TD 2 : Une interaction entre plusieurs classes

Dans ce TD nous aurons les classes suivantes : Etudiant, Console, JeuVideo, ainsi qu'une classe principale TD2. L'idée sera de donner la possibilité à un étudiant d'avoir une console, sur laquelle il pourra installer et désinstaller des jeux, jouer aux jeux (seul ou avec d'autres étudiants), etc.

Exercice I

Nous allons partir sur une classe `JeuVideo`. Les objets de cette classe auront les attributs suivants : un attribut `nomJeu` de type `String`, un attribut `nombreJoueurs` de type `int`, un attribut `prix` de type `double`, un attribut `typeConsole` de type `String`, ainsi qu'un attribut `tailleInstallation` de type `double`.

- Ecrivez un constructeur pour cette classe, qui aura la signature `JeuVideo(String, int, double, String, double)`. Ce constructeur instancie les valeurs des attributs de la classe `JeuVideo` aux valeurs données en entrée du constructeur (en ordre).

  Attention : cela veut dire qu'on suppose qu'on joue chaque jeu avec un certain nombre de joueurs à chaque fois : un joueur OU deux joueurs (mais pas les deux).
  De plus, on suppose que chaque objet de type `JeuVideo` sera utilisable pour seulement un type de console !

SOLUTION

```
	public JeuVideo(String nomJeu, int nombreJoueurs, double prix, String
typeConsole, double tailleInstallation) {
		this.nomJeu = nomJeu;
		this.nombreJoueurs = nombreJoueurs;
		this.prix = prix;
		this.typeConsole = typeConsole;
		this.tailleInstallation = tailleInstallation;
	}
```

- Nous aurons besoin de cinq méthodes (attention, on ne les écrira pas !) qui rendront les valeurs de chaque attribut. Quels seront les signatures de ces cinq méthodes ?

SOLUTION

Les méthodes qui rendent les valeurs de chaque attribut sont habituellement appelées getters, de l'anglais get. Par convention, une méthode qui rend un attribut appelé `nomAttribut` s'appellera `getNomAttribut()`.

La signature d'une méthode se compose de : la classe dans laquelle on la définit, le type de sortie, le nom de la méthode, ainsi que les types des variables en entrée (les paramètres). Les cinq signatures seront donc :

- La méthode `String getNomJeu()` dans la classe `JeuVideo`
- La méthode `int getNombreJoueurs()` dans la classe `JeuVideo`
- La méthode `double getPrix()` dans la classe `JeuVideo`
- La méthode `String getTypeConsole()` dans la classe `JeuVideo`
- La méthode `double getTailleInstallation()` dans la classe `JeuVideo`

Utiliser un autre nom que getNomAttribut() pour ce type de méthode n'aura aucun effet sur l'exécution du programme. Cependant c'est une très mauvaise pratique de ne pas respecter la convention : notamment si quelqu'un d'autre doit regarder ou utiliser notre code après, la convention l'aidera trouver ces méthodes et les utiliser plus facilement.

- Ecrivez une méthode `String toString()` qui rend les valeurs stockées par chacun des attributs d'un objet de type `JeuVideo`, dans le format suivant :

```
nomJeu : <le nom>
nombreJoueurs : <le nombre de joueurs>
prix : <son prix>
typeConsole : <le type de console>
tailleInstallation : <le nombre de Go nécessaires pour installer le jeu>
```

SOLUTION :

On est dans la classe JeuVideo. Dans la méthode toString() on veut retourner les valeurs actuelles des attributs caractérisant les objets de cette classe. Comme on est dans la classe, on a un accès direct aux attributs avec la syntaxe this.<nomAttribut>, sans faire appel aux getters.

La méthode qu'il faut écrire a la signature String toString(). Ceci veut dire qu'il faut retourner un String. Il nous faut donc aussi une méthode pour obtenir un retour de ligne lorsqu'on retourne un String. Ceci est réalisable en utilisant `"\n"`.

Voici donc la solution :

```
public String toString() {
            return("nomJeu : " + this.nomJeu + "\n" + "nombreJoueurs : "
+this.nombreJoueurs + "\n" + "prix : " + this.prix + "\n" + "typeConsole : " +
this.typeConsole + "\n" + "tailleInstallation : " + this.tailleInstallation + "Go");
      }
```

- Dans une autre classe TD2, dans une méthode `main` on a le code suivant :

```
        final JeuVideo marioKart = new JeuVideo("Mario Kart", 2, 19.99, "Wii U",
25.50);
        System.out.println(marioKart);
```

Décrivez dans vos propres mots ce que fait chaque ligne de ce code. Est-ce que le code compile. Est-ce qu'il s'exécute (quel est le résultat de l'exécution) ?

SOLUTION :

La première instruction crée un objet de type JeuVideo, dont le nom est `"Mario Kart"`. Il se joue à 2 joueurs sur une console de type `"Wii U"`, coûte 19.99 euros et prend 25.50 Go pour l'installation.

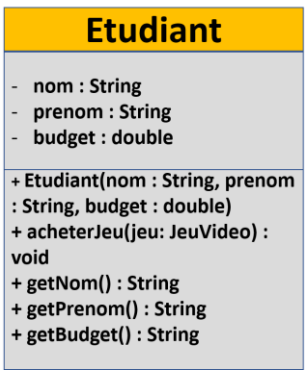
Lorsqu'on appelle la méthode System.out.println sur un objet (issu d'une certaine classe), Java cherche premièrement si la classe contient une méthode toString(). Le cas échéant, l'instruction System.out.println affiche la chaine de caractères retournée par la méthode toString(). Sinon, System.out.println affichera la référence mémoire de l'objet en question.

La deuxième instruction ci-dessus fait donc afficher le texte indiqué par la méthode toString(), appliqué à l'objet marioKart. Voici donc l'affichage :

```
nomJeu : Mario Kart
nombreJoueurs : 2
prix : 19.99
typeConsole : Wii U
tailleInstallation : 25.50Go
```

Exercice II

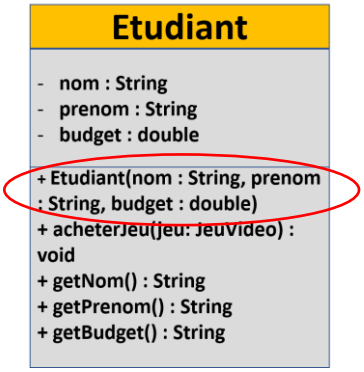
Nous allons avancer à une deuxième classe, `Etudiant`. Un etudiant (un objet de type `Etudiant`) aura la possibilité d'acheter des jeux à partir d'un budget qu'il a. Plus tard, on lui donnera la possibilité d'acheter des consoles, de faire installer ses jeux, et puis de jouer également. Mais, pour cet exercice notre but sera de réaliser la classe décrite par le diagramme ci-dessous.



- Indiquez le constructeur de cette classe sur le diagramme ci-dessus.

SOLUTION :

Le constructeur est indiquée ci-dessous (c'est la première méthode dans le diagramme de classe)



- Quels paramètres ce constructeur prend-il en entrée ?

SOLUTION :

Ce constructeur prend en entrée une variable `nom` de type `String`, une variable `prenom` de type `String` et une variable `budget` de type `double`.

- Nous allons considérer qu'on a déjà (correctement) écrit le code de toutes les méthodes de la classe étudiant sauf la méthode `void acheterJeu(JeuVideo jeu)`. Cette dernière doit vérifier si le paramètre en entrée est `null` (notamment `jeu` n'est pas un objet existant, déjà instancié). Le cas échéant on affiche le texte suivant : `Veuillez indiquer un jeu valide.` Sinon, on vérifie si le budget restant à l'etudiant suffit pour acheter le jeu (par rapport à la valeur stockée par l'attribut prix de l'objet `jeu`). Le cas échéant, on enlève le montant du prix du budget et on affiche le message : `Felicitations. Vous avez bien achete le jeu <nom du jeu> !` Si le budget ne suffit pas, affichez le message : `Le prix de ce jeu depasse votre budget !`

  Attention : on se rappelle que le langage Java gère parfois très mal les accents, donc on n'en met pas dans le texte.

SOLUTION :

Nous sommes dans la classe `Etudiant`. Ceci veut dire que nous avons un accès direct aux attributs de la classe `Etudiant` en utilisant `this.<nom de l'attribut>`. Pour le paramètre jeu de type JeuVideo, nous n'aurons pas un accès direct à ses attributs, qui sont privés. Là il faut plutôt utiliser les méthodes de type getter.

Il faut se rappeler que `null` n'est pas une valeur et n'est pas un objet. Le mot `null` se réfère à une référence vide en mémoire (l'objet en question n'existe pas et n'a jamais été initialisé).

Comment compare-t-on deux objets (en général) ? On se rappelle que lorsqu'on compare deux objets (par exemple deux `Strings`) on utilise le mot dédié `equals` (`if (string1.equals(string2))`). En particulier, on n'utilise pas == pour comparer deux objets. La syntaxe == fait une comparaison au niveau de la mémoire : autrement dit `objet1 == objet2` retourne `true` seulement si `objet1` et `objet2` partagent la même adresse mémoire. Il y a un seul cas où on utilise == lorsqu'on fait une comparaison avec un objet : notamment lorsqu'on le compare à `null`. Pourquoi ? Puisque dans ce cas-là on cherche justement si l'objet a été instancié, notamment s'il a une location mémoire.

Le code est donné ci-dessous.

Vous voyez qu'on écrit `if (null == jeu)` plutôt que if (jeu == null) . Ceci est un bonne pratique. Vérifier si l'objet est null ou non est nécessaire à ce point : notamment, si on ne verifie pas cela, nous risquons plus tard d'appeller une méthode pour l'objet `jeu`, ce qui lévera une `NullPointerException` Par contre, si `jeu` est en fait `null`, et si on fait cette vérification alors on n'aura pas cette exception.

```
	public void acheterJeu(JeuVideo jeu) {
		if (null == jeu) {
			System.out.println("Veuillez indiquer un jeu valide");
		}
		else {
			if(this.budget >= jeu.getPrix()) {
				this.budget -= jeu.getPrix();
				System.out.println("Felicitations. Vous avez bien achete le
jeu " + jeu.getNomJeu() + " !");
			}
			else {
				System.out.println("Le prix de ce jeu depasse votre budget
!");
			}
		}

	}
```

- Pourquoi affiche-t-on des messages d'erreur variés dans la méthode `acheterJeu` ?

SOLUTION :

Les messages d'erreur sont très utiles pour deux raisons principales :

1. Debugging : si on attend un certain résultat d'une méthode, mais elle retourne une autre chose, alors le message d'erreur nous dira exactement ce qui s'est passé et on aura moins de mal lorsqu'on veut améliorer le code.
2. Suivre une exécution : disons maintenant que le code marche bien (plus besoin de faire du debugging). Il peut nous être très utile de savoir comment le programme a progressé par rapport aux paramètre qu'on lui a mis en entrée.

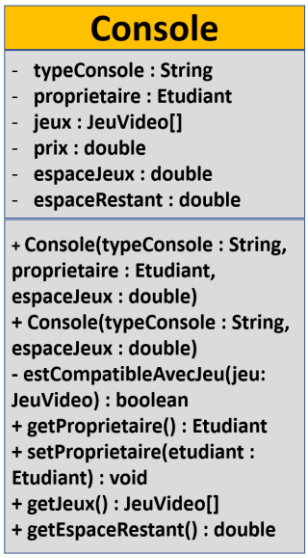
- Donnez le code (à mettre dans la classe TD2 dans la méthode `main`) qui permet de créer deux etudiants avec les caractéristiques suivantes :
    - Un premier etudiant s'appellant (prenom nom) Jean Dupont, avec un budget de 400 euros
    - Un deuxième etudiant s'appellant (prenom nom) Marie Lagrange, avec un budget de 550 euros

SOLUTION :

```
        final Etudiant jeanDupont = new Etudiant("Dupont", "Jean", 400);
        final Etudiant marieLagrange = new Etudiant("Lagrange", "Marie", 550);
```

Exercice III

Pour cet exercice nous allons créer une classe `Console`, qui permet de modéliser une console. Nous envisageons dans un premier temps une classe ressemblant celle du diagramme ci-dessous.



Les objets de type Console auront notamment six attributs :

- Un attribut `typeConsole` de type `String`
- Un attribut `proprietaire` de type `Etudiant` (celui-ci peut être mis à `null` si la console n'a pas encore un propriétaire)
- Un attribut `jeux` de type `JeuVideo[]` : on va supposer qu'une console ne peut pas stocker plus que 50 jeux vidéo en même temps.
- Un attribut `prix` de type `double` : celui-ci sera le prix, en euros, de la console

- Deux attributs de type `double` : `espaceJeux` et `espaceRestant`. Le premier stocke l'espace total (en Go que vous avez pour installer un jeu sur la console), le deuxième stocke l'espace restant (après un certain nombre d'installation). Le deuxième de ces deux attributs est susceptible de changer dans le cours de l'exécution, tandis que le premier restera constant.

Nous allons partir sur les constructeurs de cette classe. Trouvez-les sur le diagramme de classe.

- Pour le premier constructeur, nous voulons initialiser les valeurs des attributs `typeConsole`, `proprietaire`, `prix` et `espaceJeux` aux valeurs données en entrée (en ordre). De plus, on va initialiser le tableau `jeux` comme étant un tableau de type `JeuVideo[]` de taille 50 (pourquoi cette taille ?). Finalement, l'attribut `espaceRestant` sera initialisé à la même valeur que `espaceJeux`.

  Ecrivez ce constructeur.

SOLUTION :

Voici le constructeur souhaité :

```
	public Console(String typeConsole, Etudiant proprietaire, double prix, double espaceJeux) {

		this.typeConsole = typeConsole;
		this.proprietaire = proprietaire;
		this.jeux = new JeuVideo[50];
		this.prix = prix;
		this.espaceJeux = espaceJeux;
		this.espaceRestant = espaceJeux;
	}
```

Qu'est-ce que fait ce constructeur ? On lui a mis en paramètre 4 valeurs : le type de la console, le propriétaire, le prix et l'espace maximale qu'on peut utiliser pour installer les jeux. Pour l'instance la console est vierge, il n'y a pas de jeux dedans. Nous avons initialisé le tableau de jeux en tant que tableau de jeux vidéo de taille 50 puisqu'on a supposé (voir l'énoncé) qu'on ne peut pas avoir plus que 50 jeux sur la console en même temps.

- Le premier constructeur donne la possibilité de donner un paramètre un objet de type `Etudiant` déjà créé (ou la valeur `null`) comme le deuxième paramètre. Ceci modélise la possibilité de créer une console qui aura déjà un propriétaire. Pour le deuxième constructeur nous allons mettre par défaut la valeur du propriétaire à `null` (cela modélise une console sans propriétaire).

  Sans copier aucune ligne du code de votre premier constructeur, écrivez ce deuxième constructeur.

SOLUTION :

Si le propriétaire de la console sera mis à `null`, mais le fonctionnement du constructeur reste autrement le même que celui du premier constructeur, alors on voudrait bien que le constructeur fonctionne comme ci-dessous :

```
	public Console(String typeConsole, double prix, double espaceJeux) {

		this.typeConsole = typeConsole;
		this.proprietaire = null;
		this.jeux = new JeuVideo[50];
		this.prix = prix;
		this.espaceJeux = espaceJeux;
		this.espaceRestant = espaceJeux;
	}
```

Comme nous ne voulons pas copier le constructeur, nous allons juste appeler le premier constructeur (qui a plus de paramètres en entrée) dans le deuxième (avec moins de paramètres). Comment appelle-t-on le premier constructeur ? On utilise le mot dédié `this`, en mettant le deuxième paramètre à `null`.

```
	public Console(String typeConsole, double prix, double espaceJeux) {

		this(typeConsole,null,prix,espaceJeux);
	}
```

- La méthode `private boolean estCompatibleAvecJeu(JeuVideo jeu)` retourne `true` si le type de cette console est le même que le type de console indiquée parmi les attributs de l'objet `jeu`, et `false` autrement. Ecrivez cette méthode.

SOLUTION :

Nous sommes dans la classe `Console`. La méthode `estCompatibleAvecJeu` sera utilisée pour des objets de type `Console`. Nous voulons comparer le type de la console avec le type de console indiquée par l'objet de type `JeuVideo` en paramètre. Nous pouvons accéder aux attributs de l'objet de type `Console` directement (nous sommes dans la classe `Console`), mais pour l'objet de type `JeuVideo` en paramètre il faut qu'on utilise des getters.

Si on regarde de plus le diagramme de classe ci-dessus nous verrons également que cette méthode est censée d'être une méthode privée (il y a un petit - devant).

```
	private boolean estCompatibleAvecJeu(JeuVideo jeu) {
		return (jeu.getTypeConsole() == this.typeConsole);
	}
```

- Dans le diagramme de classe on a mis un petit - devant la méthode `estCompatibleAvecJeu`. Ceci veut dire que la méthode sera privée. Qu'est-ce que cela veut dire ?

SOLUTION :

On se rappelle qu'une méthode privée est seulement utilisable dans la classe dans laquelle elle est définie. La méthode `estCompatibleAvecJeu` est donc utilisable seulement dans la classe `Console`.

- Supposons maintenant que toutes les méthodes indiquées dans le diagramme de classe sont déjà (correctement) écrites. Dans la méthode `main` de notre classe TD2 on a déjà mis le code suivant :

```
final Etudiant jeanDupont = new Etudiant("Dupont", "Jean", 400);
final Etudiant marieLagrange = new Etudiant("Lagrange", "Marie", 550);
```

  Ecrivez le code qui nous permet maintenant de créer ces deux consoles :
  - Un objet `wiiU` de type `Console` qui a le `typeConsole` "`Wii U`" sans propriétaire, qui coûte 150 euros et a 150 Go de place.
  - Un objet `ps4` de type `Console` qui a le `typeConsole` "PS4" qui appartient à l'etudiant Marie Lagrange, qui coûte 200 euros et a 150 Go de place.

SOLUTION :

```
final Console wiiU = new Console("Wii U", 150, 150);
final Console ps4 = new Console("PS4", marieLagrange, 200, 150);
```

Nous utilisons le premier constructeur pour créer la console `ps4`, tandis que la console `wiiU` est définie en utilisant le deuxième constructeur. La console wiiU n'au aucun propriétaire, c'est pourquoi nous pouvons utiliser le deuxième constructeur.

Attention : pour le propriétaire de la console ps4 on utilise l'objet marieLagrange et non pas juste le nom et prénom de cet objet.

- On ajoute le code suivant au code écrit dans la question précédente :

```
final JeuVideo marioKart = new JeuVideo("Mario Kart", 2, 19.99, "WiiU", 25.50);
```

  Si on appelle (dans la classe `Console`) la méthode `estCompatibleAvecJeu(marioKart)` pour les objets `wiiU` et `ps4` créés dans la question précédente, quels seront les deux résultats ?

SOLUTION :

On se rappelle que la méthode `estCompatibleAvecJeu` est privée et donc seulement utilisable dans la classe `Console`. Cette méthode peut être appelée pour des objets de la classe `Console`, y compris les objets `wiiU` et `ps4`. La console `wiiU` est définie avec le type de console `"Wii U"`, tandis que la console

ps4 est définie avec le type de console "PS4". La méthode estCompatibleAvecJeu vérifie si le type de console est le même que celui indiquée parmi les attributs du jeu marioKart. Pour ce dernier on a défini le type de console en tant que "WiiU". Clairement ps4.estCompatibleAvecJeu(marioKart) retournera false. Mais wiiU.estCompatibleAvecJeu(marioKart) retournera false également, car la console a un espace entre Wii et U dans le type, contrairement au jeu, qui n'a aucun espace.

- Dans la classe Etudiant on veut maintenant écrire une méthode qui permet à un objet de ce type de devenir le propriétaire d'un objet de type Console. Cette méthode aura la signature void acheterConsole(Console console). Nous voulons que cette méthode fonctionne ainsi : si le paramètre en entrée est valide (non-null) et si l'etudiant en question a assez d'argent, alors il deviendra le propriétaire de la console en question (et il paiera le prix indiqué pour la console). Si le paramètre en entrée n'est pas valide (il est null) alors on affiche un message d'erreur indiquant qu'il faut bien saisir un paramètre valide en entrée. Finalement, si le paramètre donné est valide, mais l'etudiant n'a pas assez d'argent, alors on donne un autre message d'erreur indiquant cela.

  Complétez le code suivant pour cette méthode :

```
public void acheterConsole(Console console) {
	if (null != console && this.budget >= console.getPrix()) {
		// A completer

	}
	else {
		if (// A completer) {
			System.out.println("Pas assez d'argent !");
		}
		else {
			System.out.println("Il faut bien specifier une console !");
		}
	}
}
```

SOLUTION :

On regarde premièrement le code donné. Nous sommes dans la classe Etudiant. La première condition vérifie si le paramètre en entrée est bien défini et si le budget de l'étudiant (this.budget) suffit pour acheter la console (par rapport à console.getPrix()). Le cas échéant, le budget de l'étudiant sera diminué par le prix de la console et l'étudiant deviendra le propriétaire de la console. La première de ces deux instructions sera simplement this.budget -= console.getPrix(); . Pour la première instruction il nous faut une façon d'imposer à un objet de type console un propriétaire. Une façon de réaliser cela est d'appeler le constructeur -- mais ceci créerait un nouvel objet, et ce n'est pas ce qu'on veut faire.
En regardant le diagramme de classe de Console, on s'aperçoit qu'il existe une méthode setProprietaire(Etudiant etudiant). Si les méthodes de type getNomAttribut s'appellent getters, les méthodes de type setNomAttribut(<ClasseAttribut> valeurAttribut) s'appellent setters. Traditionnellement les setters mettent à jour la valeur d'un attribut à une valeur

souhaitée. Suivant cette convention, alors, la méthode `setProprietaire(Etudiant etudiant)` de la classe `Console` nous permettra justement de mettre à jour l'attribut `Proprietaire` de l'objet console. Ce qui donne le code :

```
	if (null != console && this.budget >= console.getPrix()) {
		console.setProprietaire(this);
		this.budget -= console.getPrix();
	}
```

On consulte le reste du code et nous voyons deux messages -- l'un qui spécifie que l'étudiant n'a pas assez d'argent pour acheter la console et un autre qui demande que l'objet en entrée soit valide. Nous sommes dans la partie `else` du premier `if` : ceci veut dire que le code s'exécute, soit si (`null == console`), soit si (`this.budget < console.getPrix()`). C'est ce qu'on cherche prochainement.

Le premier message écrit ci-dessus indique « Pas assez d'argent ». Nous sommes donc dans la situation (`this.budget < console.getPrix()`). Le code deviendra :

```
	public void acheterConsole(Console console) {
		if (null != console && this.budget >= console.getPrix()) {
			console.setProprietaire(this);
			this.budget -= console.getPrix();
		}
		else {
			if (this.budget < console.getPrix()) {
				System.out.println("Pas assez d'argent !");
			}
			else {
				System.out.println("Il faut bien specifier une console !");
			}
		}
	}
```

Exercice IV

Nous avons plusieurs choix lorsqu'on veut programmer les interactions entre les trois classes, notamment l'installation/la désinstallation des jeux sur la console, jouer un jeu, etc. Par exemple, un choix pourrait-être de mettre ce code dans la classe `Etudiant` (car on dit que c'est un `etudiant` qui fait installer le jeu sur la console). Nous allons choisir autrement : notamment, on va inclure la partie principale du code d'installation et de jeu dans la classe `Console`, car en fait installer le jeu a plus à faire avec ces deux classes (l'espace restant, le type de console, etc.).

- Dans la classe `Console`, nous allons avoir une méthode `void installerJeu(JeuVideo jeu)`. On va commencer par se demander si la console et le jeu sont compatibles (sinon, on envoie un message d'erreur décrivant cela). Puis, on va chercher si on a encore de la place par

rapport au nombre de jeux déjà existants sur la console -- sinon, on envoie encore un message d'erreur qui indique cela. Si on a assez de place, on fait installer le jeu : le jeu est ajouté au tableau `jeux` de l'objet de type `Console`, on diminue l'espace restant sur la console par la place prise par le jeu (voir l'attribut `tailleInstallation` dans la classe `JeuVideo`), et on affiche un message qui indique le succès de cette méthode.

Ecrivez le code de cette méthode.

SOLUTION :

Nous sommes dans la classe `Console`. La signature de la méthode à écrire est `void installerJeu(JeuVideo jeu)`, ce qui veut dire qu'on prend en entrée un objet de type `JeuVideo`. De plus cette méthode concerne l'attribut `jeux` de la classe `Console`, qui est un tableau d'objets du type `JeuVideo`.

Comment ajoute-t-on un objet `jeu` au tableau `jeux` ? Premièrement il faut chercher si on a encore de la place (voire aussi le TD précédent). En parcourant le tableau, on vérifie pour chaque élément s'il existe ou s'il a été mis à `null`. Si l'élément est `null`, alors cela veut dire qu'on a de la place pour y mettre le jeu en entrée.

On pourrait utiliser une boucle `for` pour parcourir le tableau. Mais ceci n'est pas efficace, car on veut s'arrêter du moment où on trouve une place disponible. On fera donc une boucle `while`, avec une valeur booléène qui nous indiquera si on a trouvé la place. Ceci nous donne la boucle suivante :

```
	int indexActuel = 0;
	boolean placeTrouve = false;
	while (indexActuel < this.jeux.length && !placeTrouve) {
		if (null == this.jeux[indexActuel]) {
			// on a trouve une place disponible
			// est-ce qu'on a assez de place pour l'installation ?
			if (espaceRestant >= jeu.getTailleInstallation()) {
				// on a assez de place pour l'installation
				// on met le jeu dans le tableau
				// on diminue la place disponible sur la console
				// on indique le fait d'avoir trouve de la place
				// et on informe le joueur sur le progrès
				this.jeux[indexActuel] = jeu;
				this.espaceRestant -= jeu.getTailleInstallation();
				placeTrouve = true;
				System.out.println("Jeu " + jeu.getNomJeu() + " installe
!");
			}
			else {
				// pas assez de place pour l'installation
				System.out.println("Ce jeu est trop large. Non installe.");
			}

		}
		// l'iterateur de la boucle while
		indexActuel++;
	}
```

Nous allons observer qu'on se réfère à l'attribut `jeux` par `this.jeux`. Ceci est une bonne pratique, même s'il n'y aurait aucune confusion même si on l'écrivait juste `jeux`.

La méthode sera donc :

```
public void installerJeu(JeuVideo jeu) {
	if (this.estCompatibleAvecJeu(jeu)) {
		int indexActuel = 0;
		boolean placeTrouve = false;
		while (indexActuel < this.jeux.length && !placeTrouve) {
			if (null == this.jeux[indexActuel]) {
				if (espaceRestant >= jeu.getTailleInstallation()) {
					this.jeux[indexActuel] = jeu;
					this.espaceRestant -= jeu.getTailleInstallation();
					placeTrouve = true;
					System.out.println("Jeu " + jeu.getNomJeu() + "
installe !");
				}
				else {
					System.out.println("Ce jeu est trop large. Non
installe.");
				}

			}
			indexActuel++;
		}
		if (!placeTrouve) {
			System.out.println("On n'a pas assez de place pour installer ce
jeu. Desinstaller un autre peut-etre ?");
		}
	}
	else {
		System.out.println("Jeu pas compatible avec console. Non-installe.");
	}
}
```

- Dans la classe `Console` on aura également une méthode `void jouer(Etudiant[] joueurs, JeuVideo jeu)`, qui devrait simuler l'action des étudiants dans le tableau `joueurs` de jouer au jeu indiqué dans le deuxième paramètre. Dans cette méthode il faut vérifier si le jeu indiqué est valide ; le cas échéant on vérifie si le nombre de joueurs indiqué par le paramètre joueurs est bien le même que le nombre de joueurs indiqué par l'attribut correspondant dans la classe `JeuVideo`. Si tout va bien on affiche un message de succès, sinon on envoie un message d'erreur.

  Complétez le fragment de code ci-dessous avec le bon code :

```
public void jouer(Etudiant[] joueurs, JeuVideo jeu) {
	if (null == jeu) {
		System.out.println("On ne peut pas jouer a un jeu non-valide.");
	}
```

```
            else {
                  if (// A COMPLETER) {
                        // A COMPLETER


                  }
                  else {
                        System.out.println("On commence le jeu. Amusez-vous bien
!");
                  }
            }
      }
```

SOLUTION :

Nous commençons en inspectant le code. Nous nous rappelons que ce code se trouve dans la classe Console. Le premier if est facile à comprendre : on cherche juste savoir si le jeu en entrée est `null` (non-valide).

À partir de maintenant nous sommes sûrs que le jeu en entrée est valide. Pour le deuxième `if`, il semble que la branche `else` indique un succès : notamment on peut jouer au jeu en entrée. En regardant la description de la méthode ci-dessus, il semble que ce deuxième if est censé vérifier que la taille du tableau d'étudiants en entrée a la même taille que le nombre de joueurs indiquée par le jeu. Ce qui indique la méthode suivante :

```
      public void jouer(Etudiant[] joueurs, JeuVideo jeu) {
            if (null == jeu) {
                  System.out.println("On ne peut pas jouer a un jeu non-valide.");
            }
            else {
                  if (jeu.getNombreJoueurs() != joueurs.length) {
                        System.out.println("Ce jeu est a jouer a " +
jeu.getNombreJoueurs() + " joueurs. Vous etes a " + joueurs.length);


                  }
                  else {
                        System.out.println("Amusez-vous bien !");
                  }
            }
      }
```

- (Le petit extra) On va ajouter des méthodes dans la classe `Etudiant` qui appelleront les méthodes que nous venons d'écrire dans la classe `Console`.

  Ecrivez les deux méthodes décrites ci-dessous :

  - Une méthode avec la signature `void jouerSinglePlayer(Console console, JeuVideo jeu)`, qui permettra à l'etudiant de jouer un jeu tout seul sur la console.

- Une méthode avec la signature `void jouerMultiPlayer(Console console, JeuVideo jeu, Etudiant[] autreJoueurs)`, qui permettra à l'etudiant de jouer avec les joueurs listés dans `autreJoueurs`.

SOLUTION :

Pour toutes les deux méthodes nous allons utiliser la méthode écrite pour l'exercice précédent, notamment la méthode `void jouer(Etudiant[] joueurs, JeuVideo jeu)` de la classe `Console`. Nous nous rappelons que les joueurs du jeu sont tous inclus dans le tableau joueurs donné en entrée. Au contraire, les méthodes `jouerSinglePlayer` et `JouerMultiPlayer` sont censées d'être appelées pour un objet de la classe Etudiant. Celui-ci fera partie du groupe de joueurs qui jouera au jeu donné. L'astuce sera donc de mettre ce joueur avec ses compagnons dans un seul tableau, puis appeler la méthode jouer pour la console donnée en entrée.

Prenons la méthode `jouerSinglePlayer(Console console, JeuVideo jeu)`. Cette méthode permettra à l'étudiant pour lequel on appelle la méthode de jouer au jeu donné.

Notre premier pas sera de créer un tableau à un seul élément : l'étudiant actuel, `this`. Puis, nous appellerons la méthode jouer pour la console donnée, en mettant en paramètre le tableau à un élément et le jeu en entrée. Nous avons plusieurs choix pour créer le tableau à un etudiant. Une façon de le faire serait de premièrement déclarer la variable (et lui donner un nom), puis de l'instacier à la valeur `this`. Notamment :

```
Etudiant[] mesJoueurs = new Etudiant[1];
mesJoueurs[0] = this;
```

Mais on peut faire mieux que ça. Dans cette situation nous connaissons déjà les contenus du tableau, donc on peut l'instancier directement, par exemple en utilisant :

```
Etudiant[] mesJoueurs = new Etudiant[] {this};
```

Nous avons vu que cette syntaxe a un double rôle : elle instancie le tableau et en même temps lui donne la taille demandée. Dans notre cas, nous pouvons faire encore mieux. Il faut noter que notre tableau est à utiliser comme paramètre, et on peut donc ne pas lui donner un nom. Nous allons notamment utiliser la syntaxe ci-dessous :

```
public void jouerSinglePlayer(Console console, JeuVideo jeu) {
	if (null != console) {
		console.jouer(new Etudiant[]{this}, jeu);
	}
	else {
		System.out.println("Il faut bien specifier une console.");
	}
}
```

Vous voyez dans le code donné qu'on vérifie à priori si l'objet `console` donné en entrée existe (n'est pas `null`). Ceci est nécessaire car on veut appeler une méthode pour cet objet. Si console était `null`, alors l'appel pourrait causer une erreur de type `NullPointerException`.

Passons à la méthode multijoueur. Là, un tableau de joueurs existe déjà, mais il ne contient pas tous les joueurs. Un joueur manque : celui pour lequel on appelle la méthode. On commence en se demandant : peut-on juste ajouter l'étudiant actuel, `this`, à `autresJoueurs` ? La réponse est non, car `autresJoueurs` a déjà une taille fixe et nous voulons y ajouter un élément. C'est un inconvénient des tableaux. La seule solution serait de créer un nouveau tableau à partir de zéro et y mettre les éléments souhaités.

Attention : c'est faux d'écrire `Etudiant[] joueurs = new Etudiant[] {this, autresJoueurs};` ! Nous ne pouvons pas utiliser la syntaxe `new Etudiant[] {<liste d'éléments séparés par une vergule>}` avec un élément de type `Etudiant[]` (notamment `autresJoueurs`). La solution sera donc de créer un nouveau tableau, y ajouter premièrement l'élément actuel (`this`) au tableau sur la première position, puis on va parcourir le tableau `autresJoueurs`, en l'ajoutant élément par élément au nouveau tableau créé.

Voici donc le code de la méthode :

```
	public void jouerMultiPlayer(Console console, JeuVideo jeu, Etudiant[]
autresJoueurs) {
		if (null != console) {
			Etudiant[] joueurs = new Etudiant[autresJoueurs.length + 1];
			joueurs[0] = this;
			for (int i = 0; i<autresJoueurs.length; i++) {
				joueurs[i+1] = autresJoueurs[i];
			}
			console.jouer(joueurs, jeu);
		}
		else {
			System.out.println("Il faut bien specifier une console.");
		}
	}
```