Conception orientée objet

Bloc 1

Haute-École de Namur-Liège-Luxembourg

Langage de programmation orientée objet - Série 2

Objectifs

- Protéger l'accès aux données : appliquer le principe de l'Information Hiding
- Surcharger des constructeurs et des méthodes

Exercice 1 : Classe Rectangle et Information Hiding

Étape 1 : Déclarer privées les variables d'instance

Constatation

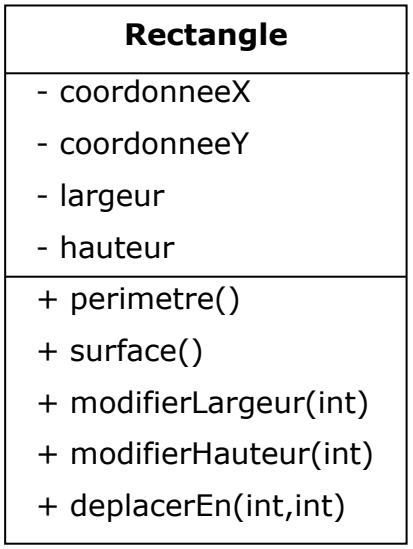
En laissant l'accès public aux variables d'instance, on permet au "monde extérieur" non seulement de les consulter, mais aussi de venir les modifier de n'importe quelle manière et donc d'y affecter éventuellement des valeurs erronées ou non valides.

De même, si on ne précise aucune protection (aucun mot-clé), la protection par défaut est de type package : les accès sont permis à partir du même package.

Le concepteur de la classe peut donc décider de cacher certaines parties de la classe au monde extérieur en déclarant **privées** ces parties, via le mot-clé `private`.

En général, on ne peut accéder aux valeurs des variables d'instance d'un objet que via ses méthodes (se référer à la notion d'**encapsulation** du cours de principes de programmation orientée objet).

Pour empêcher le monde extérieur d'accéder directement aux variables d'instance de la classe `Rectangle`, on les déclare avec la protection `private`. On ne pourra donc y accéder que via des méthodes, si du moins de telles méthodes sont prévues dans la classe.



Déclarez la classe `Rectangle` avec la protection `public`.

Déclarez toutes les variables d'instance de la classe `Rectangle` avec la protection `private`.

Déclarez le constructeur et les méthodes `avec` la protection `public`.

Reprenez la méthode `main` de la classe Principale qui contient les instructions ci-dessous (cf. série 1).

- La déclaration d'un objet appelé `premierRectangle` de type `Rectangle.`
- L'instanciation de `premierRectangle` en utilisant le constructeur (valeurs à transmettre lors de cette instanciation : 1 pour la coordonnée X, 2 pour la coordonnée y, 10 pour la largeur et 4 pour la hauteur).
- L'affichage (via `System.out.println`) des coordonnées x et y, la largeur et la hauteur de l'objet `premierRectangle.`
- La modification des variables d'instance de l'objet `premierRectangle` : 0 pour la coordonnée X, 3 pour la coordonnée y, 5 pour la largeur et 6 pour la hauteur.
- De nouveau, l'affichage des coordonnées x et y, de la largeur et la hauteur de l'objet `premierRectangle.`

Exécutez ce code et interprétez les erreurs détectées par le compilateur.

Étape 2 : Accesseurs / getters publics

But

On souhaite permettre au monde extérieur d'accéder en lecture et/ou écriture aux informations contenues dans certaines variables d'instance. On souhaite cependant que ces accès se fassent par l'intermédiaire d'appels de méthode.

Les méthodes permettant l'accès en lecture sont appelées des **getters** ou accesseurs. Par convention, le nom de ces méthodes commence par **get** suivi du nom de la variable d'instance. N'oubliez pas d'appliquer la camélisation.

Pour la syntaxe de ces méthodes, référez-vous au syllabus (cf. point 3.2).

Si on souhaite permettre l'accès en lecture aux coordonnées X et Y ainsi qu'à la largeur et la hauteur d'objets de type Rectangle après leur création, il faut définir les accesseurs/getters suivants : `getCoordonneeX()`, `getCoordonneeY()`, `getLargeur()` et `getHauteur()`.

À l'aide de ces méthodes, corrigez le code de la méthode `main` pour qu'il s'exécute correctement puis exécutez votre projet.

Étape 3 : Modificateurs / setters publics

Filtre

Pour permettre la modification de valeurs de variables d'instance privées, il faut implémenter des méthodes appelées **setters, modificateurs** ou encore mutateurs. Par convention, le nom de ces méthodes commence par **set** suivi du nom de la variable d'instance (+ camélisation).

Ne permettre l'accès en écriture aux variables d'instance que par l'intermédiaire d'appels de méthode a notamment pour avantage de pouvoir placer des filtres lors des accès en écriture. Ces filtres permettent de restreindre les valeurs permises à affecter aux variables d'instance.

Pour la syntaxe de ces méthodes, référez-vous au syllabus (cf. point 3.2).

Pour les besoins de l'exercice, partons du principe que l'on souhaite permettre de modifier les coordonnées X et Y ainsi que la largeur et la hauteur d'objets de type Rectangle déjà créés.

On va donc prévoir les setters suivants : `setCoordonneeX(…)` , `setCoordonneeY(…)`, `setLargeur(…)` et `setHauteur(…)`.

Les setters ont un rôle de filtre : dans ce cas-ci, **seul un nombre positif ou nul peut être affecté aux différentes variables d'instance**. Si une valeur négative est fournie en argument du setter, la valeur de la variable d'instance reste inchangée.

Valeurs par défaut des variables

Si une variable d'instance n'est pas initialisée explicitement, par exemple via un constructeur, elle contient une valeur par défaut. Les valeurs par défaut sont : **0** pour un nombre, **false** pour un booléen, '**\u0000**' pour une variable de type char et la référence **null** pour un objet (exemple, pour une variable d'instance de type String).

Testez ces méthodes en modifiant la méthode `main` : tentez une (ou plusieurs) modification(s) autorisée(s) puis affichez le contenu des variables d'instance modifiées (via appel aux getters correspondants).

Tentez ensuite d'affecter une valeur négative aux différentes variables d'instance et vérifiez que les setters jouent bien leur rôle de filtre (affichez de nouveau le contenu de ces variables d'instance).

Choix de conception

Il n'est pas obligatoire de prévoir des getters et setters pour toutes les variables d'instance déclarées privées. C'est le concepteur de la classe qui décide des variables d'instance qui seront rendues accessibles en lecture et/ou en écriture. Il peut décider de ne pas permettre l'accès en lecture et/ou en écriture à certaines variables d'instance privées.

Étape 4 : Utiliser les filtres dans le constructeur

Grâce aux setters, vous assurez que, une fois créé, il est impossible de modifier un rectangle en affectant une valeur négative à ses coordonnées ainsi qu'à sa largeur et à sa hauteur.

Réflexion

Vous avez cependant prévu un constructeur qui permet d'initialiser toutes les variables d'instance à la création du rectangle. Que faut-il faire pour s'assurer qu'on ne puisse pas créer de rectangle avec des valeurs négatives pour ses variables d'instance ?

Suggestion

Evitez de faire de simples affectations pour initialiser les variables d'instance au sein du constructeur ; faites appel aux méthodes adéquates pour affecter des valeurs aux variables d'instance.

Si vous ne trouvez pas, rendez-vous à la section 3.3 du syllabus.

⇨ Modifiez le constructeur de manière à ne pouvoir créer que des objets valides (pour rappel, seul un nombre positif ou nul peut être affecté aux différentes variables d'instance).

Tentez ensuite de créer un rectangle avec des valeurs non permises pour les variables d'instance, puis affichez les valeurs des variables d'instance de ce nouveau rectangle afin de vous assurer que les filtres des setters ont bien joué leur rôle.

Étape 5 : Utiliser les filtres dans les méthodes

Pour rappel, la méthode `modifierLargeur` permet d'adapter la largeur d'un rectangle en l'augmentant ou en la diminuant d'une valeur donnée en argument.

Si l'on fournit une valeur **négative** en argument, le rectangle rétrécit en largeur. Un rectangle ne peut cependant pas rétrécir indéfiniment ; sa largeur ne peut pas être inférieure à 0. Faites-en sorte qu'on ne puisse pas affecter de valeur finale négative à la largeur d'un rectangle via la méthode `modifierLargeur`, et ce, en appelant à bon escient le setter correspondant au sein de la méthode `modifierLargeur`.

Tentez ensuite d'affecter une valeur négative à la largeur d'un rectangle déjà créé en appelant sur cet objet la méthode `modifierLargeur` (avec une valeur suffisamment négative comme argument). Affichez ensuite la largeur de ce rectangle et vérifiez que celle-ci n'a pas été modifiée.

De même, que faut-il faire pour qu'on ne puisse pas affecter de valeur négative à la hauteur via la méthode `modifierHauteur` ?

Faut-il adapter de la même manière la méthode `deplacerEn` ?

Dans la classe `Principale`, appelez ces différentes méthodes sur l'objet `premierRectangle` :

- affichez à la console le périmètre et la surface de l'objet `premierRectangle` ;
- déplacez `premierRectangle` en appelant la méthode `deplacerEn` ;
- affichez ensuite de nouveau la largeur, la hauteur et les coordonnées x et y de `premierRectangle`.

Étape 6 : Surcharge du constructeur

Surcharge de constructeur

Plusieurs constructeurs peuvent coexister au sein d'une même classe. Ils portent tous le nom de la classe, mais se différencient par leur signature (nombre et/ou types des arguments différents). C'est ce qu'on appelle la surcharge de constructeurs.

Il existe une notation "raccourcie" qui permet d'appeler un autre constructeur de la même classe, ce qui favorise le point de modification unique. Il s'agit de l'instruction `this(…)`. Cette instruction ne peut se placer qu'au sein d'un constructeur (cf. point 5.1 du syllabus).

Prévoyez un second constructeur qui permettra de créer plus aisément des rectangles dont le point d'ancrage est (0,0) ; en effet, le programmeur qui appellera ce constructeur sera dispensé de donner des valeurs pour les coordonnées X et Y du point d'ancrage.

Le constructeur n'aura que deux arguments : la largeur et la hauteur du nouveau rectangle.

Utilisez l'instruction raccourcie (via `this(…)`) qui fait appel au constructeur à 4 arguments, en mettant les coordonnées X et Y à 0.

Dans la classe `Principale`, créez un nouveau rectangle de coordonnées (0,0). Affichez ensuite ses coordonnées x et y ainsi que sa largeur et sa hauteur.

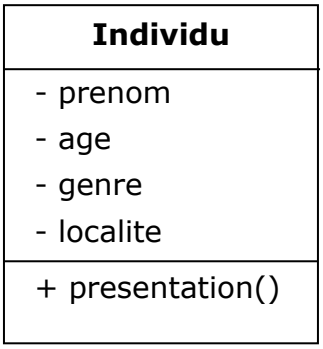
Exercice 2 : Classe Individu et Information Hiding

Étape 1 : Déclarer privées les variables d'instances

Déclarez la classe `Individu` avec la protection `public`.

Déclarez toutes les variables d'instance de la classe `Individu` avec la protection `private`.

Déclarez le constructeur et les méthodes avec la protection `public`.



Reprenez la méthode `main` de la classe `Principale` qui contient les instructions ci-dessous (cf. série 1).

- La déclaration d'un objet appelé `moi` de type `Individu`.
- L'instanciation de l'objet `moi` en utilisant le constructeur (utilisez les valeurs qui vous décrivent ou d'autres, au choix).
- L'affichage du prénom, du genre et de l'âge de l'objet `moi`.
- Une instruction modifiant le genre de `moi` en W et l'âge de `moi` en -124.
- De nouveau la même instruction d'affichage que ci-dessus.

Exécutez ce code et interprétez les erreurs détectées.

Étape 2 : Accesseurs / getters publics

Ne définissez que les accesseurs/getters nécessaires pour que la méthode `main` puisse s'exécuter.

À l'aide de ces méthodes, corrigez le code de la méthode `main` pour qu'il s'exécute correctement puis exécutez votre projet.

Étape 3 : Modificateurs / setters publics

Ne définissez que les modificateurs/setters nécessaires pour que la méthode `main` puisse s'exécuter.

Rôles de filtre de ces setters

- Si une valeur négative ou supérieure à 120 est fournie pour la variable d'instance `age`, la valeur existante reste inchangée.
- Si une valeur autre que M, F ou X est fournie pour la variable d'instance `genre`,
    - si la variable d'instance ne contient pas encore de valeur (rappel, elle contient '\u0000' si elle n'a pas encore été initialisée), affectez la valeur X ;
    - sinon, la valeur existante reste inchangée.

À l'aide de ces méthodes, corrigez le code de la méthode `main` pour qu'il s'exécute correctement puis exécutez votre projet.

Étape 4 : Adapter le constructeur

Assurez-vous également qu'on ne puisse pas non plus **créer** d'individus avec des valeurs erronées pour ses variables d'instance !

Tentez ensuite de créer un nouvel individu avec -3 comme valeur pour l'âge et R comme valeur pour le `genre`.

Afficher ensuite la valeur de la variable `genre` de cet individu ainsi que le résultat de l'appel de la méthode `presentation` sur cet objet et interprétez le résultat.

Étape 5 : Surcharge de méthode

Surcharge de méthode

On peut écrire dans une même classe plusieurs méthodes de même nom à condition que les signatures soient différentes, c'est-à-dire que le nombre et/ou les types des arguments soient différents. C'est ce qu'on appelle la surcharge de méthodes. Le type de retour des méthodes surchargées doit quant à lui être toujours le même (cf. point 5.2 du syllabus).

L'unilinguisme, c'est dépassé ! Écrivez une seconde méthode `presentation` qui reçoit un entier comme argument ; cet entier représente un code-langue : 1 pour français, 2 pour anglais et 3 pour néerlandais (si la méthode reçoit un autre code, elle retourne une chaîne de caractères vide). Selon le code reçu, la méthode doit retourner une des phrases ci-dessus.

En français :

Je m'appelle (prenom) et je suis âgé(e) de (age) an(s).
Je réside à (localite).

En anglais :

My firstname is (prenom) and I am (age) years old.
I live in (localite).

En néerlandais :

Ik heet (prenom) en ik ben (age).
Ik woon in (localite).

Point de modification unique

Si les deux méthodes `presentation` coexistent, peut-être peut-on coder une partie plus intelligemment ?

Pour rappel, la méthode `presentation` sans argument retourne la présentation en français.

Modifiez `main` pour tester la nouvelle version de `presentation` (avec les trois codes valables, puis avec un code erroné).

Exercice 3 : Classe BilletSpectacle et Information Hiding

Étape 1 : Déclarer privées les variables d'instances

Déclarez la classe `BilletSpectacle` avec la protection `public`.

Déclarez toutes les variables d'instance de la classe `BilletSpectacle` avec la protection `private`.

Déclarez le constructeur et les méthodes avec la protection `public`.

BilletSpectacle
- intituleSpectacle - dateSpectacle - categorie - coutDeBase - avecCarteEtudiant
+ prixBillet() + descriptionBillet()

Étape 2 : Accesseurs / getters publics

Reprenez un objet de type `BilletSpectacle` que vous avez créé dans la méthode `main` de la classe `Principale` (cf. série 1).

Tentez d'afficher l'intitulé du spectacle, la date du spectacle, la catégorie, le coût de base de cet objet de type `BilletSpectacle`. De même, tentez d'afficher le message "avec carte étudiant" si le booléen `avecCarteEtudiant` est à vrai, "sans carte d'étudiant" si le booléen est à faux.

Interprétez les résultats.

Définissez les accesseurs/getters nécessaires pour exécuter ces instructions d'accès en lecture.

À l'aide de ces méthodes, corrigez le code de la méthode `main` pour que les accès en lecture aux variables d'instance de l'objet de type `BilletSpectacle` s'exécutent correctement puis exécutez votre projet.

Étape 3 : Modificateurs / setters publics

Tentez de modifier la catégorie, le coût de base et la valeur du booléen `avecCarteEtudiant` de cet objet de type `BilletSpectacle`.

Interprétez les résultats.

Définissez les modificateurs/setters nécessaires pour exécuter les instructions d'accès en écriture.

Rôles de filtre de ces setters

- Si un coût de base < 1 est fourni en argument,
  - si la variable d'instance ne contient pas encore de valeur (elle contient 0 par défaut), affectez la valeur 1 ;
  - sinon, la valeur existante reste inchangée.
- Les seules catégories permises sont A, B et C. Si une autre catégorie est fournie en argument,
  - si la variable d'instance ne contient pas encore de valeur (elle contient '\u0000' par défaut), affectez la valeur C ;
  - sinon, la valeur existante reste inchangée.

Testez ces méthodes en modifiant la méthode `main` : tentez une (ou plusieurs) modification(s) autorisée(s) puis affichez le contenu des variables d'instance de l'objet de type `BilletSpectacle` ; tentez ensuite une (ou plusieurs) modification(s) interdite(s) et vérifiez que tout se déroule comme prévu.

Étape 4 : Adapter le constructeur

Assurez-vous également qu'on ne puisse pas non plus **créer** des billets de spectacle avec des valeurs erronées pour ses variables d'instance !

Tentez ensuite de créer un billet avec une catégorie incorrecte et un coût de base non valide. Affichez ensuite sa catégorie et son coût de base.

Étape 5 : Surcharge de méthode

Prévoyez une seconde méthode `prixBillet(int)` qui surchargera la méthode `prixBillet()`. Cette seconde méthode reçoit un pourcentage de réduction en argument et applique ce pourcentage au prix normal du billet.

Pensez au point de modification unique. La méthode sans argument doit donc maintenant faire appel à cette nouvelle méthode avec en argument un pourcentage de réduction de 0.

Affichez le prix sans réduction (en appelant la méthode `prixBillet()`) puis le prix avec réduction des billets que vous avez créés.

Étape 6 : Surcharge du constructeur

Prévoyez un second constructeur qui permettra de créer des billets de spectacle sans carte d'étudiant. Ce second constructeur n'aura donc que 4 arguments en entrée.

Pensez au point de modification unique. Faites donc appel au constructeur déjà écrit (via l'instruction `this(…)`).

Créez un nouveau billet sans carte d'étudiant en appelant le nouveau constructeur. Testez le booléen `avecCarteEtudiant` : s'il est à vrai, affichez le message "avec carte étudiant" si le booléen `avecCarteEtudiant` est à vrai.

Exercice 4 : Classe SejourDisney et Information Hiding

Étape 1 : Déclarer privées les variables d'instances

Déclarez la classe `SejourDisney` avec la protection `public`.

Déclarez toutes les variables d'instance de la classe `SejourDisney` avec la protection `private`.

Déclarez le constructeur et les méthodes avec la protection `public`.

SejourDisney
- nbEnfants - nbAdultes - nbJours - nbVehiculesParking - avecPMR
+ estLongSejour() + coutEntreeParc() + nbNuiteesEnfantsGratuites() + coutHotel() + coutParking() + coutTotal() + aAccesPrioritaire() + resumeSejour()

Étape 2 : Accesseurs / getters publics

Reprenez l'objet `sejour` de type `SejourDisney` que vous avez créé dans la méthode `main` de la classe `Principale` (cf. série 1).

Tentez d'afficher le nombre d'enfants, le nombre d'adultes, le nombre de jours et le nombre de places de parking souhaité de cet objet `sejour`. De même, tentez d'afficher le message "avec accès prioritaire" si le booléen `avecPMR` est à vrai.

Interprétez les résultats.

Définissez les accesseurs/getters nécessaires pour exécuter ces instructions.

À l'aide de ces méthodes, corrigez le code de la méthode `main` pour que les accès en lecture aux variables d'instance de l'objet `sejour` s'exécutent correctement puis exécutez votre projet.

Étape 3 : Modificateurs / setters publics

Tentez de modifier le nombre d'enfants, le nombre d'adultes, le nombre de jours, le nombre de places de parking souhaitées ainsi que la valeur du booléen `avecPMR` de cet objet `sejour`.

Interprétez les résultats.

Définissez les modificateurs/setters nécessaires pour exécuter ces instructions.

Rôles de filtre de ces setters

- Si une valeur négative est fournie en argument pour le nombre d'enfants, d'adultes ou de places de parking, la valeur existante reste inchangée.
- Si un nombre de jours < 1 est fourni en argument,
  - si la variable d'instance contient déjà une valeur >= 1, la valeur existante reste inchangée ;
  - sinon, affectez la valeur 1.

À l'aide de ces setters, corrigez le code de la méthode `main` pour que les accès en écriture aux variables d'instance de l'objet `sejour` s'exécutent correctement puis exécutez votre projet.

Testez ces méthodes en modifiant la méthode `main` : tentez une (ou plusieurs) modification(s) autorisée(s) sur l'objet `sejour` puis réaffichez le nombre d'enfants, le nombre d'adultes, le nombre de jours, le nombre de places de parking et la valeur du booléen `avecPMR`.

Tentez ensuite une (ou plusieurs) modification(s) interdite(s) et vérifiez que tout se déroule comme prévu en réaffichant les valeurs des variables d'instance de l'objet `sejour`.

Étape 4 : Adapter le constructeur

Assurez-vous également qu'on ne puisse pas non plus **créer** des séjours Disney avec des valeurs erronées pour ses variables d'instance !

Exercice 5 : Atelier découverte

Objectif spécifique
- Faire appel implicitement à la méthode toString

Étape 1 : Créer la classe AtelierDecouverte

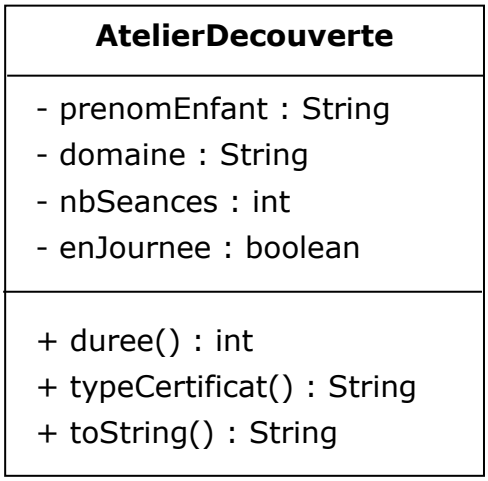
Créez dans `packageLoisir` la classe `AtelierDecouverte`.

Cette classe permet de gérer des ateliers de découverte suivis par des jeunes enfants en dehors des heures scolaires (mercredi après-midi, samedi, vacances scolaires et autres jours après 16h). Les domaines sont variés : dessin, nature, sport, musique, théâtre…

En fonction du nombre total d'heures suivies, au terme de l'atelier découverte, un certificat est décerné à l'enfant lui indiquant le niveau qu'il a atteint.

Un atelier découverte est décrit par un prénom d'enfant, un domaine, le nombre de séances choisies par l'enfant et s'il les suit en journée ou après 16h.

Déclarez les différentes variables d'instance avec la protection `private`.



Étape 2 : Constructeur

Créez un constructeur pour la classe `AtelierDecouverte` ; ce constructeur doit permettre d'initialiser toutes les variables d'instance.

Étape 3 : Surcharge du constructeur

Prévoyez un second constructeur qui permet de créer des ateliers découverte organisés **en journée**. Ce second constructeur n'aura donc que 3 arguments en entrée.

Pour rappel, pensez au point de modification unique. Faites donc appel au constructeur déjà écrit (via l'instruction `this(…)`).

Étape 4 : Méthodes

Prévoyez dans la classe `AtelierDecouverte` les méthodes ci-dessous.

- La méthode `duree` calcule le nombre total d'heures que compte l'atelier sachant que les séances en journée durent 4 heures, alors que les séances après 16h ne durent que 2 heures.
- La méthode `typeCertificat` retourne la chaîne de caractères décrivant le type de certificat (le niveau) qui est décerné à l'enfant à la fin de l'atelier : en dessous de 8 heures, le niveau est "Benjamin", au-dessus de 24 heures, le niveau est "Capitaine". Dans les autres cas, le niveau est "Explorateur".
- La méthode `toString` retourne la chaîne de caractères décrivant (l'état de) l'atelier en respectant la structure ci-dessous.

Pour un atelier organisé en journée :

↱ Appel à la méthode duree
Au terme d'un atelier de … heures organisé en journée
… a reçu le titre de … en … *⇨ Domaine*
↳ Prénom de l'enfant *↳ Appel à la méthode typeCertificat*

Pour un atelier organisé après 16 heures :

↱ Appel à la méthode duree
Au terme d'un atelier de … heures organisé après 16 heures
… a reçu le titre de … en … *⇨ Domaine*
↳ Prénom de l'enfant *↳ Appel à la méthode typeCertificat*

Méthode toString()

La méthode `toString` est une méthode dont le nom est un mot réservé et qui est appelée automatiquement chaque fois qu'un nom d'objet est placé là où on attend une chaîne de caractères. Cette méthode permet de décrire l'état d'un objet.

Pour plus d'explications sur cette méthode, référez-vous au syllabus (cf. point 4.4).

Étape 5 : Création d'objets et appels de méthode

Dans la méthode `main` de la classe `Principale`, écrivez les instructions ci-dessous (ne prévoyez que les getters/setters nécessaires pour exécuter ce code) :

- Créez et initialisez un objet appelé `atelier` de type `AtelierDecouverte` pour Louise qui a choisi 6 séances en théâtre organisées après 16h.
- Affichez à la console le prénom de l'enfant, le domaine et le nombre de séances à partir de cet objet `atelier` que vous avez créé.
- Écrivez un test qui permet d'afficher le texte "en journée" ou "après 16 heures" en fonction de la valeur du booléen `enJournee` de l'objet `atelier`.
- Affichez à la console la durée de l'objet `atelier` ainsi que son certificat.
- Affichez à la console la description de l'état de cet objet `atelier` ; pour ce faire, utilisez l'instruction `System.out.println(atelier)` qui appelle **implicitement** la méthode `toString`.
- Modifiez le domaine de l'objet `atelier` que vous avez créé.
- Modifiez le nombre de séances de l'objet `atelier` que vous avez créé.
- Adaptez l'objet `atelier` : il est organisé non plus après 16h mais en journée.
- Recopiez le test du troisième point et réexécutez-le, toujours sur l'objet `atelier` ; vérifiez que c'est la bonne chaîne de caractères qui est affichée.
- Affichez à nouveau à la console la description de l'état de l'objet `atelier` en utilisant l'instruction `System.out.println(atelier)` qui fait un appel **implicite** à la méthode `toString` et assurez-vous que les modifications ont bien été prises en compte.
- Créez un nouvel objet de type `AtelierDecouverte` en **initialisant le booléen `enJournee` à vrai** (veillez à utiliser le constructeur le plus adéquat) ; appelez la méthode `toString` sur cet objet.
- Créez d'autres objets de type `AtelierDecouverte` et affichez la description de leur état (via des appels implicites à `toString`).