الجمهورية الجزائرية الديمقراطية الشعبية

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
Université 8 Mai 1945 Guelma
Faculté de mathématiques et de l'informatique et de sciences de la matière

وزارة التعليم العالي والبحث العلمي
جامعة 8 ماي 1945 قالمة
كلية الرياضيات ، الاعلام الآلي و علوم المادة

# Module : Programmation orientée objet II
# TP4 : BDD, Parallélisme et Objets distribués

## Partie I : Base de données orientées objets

### Exercice 1 :

Dans le contexte d'une entreprise, la gestion des pièces se divise en deux catégories : les pièces de base, acquises à l'externe, et les pièces composites résultant de l'assemblage d'autres pièces, qu'elles soient composites ou non. Pour modéliser cette gestion de pièces, nous souhaitons créer une base de données orientée objet décrivant les différents modèles de pièces. Chaque modèle de pièce composite sera caractérisé par son nom, son coût d'assemblage, ainsi que la liste et la quantité des pièces nécessaires à sa fabrication. Pour les modèles de pièces de base, nous enregistrerons leur nom et leur prix unitaire. Développer un gestionnaire de pièces *GestionairePiécesJPQL* qui vise à offrir la possibilité, à travers des requêtes JPQL de :

1. Calculer le prix de revient de toute pièce, qu'elle soit de base ou composite.
2. Afficher les noms des pièces utilisées dans la fabrication d'une pièce composite.

### Exercice 2 :

Dans une institution académique, l'efficacité de la planification des salles et des enseignements est cruciale. Proposer un système qui modélise robustement cette tâche en utilisant une base de données orientée objet. Le système doit répondre aux besoins spécifiques suivants :

1. Chaque salle est identifiable par un code unique, et peut être soit de type Cours (*SalleCours*) ou de type TP (*SalleTP*), avec des attributs spécifiques comme le nombre d'ordinateurs et la disponibilité d'un vidéoprojecteur.
2. Chaque enseignement (*Cours* ou *TP*) est caractérisé par un code unique et des détails tels que l'effectif d'étudiants dans le cas d'un *TP*.
3. Les affectations représentent l'assignation d'un enseignement à une salle pendant un créneau horaire donné, avec des informations sur le début et la fin de la séance.
4. Avant d'introduire une nouvelle affectation, le système effectue une vérification minutieuse des conflits avec les affectations existantes, dans la base, pour la même salle et le même créneau horaire. En cas de conflit, une boîte de dialogue s'affiche, forçant l'arrêt du programme.
5. Le système doit implémenter les requêtes fonctionnelles suivantes en JPQL :
    a. Lister toutes les salles de l'institution.
    b. Lister tous les enseignements.
    c. Afficher toutes les affectations effectuées.
    d. Afficher le nombre d'enseignements différents par salle.
    e. Lister les numéros de salles libres.
    f. Afficher les informations des salles informatiques : numéros de salle et nombre d'ordinateurs.
    g. Afficher des informations sur toutes les salles : numéros de salle et nombres d'ordinateurs.
    h. Liste des codes d'enseignements, besoins de rétroprojecteur et numéros de salles avec disponibilité de rétroprojecteur.

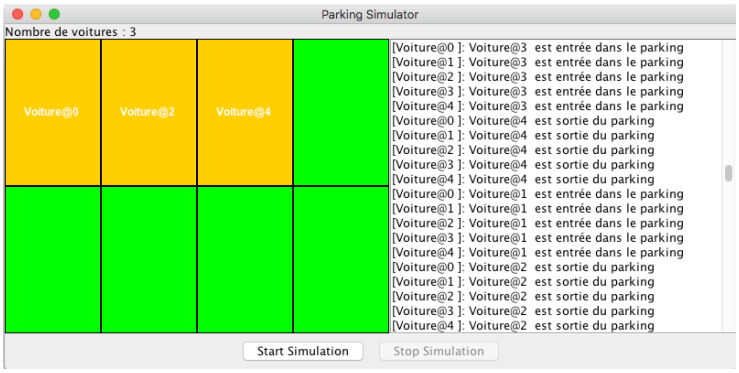
## Partie II : Parallélisme

### Exercice 3 :

Concevoir un système de réservation de sièges en utilisant le patron de conception Singleton pour la classe de base *SeatReservationSystem*, garantissant ainsi qu'une seule instance du système est disponible. Ce système permet à plusieurs utilisateurs de sélectionner et de réserver un nombre précis de sièges simultanément de manière aléatoire. Pour assurer une gestion robuste des cas d'utilisation concurrents, notamment lorsque plusieurs utilisateurs tentent de réserver le même siège simultanément, la synchronisation est mise en place. De plus, une fonctionnalité doit être intégrée pour afficher l'état actuel des sièges à chaque tentative réussie de réservation. En cas de non-disponibilité de sièges, les tentatives de réservation doivent être bloquées afin d'éviter une boucle infinie.

### Exercice 4 :

Concevoir un simulateur de parking doté d'une interface graphique conviviale. Ce simulateur doit efficacement gérer l'entrée et la sortie de voitures dans un parking ayant une capacité définie. Pour modéliser ces interactions, chaque voiture sera représentée par un thread distinct. Le système de parking devra garantir qu'il n'accepte pas plus de voitures que sa capacité initiale. Pour assurer une interaction dynamique et en temps réel, intégrez le patron de conception Observateur. Ainsi, les voitures pourront observer le parking et afficher leurs activités dans une console dédiée sur l'interface graphique.

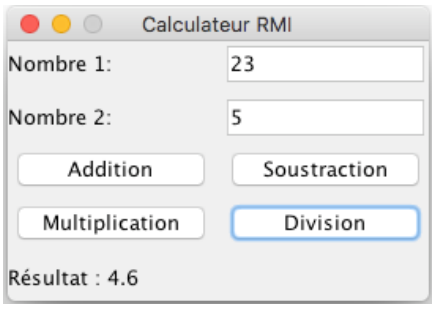



L'interface graphique doit offrir une expérience utilisateur intuitive avec deux boutons essentiels : un pour démarrer la simulation et un pour l'arrêter. Lorsque la simulation est en cours, les voitures doivent entrer et sortir du parking de manière aléatoire, créant ainsi un environnement dynamique et réaliste. Visualisez l'état actuel du parking de manière graphique en utilisant des couleurs distinctes pour indiquer les places occupées et libres. Chaque place de parking doit afficher le nom de la voiture qui l'occupe, améliorant ainsi la traçabilité des actions des véhicules.

## Partie III : Objets distribués

### Exercice 5 :

Concevoir un système de calcul distribué en utilisant la technologie RMI. L'exercice consistera à développer une interface graphique conviviale permettant à l'utilisateur d'effectuer des opérations mathématiques basiques à distance. Lorsque l'utilisateur déclenche une opération via un bouton, le client doit utiliser RMI pour invoquer la méthode correspondante sur un service de calcul distant. Le résultat de l'opération doit être affiché de manière claire et intuitive dans l'interface graphique du client, offrant ainsi une expérience interactive et transparente.



**Exercice 6 :**

Concevoir une application de messagerie instantanée mettant en œuvre un serveur de discussion robuste capable de gérer simultanément plusieurs clients. Cette application, développée avec la programmation Socket et le multithreading, permet à chaque client d'interagir avec le serveur en envoyant et recevant des messages. Chaque client est identifié par un nom unique, choisi lors du lancement de l'application cliente. L'interface graphique intuitive offre aux utilisateurs la possibilité de composer, envoyer et lire les messages échangés entre tous les clients connectés en temps réel, facilitant ainsi une communication fluide et conviviale.