, int dY)** déplace le centre du cercle de +dX et +dY. Si le centre est en (2,3) et qu'il est déplacé de dX=1 et dY=5, le centre sera alors en (3,8).

L'affichage d'un cercle se fera à l'aide d'une méthode **afficheCercle(Cercle)** que vous créerez dans la classe **FormeUtilitaire**.

Créer et tester votre classe **Cercle**. La classe **TestCercle** et la trace attendue sont données ci-après.

AIDE : Vous trouverez toutes les fonctions mathématiques dans la classe **Math**.

JAVADOC Math : <https://docs.oracle.com/javase/9/docs/api/java/lang/Math.html>

Classe TestCercle :

```

public static void main(String[] args) {

    // créer un point
    Point point = new Point(0,2);

    // créer deux cercles avec le même point pour centre
    Cercle cercle = new Cercle(10, point);
    Cercle cercle2 = new Cercle(2, point);

    // afficher le point et les deux cercles
    FormeUtilitaire.affichePoint(point);
    FormeUtilitaire.afficheCercle(cercle);
    FormeUtilitaire.afficheCercle(cercle2);

    // déplacer un cercle
    System.out.println("On déplace le point de dX=10 et dY=4");
    cercle.deplaceCentre(10,4);

    // afficher le point et les deux cercles
    FormeUtilitaire.affichePoint(point);
    FormeUtilitaire.afficheCercle(cercle);
    FormeUtilitaire.afficheCercle(cercle2);

}
  
```

Trace attendue :

```

Point : x=0, y=2
Cercle : R=10, Centre=(0,2), Périmètre=62.83185307179586,
Aire=314.1592653589793
Cercle : R=2, Centre=(0,2), Périmètre=12.566370614359172,
Aire=12.566370614359172
  
```

```

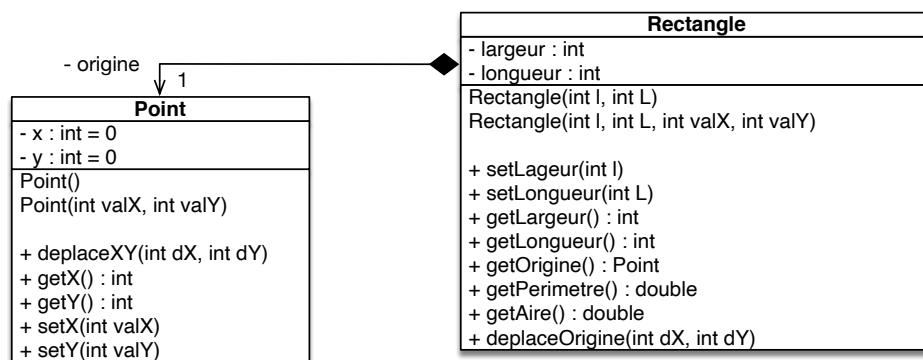
On déplace le point de dx=10 et dy=4
Point : x=10, y=6
Cercle : R=10, Centre=(10,6), Périmètre=62.83185307179586,
Aire=314.1592653589793
Cercle : R=2, Centre=(10,6), Périmètre=12.566370614359172,
Aire=12.566370614359172

```

Exercice 3.3 : Rectangle (composition)

Objectifs R2-03 : Prendre conscience des liens entre objets et de leurs conséquences

Dans le package forme, créer les classes **Rectangle** représentée par le diagramme UML ci-dessous et **TestRectangle**. Que représente le lien entre la classe **Rectangle** et le **Point** ? C'est une **composition** —◆: une agrégation forte. Pour faire simple, un **Rectangle** a pour origine un **Point**, a donc un attribut de type **Point** pour représenter son origine **MAIS l'usage du point devra être « caché » et « protégé » des autres classes**. C'est pour cela que le constructeur demandera les valeurs de X et Y en paramètre pour ensuite créer le point correspondant à son origine. Ce point ne devra être connu que de l'objet de type **Rectangle** dans lequel il a été construit.



Explication de certaines méthodes du diagramme UML :

- **getPerimetre()** et **getAire()** retournent respectivement le périmètre du rectangle et son aire.
- **deplaceCentre(int dX, int dY)** déplace l'origine du rectangle de +dX et +dY. Si l'origine est en (2,3) et qu'il est déplacé de dX=1 et dY=5, l'origine sera alors en (3,8).

L'affichage d'un rectangle se fera à l'aide d'une méthode **afficheRectangle(Rectangle)** que vous créerez dans la classe **FormeUtilitaire**.

Créer et tester votre classe **Rectangle**. La classe **TestRectangle** et la trace attendue sont données ci-après.

Classe **TestRectangle**

```

public class TestRectangle {

    public static void main(String[] args) {

        // créer un rectangle
        Rectangle rectangle = new Rectangle(2, 4, 2, 3);

        // afficher le rectangle
        FormeUtilitaire.afficheRectangle(rectangle);
    }
}

```

```

// déplacer le rectangle
System.out.println("On déplace l'origine de dx=10 et dy=4");
rectangle.deplaceOrigine(10,4);

// afficher le rectangle
FormeUtilitaire.afficheRectangle(rectangle);

// Récupérer l'origine
Point origine = rectangle.getOrigine();

// déplacer le point origine
System.out.println("On déplace le point donné par la méthode
getOrigine() de dx=-5 et dy=-6");
origine.deplaceXY(-5,-6);

// afficher le point et le rectangle
FormeUtilitaire.affichePoint(origine);
FormeUtilitaire.afficheRectangle(rectangle);
}
}

```

Trace attendue :

```

Rectangle : l=2, L=4, Origine=(2,3), Périmètre=12.0, Aire=8.0
On déplace l'origine de dx=10 et dy=4
Rectangle : l=2, L=4, Origine=(12,7), Périmètre=12.0, Aire=8.0
On déplace le point donné par la méthode getOrigine() de dx=-5 et dy=-6
Point : x=7, y=1
Rectangle : l=2, L=4, Origine=(7,1), Périmètre=12.0, Aire=8.0

```

IMPORTANT : un problème s'est glissé dans votre code. La méthode `getOrigine()` retourne l'objet de type `Point` utilisé par le rectangle. Toute modification de ce point, entraine une modification de l'objet de type `Rectangle` associé. Nous souhaitons mettre en place un lien de composition et donc « cacher » et « protéger » ce point. Comment faire avec une méthode `getOrigine()` qui retourne l'objet ? Il suffit de faire un clone de l'objet retourné !

Modifier la méthode `getOrigine()` pour qu'elle retourne un clone de l'objet origine : soit en créant un nouvel objet de type **Point**, soit, pour les plus intrépides, en passant par une méthode `clone()`. **Relancer** le test, normalement seul le point a été déplacé.

Trace attendue :

```

Rectangle : l=2, L=4, Origine=(2,3), Périmètre=12.0, Aire=8.0
On déplace l'origine de dx=10 et dy=4
Rectangle : l=2, L=4, Origine=(12,7), Périmètre=12.0, Aire=8.0
On déplace le point donné par la méthode getOrigine() de dX=-5 et dY=-6
Point : x=7, y=1
Rectangle : l=2, L=4, Origine=(12,7), Périmètre=12.0, Aire=8.0

```

Exercice « fil rouge » : La bataille de Faërun (étape 1)

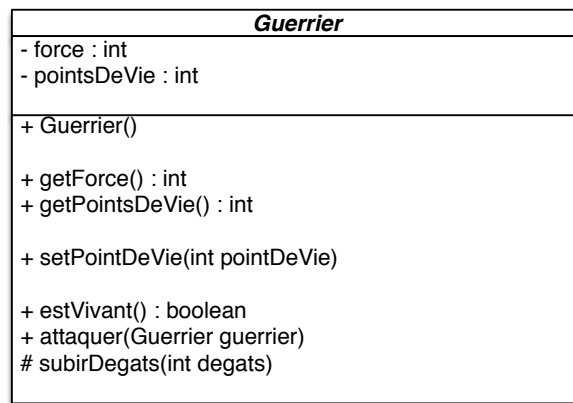
Cet exercice sera présent dans chaque TP et permettra de revenir sur les notions abordées dans les TDs et les TP sous la forme d'un petit jeu sur le terminal. Cette séquence n'est pas forcément à commencer dès les premières semaines, elle pourra être commencée à tout moment quand vous serez plus à l'aise avec la notion d'objet.

IMPORTANT : cet exercice sera évalué dans le cadre de la ressource R02.03 : bonne pratique, réalisation, documentation, gestion de version, test, etc.

Créer un package **jeu** au même niveau que **tp1** directement sous **src**.

Le but de cet exercice est de réaliser le mini-jeu Faërun. Les documents concernant ce jeu sont disponibles sur Chamilo dans le dossier « Jeu Faërun ». Dans cette première étape du jeu, vous devez **créer** la classe **Guerrier** (voir diagramme UML ci-après) :

- La **force** et les **pointsDeVie** seront initialisés respectivement à 10 et 100
- la méthode **estVivant()** retourne vrai si le guerrier a encore des pointsDeVie (> 0)
- la méthode **attaquer(Guerrier)** permet au guerrier courant d'attaquer le guerrier passé en paramètre et de lui faire subir des dégâts (voir le calcul des dégâts dans les documents).
- la méthode **subirDegats(int)** fait subir les dégâts passés en paramètre au guerrier courant : on soustrait les dégâts aux pointsDeVie.



Pour tester votre classe **Guerrier**, créer une classe **TestGuerrier** contenant :

- la création de 2 objets de type **Guerrier**
- une instruction décrivant une attaque du guerrier 1 sur le guerrier 2. Affichez les pointsDeVie restants du guerrier 2.
- Une itération simulant un combat entre ces 2 guerriers. L'arrêt de l'itération étant provoqué par la mort de l'un des 2 guerriers. Affichez un état des guerriers à chaque tour de l'itération.

La méthode d'affichage du guerrier est dans une classe **GuerrierUtilitaire**. Vous pourrez également y ajouter des méthodes **de3()** et **de3(int nombreLance)** qui respectivement retourne un entier entre 1 et 3 représentant un lancé d'un dé de 3 et retourne la somme du nombre de lancés de dé de 3 passé en paramètre.