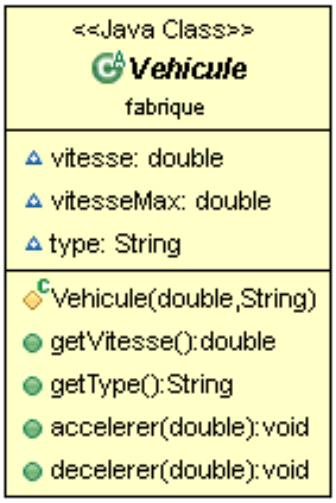
TD/TP5

PATRONS DE CONCEPTION – FABRIQUE - SINGLETON

1. SIMULATION INTERSECTION DE ROUTES – PATRON FABRIQUE

L'objectif de cet exercice est de simuler différents trafics en utilisant le patron Fabrique.



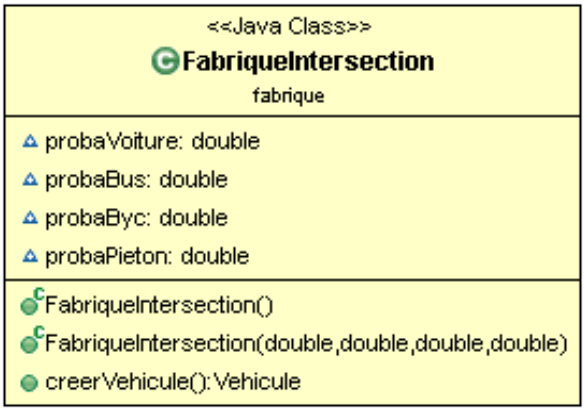
1.1. Ecrire une classe abstraite ***Vehicule*** conforme au diagramme ci-contre. L'attribut ***type*** désignera le type du **Vehicule** des sous-classes concrètes.

1.2. Écrire les classes suivantes qui dérivent de la classe ***Vehicule***. La classe **Voiture** avec une vitesse maximum de 120 km/h. La classe **Bus** avec vitesse maximum de 92 km/h. La classe **Bicyclette** avec une vitesse maximum de 25 km/h. La classe **Pieton** avec une vitesse maximum de 4 km/h.

1.3. Mettre en place le patron fabrique : écrire une interface ***FabriqueVehicule*** qui possède une méthode **creerVehicule** et écrire une fabrique concrète ***FabriqueVoiture*** qui produit des voitures.

1.4. Mettre en place une seconde fabrique ***FabriquePieton*** qui produit des piétons.

1.5. **Faire un diagramme de classe et le faire valider auprès de votre enseignant.**



1.6. Écrire une classe concrète ***FabriqueIntersection*** en utilisant le patron Fabrique. Cette fabrique retourne des instances des différentes sous-classes avec des fréquences différentes. Ces pourcentages sont soit passés dans le constructeur (avec la contrainte que la somme doit faire 100% et que les valeurs sont non négatives) soit prennent des valeurs par défaut (80% pour Voiture, 10% pour Pieton et 5% pour Bus et Bicyclette).

1.7. Écrire une classe ***Simulateur*** permettant de construire un simulateur à partir d'une fabrique passée en paramètre du constructeur. Ce simulateur peut-être très complexe : par exemple, il peut s'agir d'un jeu de type SimCity dans lequel les points d'arrivée des véhicules sont représentés par des fabriques. Pour l'exercice, le simulateur se chargera simplement d'effectuer des statistiques sur les véhicules produits. Écrire la méthode ***genererStats*** qui demande la création de 100 véhicules et retourne une **table associative** associant à chaque type le nombre de véhicules produits.

1.8. Créer une méthode ***ecrireStats*** qui se charge d'afficher ces résultats statistiques dans la console.

1.9. Écrire une classe **MainSimulation** qui affiche les statistiques pour chaque fabrique et vérifier visuellement que les résultats obtenus sont compatibles avec ceux attendus.

1.10. Créer une méthode ***dessinerStats*** dans la classe ***Simulateur*** qui affiche une fenêtre. Cette fenêtre contiendra un rectangle découpé en zones de plusieurs couleurs correspondant à chaque catégorie (voiture, bus, bicyclette ou piéton). La taille de chaque zone sera proportionnelle aux nombre de véhicules dans la catégorie associée à la zone.

1.11. (Optionnel) Écrire une classe ***FabriqueJonction*** simulant une jonction de route (deux routes se rejoignent). Cette classe possède deux fabriques en attribut (passés en paramètres au constructeur). Lorsqu'un objet ***FabriqueJonction*** doit créer un véhicule, il utilise les fabriques qu'il possède en attribut et alterne entre les deux à chaque création.

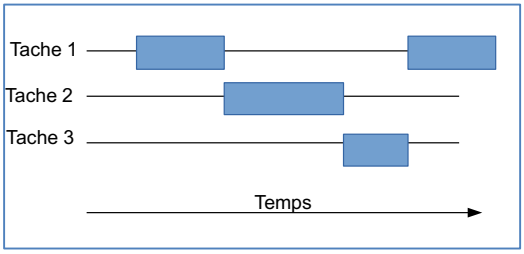
2. AEROPORT – PATRON SINGLETON

On veut simuler un aéroport avec plusieurs avions susceptibles de décoller.

Cet exercice utilise la classe Thread. Vous étudierez cette classe en détail en fin de S3. Voici un résumé des éléments à connaitre sur cette classe pour aborder l'exercice :

INTRODUCTION A LA CLASSE THREADS

En java, il est possible d'exécuter plusieurs traitements en parallèle dans une seule JVM grâce à des Thread. Les traitements ne s'exécutent pas réellement en même temps mais se répartissent dans le temps pour simuler le parallélisme. Cela donne par exemple le schéma ci-contre dans lequel chaque tâche est réellement exécutée par la JVM lorsque le rectangle bleu la recouvre. Cette répartition de la JVM est faite de manière transparente et peut fluctuer en fonction des exécutions.



Un **Thread** est un objet JAVA qui repose sur deux méthodes principales :

- une méthode **start()** qui permet de lancer le traitement,
- une méthode **run()** qui contient les traitements à effectuer. Cette méthode est habituellement surchargée pour modifier le traitement que le **Thread** doit faire (par défaut, la méthode run de la classe **Thread** ne fait rien).

La méthode statique **sleep(int duree)** de la classe **Thread** est aussi utile car elle permet de mettre en pause pendant **duree** ms le traitement qui appelle cette méthode. La méthode **sleep** peut lever une exception **InterruptedException** qu'il est nécessaire de catcher.

Pour construire un traitement, il suffit simplement d'hériter de la classe **Thread** en redéfinissant la méthode **run()**. Ainsi, si on voulait compter jusque 2 plusieurs fois en parallèle, on construirait d'abord la classe **CompterJusque2** suivante.

```
/**
* la classe CompterJusque2 modelise des traitements a executer en parrallele
* Dans ce cas, il s'agit de compter jusque 2
*/
public class CompterJusque2 extends Thread {

	int idTraitement;

	/**
```

```
        * constructeur
        */
        public CompterJusque2(int n) {
            this.idTraitement = n;
        }

        /**
        * redefinition de la methode run() de thread
        * le traitement a executer
        */
        public void run() {
            System.out.println("traitement " + idTraitement + "-> se lance");
            //compte de 1 a 2
            for (int i = 1; i < 3; i++) {
              System.out.println("traitement " + idTraitement + "-> " + i);
              //attends un centieme de seconde
              try {
                Thread.sleep(100);
              } catch (InterruptedException e) {
                e.printStackTrace();
              }
            }
            System.out.println("traitement " + idTraitement + "-> se finit");
        }
}
```

La classe **Principale** suivante permet de lancer 3 traitements en parallèle :

```
public class Principale {
   public static void main(String[] args) {
        //construction des traitements
        CompterJusque2 cmp=new CompterJusque2(1);
        CompterJusque2 cmp2=new CompterJusque2(2);
        CompterJusque2 cmp3=new CompterJusque2(3);
        //lancement des traitements
        cmp.start(); // le main continue
        cmp2.start(); // le main continue
        cmp3.start(); // le main continue jusqu'à la fin des traitements
        System.out.println("traitements encore en attente");
   }
}
```

L'affichage obtenu pourra avoir cette forme :

```
traitements encore en attente
traitement 3-> se lance
traitement 3-> 1
traitement 2-> se lance
traitement 1-> se lance
traitement 2-> 1
traitement 1-> 1
traitement 1-> 2
traitement 2-> 2
traitement 3-> 2
traitement 1-> se finit
traitement 2-> se finit
traitement 3-> se finit
```

Dans ce cas particulier, le main s'est arrêté avant que les traitements n'aient été lancés (ligne 1). Le premier traitement lancé est le 3 mais c'est néanmoins celui qui se terminera en dernier. Comme les temps d'attentes de 100ms sont assez longs, le reste des traitements se fait en parallèle dans le même ordre, mais sans temps d'attente, le résultat aurait été différent.
Il faut retenir que tels quels, la répartition des traitements dans la JVM peut être très aléatoire (et dépend de nombreux facteurs extérieurs liés entre autres à la machine utilisée). Plusieurs exécutions différentes peuvent ainsi donner des résultats différents. La question devient problématique quand des

traitements manipulent la même variable et il est alors nécessaire de mettre en place des processus de protection (proche des mécanismes de transactions vus en BDD).

2.1. Recopier les classes ci-dessous qui correspondent à un premier essai de modélisation d'un aéroport. Voici le code de la classe **Aeroport.**

```
public class Aeroport{
       private boolean piste_libre;
       public Aeroport(){
              this.piste_libre=true;
       }
}
```

Voici le code de la classe **Avion** :

```
class Avion extends Thread {
       private String nom;
       private Aeroport a;
       public Avion(String s){
              this.nom=s;
       }
       public void run(){
              this.a=new Aeroport();
              System.out.println("Je   suis   avion
"+this.nom+" sur aeroport "+this.a);
       }
}
```

Enfin, voici la classe permettant la création de plusieurs instances d'Avion. Exécuter cette classe.

```
   class MainAeroport{
    public static void main(String[] args){
     Avion v1 = new Avion("Avion 1");
     Avion v2 = new Avion("Avion 2");
     Avion v3 = new Avion("Avion 3");
     Avion v4 = new Avion("Avion 4");
     v1.start();
     v2.start();
     v3.start();
     v4.start();
    }
   }
```

2.2. Que fait la méthode **start** lorsqu'elle est appelée sur les avions ?

2.3. On souhaite que les objets de type **Avion**, ne puissent pas créer plus d'un **Aeroport**, afin qu'ils se situent tous dans un même **Aeroport**. Mettre en place la **version 1** du patron Singleton pour résoudre ce problème. Est-ce suffisant ? Proposer une solution.

2.4. Gestion des décollages : Créer dans la classe **Aeroport** deux méthodes :

- public synchronized boolean autoriserAdecoller() qui renvoie true si l'aéroport donne l'autorisation de décoller (la piste est libre)

- public synchronized boolean liberer_piste()qui renvoie true après que l'aéroport ait libèré la piste

Compléter ensuite la méthode run de la classe **Avion** pour que

- l'avion se mette en attente pour le décollage, attendant que la piste soit libre,
- le décollage prend ensuite un temps aléatoire,
- enfin l'avion libère la piste.