Nanor Janjikian (1901777)
Stephanie Mansour (1935595)

Lundi le 3 décembre 2018

INF1010
Questions TP6

1. La solution du tp6 utilise deux exceptions : ExceptionArgumentInvalide et ExceptionUtilisateurInexistant dans le projet QT. On aurait pu mieux gérer les erreurs en créant d'autres classes d'exceptions. Identifier deux autres classes d'exception et justifier leur rôle dans la solution du tp6.

Nous aurons pu créer ces deux autres classes pour mieux gérer les erreurs :

ExceptionUtilisateurExistant→ classe qui vérifie si un utilisateur qu'on tante ajouter existe déjà. Si oui, un signal est envoyé ne permettant pas de l'ajouter (pas de duplication).

ExceptionUtilisateurGrouper→ classe qui vérifie si un utilisateur régulier qu'on veut ajouter dans un groupe existe déjà dans celui-ci. Si oui, un signal est envoyé ne permettant pas d'ajouter le même utilisateur dans le groupe.

2. Pourquoi, il n'y a aucune section SIGNAL dans les définitions des classes du TP6?

Il n'y a aucune section SIGNAL dans les définitions des classes du TP6, car, dans ce cas, les signaux sont générés par les changements fait par l'utilisateur, et non les changements dans les classes elles-mêmes (ce qui est toujours le cas pour QT).

3. Pourquoi, il n'y a aucun appel à `emit` dans la liste des instructions de l'implémentation des classes du TP6.

Parce que, comme expliqué dans la question 2, il n'y a aucun signal à émettre dans les classes elles-mêmes, il n'y a donc aucun signal à émettre. Les appels à 'emit' sont faits seulement lorsqu'il y a un changement générer par l'utilisateur.

4. Dans la méthode `MainWindow::setUI()`, que représente `QListWidgetItem*` dans le connect

```
connect(listeDepenses_,
SIGNAL(itemClicked(QListWidgetItem*)),                    this,
SLOT(selectionnerDepense(QListWidgetItem*)));
```

QListWidgetItem* représente le type de l'objet sur lequel le signal (itemClicked) et le slot (selectionnerDepense) auront un effet.

5. Dans le TP6, identifier la classe qui représente le Modèle et la classe qui représente les Vue et Contrôle.

Modèle : classe groupe puisqu'elle gère les données du programme

Vue et contrôle : classe MainWindow puisqu'elle gère les actions de l'utilisateur (contrôle) et est en charge de l'esthétique du programme (vue).

6. Dans l'instruction `connect`, il y a 4 paramètres. Expliquer le type et le rôle de chacun des paramètres.

Si on prend l'exemple de l'instruction connect suivante:
`Qobject ::connect(&c1, SIGNAL(valueChanged(int)), &c2, SLOT (setValue(int)));`

Premier paramètre :
`c1` représente l'émetteur du signal
Deuxième paramètre :
`SIGNAL(valueChanged(int))` détermine quel signal venant du émetteur sera géré.
Troisième paramètre :
`c2` représente le receveur du signal
Quatrième paramètre :
`SLOT(setValue(int))` représente le slot du receveur (troisième paramètre) qui sera appelé lorsque le signal (deuxième paramètre) est émis.

7. Dans le programme suivant, il y a des erreurs de compilation et d'exécution. Corriger-les.

```
class AA {
public:
	AA(int* p = nullptr) : attributA_(p) {};
	int * getAttributA_() { return attributA_; }
private:
	int* attributA_;
};

int main(){
	int* p(new int(10));
	unique_ptr<AA> un(AA (p));
	cout << *(un->getAttributA_()) << endl;
	unique_ptr<AA>  deux(AA(p));
	deux = un;
	cout << *(deux->getAttributA_()) << endl;
	cout << *(un->getAttributA_()) << endl;
	delete p;
	delete un;
	delete deux;
	return 0;
}
```

1) On ne peut pas passer l'objet de type AA au constructeur `unique_ptr<AA>`, il faut passer un pointeur vers l'objet de type AA (toujours allouer un pointeur et le passer en paramètre lorsqu'on utilise le constructeur `unique_ptr`)
2) Puisqu'on est en train de manipuler des unique_ptr, on doit utiliser un `move()` pour pouvoir transférer la ressource à un nouvel objet
3) Impossible de faire appel à l'unique_ptr `un` puisqu'il a été copié à l'unique_ptr `deux`. Lorsqu'il est copié, il devient invalide (pointe vers 0).
4) unique_ptr désalloue automatiquement la mémoire associée à la sortie du programme (il fait la gestion de la mémoire). Il est donc impossible de faire appel à delete sur `un` et `deux`.

Voici la correction du programme :

```
class AA
{
public:
	AA(int* p = nullptr) : attributA_(p) {};
	int * getAttributA_() { return attributA_; }
private:
	int* attributA_;
};

int main(){
	int* p(new int(10));
	unique_ptr<AA> un(new AA (p));
	cout << *(un->getAttributA_()) << endl;
	unique_ptr<AA>  deux(new AA(p));
	deux = move(un);
	cout << *(deux->getAttributA_()) << endl;

	delete p;
	 return 0;
}
```