

Développement logiciel

Travaux pratiques

ENSMM

Ingénierie des systèmes
Semestre vert
INFO 2
2022



Ecole Nationale Supérieure de
Mécanique et des Microtechniques
26 chemin de l'épitaphe
25030 Besançon Cedex – FRANCE
<http://intranet-tice.ens2m.fr>

Rôles

Product Owner (P.O) :

Personne représentant le client et l'utilisateur auprès du ScrumMaster et de l'équipe de développement. Il définit le produit et priorise les fonctionnalités voulues.

Scrum Master :

C'est le *facilitateur*, le garant du processus agile. Il n'est PAS un chef de projet mais un leveur d'obstacles qui empêcheraient l'équipe de développement d'avancer. Il protège et guide l'équipe des interférences extérieures pendant le *sprint*.

(équipe de développement) :

Auto-gérée et multi-disciplinaire (développeurs, testeurs, architectes, etc), les membres travaillent idéalement dans une seule et même pièce. Elle livre un produit utilisable à la fin de chaque sprint.

Concepts

Story points (points d'*histoire*) :

Outil d'estimation de l'effort nécessaire pour développer des fonctionnalités. Les points d'*histoire* permettent de se soustraire du concept de jour/homme. Les points sont attribués à une user story relativement à d'autres user stories. Par exemple, une user story estimée à deux points demandera deux fois plus d'effort pour la terminer qu'une user story estimée à un point, ceci sans indication de la durée en jour.

Product backlog

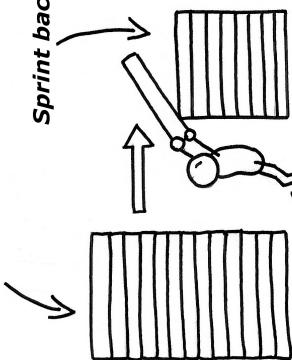
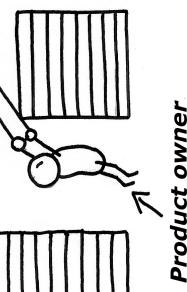


Schéma du cycle Scrum

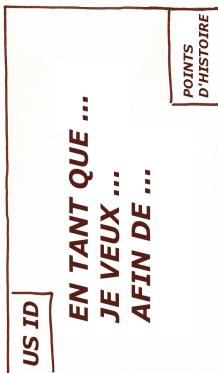


Velocity :

L'effort, exprimé en nombre de points d'*histoire*, que l'équipe de développement peut fournir dans un *sprint*. La valorisation en points des user stories permet de déterminer le panier de fonctionnalités absorbable par l'équipe de développement en un *sprint*.

User Story (U.S) :

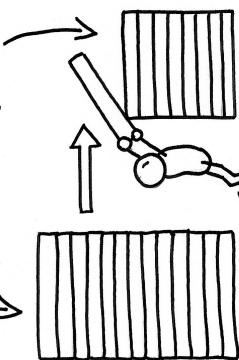
Description d'une fonctionnalité du point de vue utilisateur. Elle prend le formalisme "En tant que... Je veux... afin de...". Une user story peut être divisée en tâches si elle est complexe.
Ci-dessous un exemple de représentation d'une user story sur une carte.



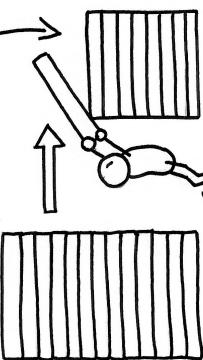
Definition of Done (Fini) :

La "définition de fini" est la liste de critères qu'une user story doit remplir pour être considérée comme ayant l'état "fini", donc livrable. Cette liste de critères peut inclure, par exemple, une couverture de test minimum, une revue de code d'un autre membre de l'équipe, une javadoc suffisante, etc. Il est important d'avoir une DoD déterminée de façon claire et conjointe entre l'équipe de développement et le Product Owner. Ce dernier exprime son acceptation d'une user story via des tests d'*acceptance*.

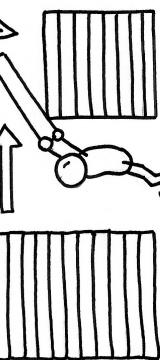
Sprint backlog



Daily meeting



Equipe



Sprint (2 à 4 semaines)

Réunion faite debout pour ne pas durer trop longtemps et à heure fixe (généralement le matin), lors de laquelle chaque participant répond aux trois questions : "Qu'ai-je fait hier?", "Que vais-je faire aujourd'hui?" et "Ai-je un point de blocage?"

Les estimations sont faites face cachée et dévoilées en même temps pour éviter d'influencer les autres membres de l'équipe :

0, 1, 2, 3, 5, 8, 13...

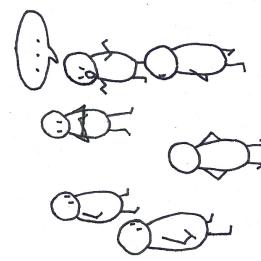
Les estimations sont faites face cachée et dévoilées en même temps pour éviter d'influencer les autres membres de l'équipe.

Rituels

Sprint : Période de 2 à 4 semaines dédiée au développement des user stories du *backlog*, et permettant d'avoir un produit potentiellement livrable à la fin de celle-ci.

Daily stand-up meeting :

Réunion faite debout pour ne pas durer trop longtemps et à heure fixe (généralement le matin), lors de laquelle chaque participant répond aux trois questions : "Qu'ai-je fait hier?", "Que vais-je faire aujourd'hui?" et "Ai-je un point de blocage?"



Sprint review (démonstration de fin de sprint) :

Réunion tenue en fin de sprint durant laquelle l'équipe de développement montre le travail accompli pendant le sprint (i.e. les fonctionnalités, les user stories demandées par le *Product owner*).

Planning poker :

Séance d'estimation menée par l'équipe de développement qui évaluent ensemble l'effort nécessaire pour traiter les user stories du *backlog*. Pour cela, ils utilisent chaque un jeu de carte sur lesquelles sont inscrit des nombres de points d'*histoires* dont les valeurs suivent généralement la suite de Fibonacci : 0, 1, 2, 3, 5, 8, 13... Les estimations sont faites face cachée et dévoilées en même temps pour éviter d'influencer les autres membres de l'équipe.

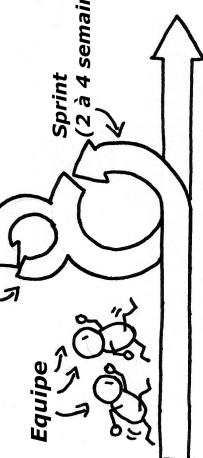
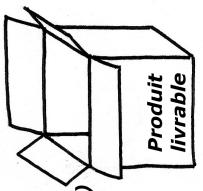


Diagramme du cycle Scrum : montre l'itération entre le Sprint backlog (en haut) et le Product backlog (en bas), avec l'équipe au centre effectuant des daily meetings.

Retrospective :

Réunion permettant à l'équipe de faire un bilan du sprint qui vient de se terminer. On y note ce qui fait avancer le projet et ce qui le ralentit. Dans ce dernier cas, l'équipe cherche des actions pour lever les obstacles.

Elle est généralement menée par le ScrumMaster et s'organise en 5 étapes :

- set the stage, prendre la température des participants via un vote (de confiance et/ou d'itération);
- gather data, liste le ressenti de l'équipe, les problèmes, les points positifs, les émotions qui l'ont marquée pendant le sprint qui vient de se terminer;
- generate insights, permet une réflexion de groupe sur la perception et les causes des obstacles évoqués précédemment;
- decide what to do, est l'étape qui permet de générer des actions à appliquer lors du sprint suivant pour tenter de lever les obstacles évoqués;
- close the retrospective, marque la fin de cette réunion. On y fait en général un vote nommé ROTI (Return On Time Invested) pour indiquer le degré de satisfaction sur le temps consacré à la rétrospective.

Artifacts

Product backlog :

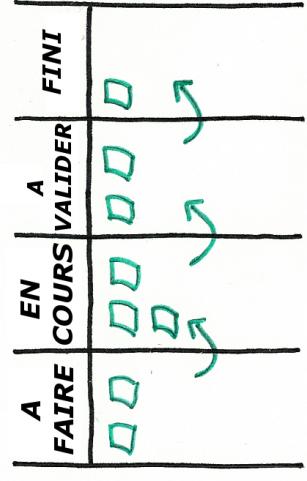
Ensemble des caractéristiques (fonctionnalités ou besoins techniques) qui constituent le produit souhaité. Il doit être priorisé pour permettre de développer les éléments de plus haute importance en premier.

Sprint backlog :

Sous-ensemble des éléments du backlog de produit. Les éléments constituent les user stories à développer au cours du sprint et sont préalablement détaillés pour pouvoir être estimés par l'équipe de développement. Il est également priorisé.

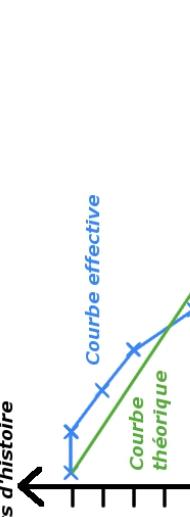
Task board (tableau des tâches) :

Tableau physique ou logiciel reprenant les éléments du *backlog de sprint*. Il possède plusieurs colonnes (ex. à faire, en cours, à valider, validée) permettant de suivre l'avancement des user stories affichées via des post-it ou des cartes.



Burn down chart :

Graphique permettant de suivre le "reste à faire" durant le sprint. Il possède en abscisse le temps et en ordonnée les points d'histoire. La courbe indique le nombre de points d'histoire abattus pendant le sprint. Elles sont mises à jour en continu. Cela permet d'anticiper les dérives et les ruptures de charge. L'idéal étant bien sûr d'arriver à zéro point le dernier jour du sprint.



Hing CHAN

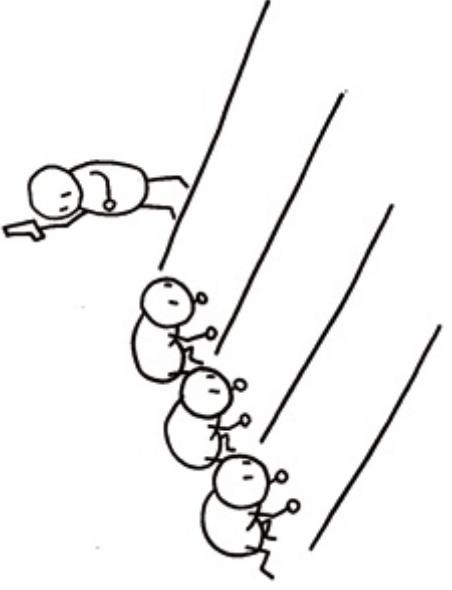
<http://hingchanscrum.blogspot.com>
@HingCChan

Thierry LERICHE

<http://icauda.com>
@thierryleriche

SCRUM

MEMENTO A DESTINATION DE L'EQUIPE



v1.1 / Nous mettons ce némento en téléchargement gratuit afin de le diffuser au maximum. N'hésitez pas à nous faire des retours pour que nous puissions l'améliorer.



Organisation du projet

Ce module aborde la gestion de projets informatiques à l'aide la méthode SCRUM. Il est organisé autour d'un projet commun réalisé en équipe. Chaque étudiant doit choisir un rôle dans l'équipe projet. Il est entendu que les étudiants ne développent pas les mêmes compétences en fonction de leur rôle, d'où un mode d'évaluation spécifique.

I Équipe projet

Chaque projet doit être réalisé par une équipe de 5 à 7 étudiants comprenant :

- 1 scrum master,
- 1 product owner,
- 3 à 5 développeurs.

II Attendus du projet

Le projet a pour objectif de développer un jeu en réseau avec un minimum 4 joueurs jouant simultanément et sera réalisé en Java et SQL sur un modèle client-serveur.

Les règles du jeu sont libres et sont proposées par chaque équipe. Néanmoins, les jeux doivent s'inscrire dans le thème du semestre notamment à travers leurs habillages graphiques et leurs "scénario".

De plus, chaque projet devra mobiliser les compétences suivantes :

- Gestion de projet avec SCRUM,
- Programmation orientée objet en Java (héritage, interface, polymorphisme),
- Conception et utilisation d'une base de donnée SQL,
- Animation graphique 2D (le déplacement des avatars et des PNJ en case par case est suffisant),
- Réalisation d'un décor avec la technique du tile mapping.

Enfin, il est primordiale qu'une version **jouable** (même minimaliste) soit présentée à l'issu du semestre.

III

Déroulement du projet

Quatre séances de quatre heures sont dédiées au projet ce qui permet de réaliser un sprint de développement. Il est essentiel de bien s'organiser et d'anticiper pour éviter une surcharge importante de travail en fin de semestre.

1

Avant la première séance de projet

Avant la première séance de projet, vous devez :

- Former une équipe autour d'un concept de jeu,
- Réaliser le story mapping (sur Trello par exemple) sous la supervision du product owner.

2

Première séance de projet

La première séance de projet commence par la validation du story mapping par votre enseignant suivie d'un sprint planning meeting et du début du sprint :

- Présentation du story mapping par le product owner,
- Transformation de la story map en product backlog par le scrum master (stories rangées par priorité dans une colonne "à faire", n'oubliez pas de garder une trace de votre story map pour le rapport final)
- Sprint planning meeting animé par le scrum master (répartition des rôles et des stories),
- Conception et création de la base de données,
- Début du codage, réalisation par chacun des membres d'un petit programme interagissant avec la BDD et permettant de commencer à réaliser une US (par exemple, créer un joueur, changer les coordonnées d'un joueur, créer un monstre/piège, détecter si un joueur touche un monstre/-piège et lui enlever des points de vie, etc.). Attention, à ce stade, il ne s'agit pas de programmes graphiques, les informations doivent seulement apparaître dans la console.

3

Autres séances de projet

Les séances suivantes commencent par un stand-up meeting pour faire le point sur l'avancement de chacun et du projet en général :

- Stand-up meeting animé par le scrum master,
- Codage des stories,
- Validation des stories au fil de l'eau par le product owner et mise à jour du burndown chart.

IV

Évaluation du projet

L'évaluation du projet se déroule à l'orale sous la forme d'un sprint review de 30 minutes par équipe. Elle se compose pour moitié d'une présentation du projet et des rôles de chacun suivie d'une démonstration du projet. La présentation pourra suivre le plan suivant :

- Introduction du product owner (concept, règles du jeu, story mapping, 3 diapos)
- Présentation de l'architecture logicielle par le scrum master (diagramme de classes, base de données, 3 diapos)

4

- Présentation de quelques stories par les développeurs (2 diapos/personne)
- Bilan du sprint (stories terminées/en cours/à faire, burndown chart, retrospective du sprint, 2 diapos)

L'évaluation du projet tient également compte des résultats obtenus (étendue des fonctionnalités, absence de bug) et de la qualité du code (lisibilité, architecture, utilisation des techniques mentionnées dans les attendus).

Le projet doit être rendu par le scrum master sous forme d'une archive compressée **zip** contenant tous les fichiers du projet NetBeans (répertoire complet du projet NetBeans à l'exception du dossier build).

Une vidéo de démonstration du jeu doit également être déposée par le product owner ou le scrum master.



Évaluation individuelle

Une note individuelle est également attribuée à chaque étudiant. Elle est fondée sur une auto-évaluation et un rapport présentant le travail fourni.

Le rapport est individuel et doit décrire le rôle et les tâches réalisées dans l'équipe en une dizaine de pages. Son contenu sera différent en fonction du rôle de chacun.

Contenu du rapport pour un product owner :

- [1] Présentation du jeu (concept, règles du jeu, etc.)
- [2] Présentation de la story map
- [3] Liste des stories réalisées
- [4] Bilan (stories terminées/en cours/à faire)

Contenu du rapport pour un scrum master :

- [1] Présentation de l'architecture (diagramme de classes, base de données, etc.)
- [2] Liste des stories réalisées
- [3] Synthèse du déroulement du sprint (burndown chart)
- [4] Rétrospective (ce qui a marché ou pas)

Contenu du rapport pour un développeur :

- [1] Présentation détaillée d'une story réalisée (diagramme d'activités, maquette, tests d'acceptation, etc.)
- [2] Explication de sa réalisation (diagramme des classes réalisées, exemples d'utilisation, etc.)
- [3] Liste des autres stories réalisées

Java DataBase Connectivity (JDBC)

Dans ce tutoriel, vous allez apprendre à utiliser JDBC (Java DataBase Connectivity). JDBC est une bibliothèque Java qui permet d'accéder à des bases de données par le biais d'une interface commune.

Nous allons utiliser une base de données MySQL hébergée sur un serveur de l'ENSMM dont le nom DNS est nemrod.ens2m.fr. Cette base porte le nom tp_jdbc. Son accès est possible à l'aide de l'identifiant etudiant et du mot de passe YTDTvj9TR3CDYCmP.

Cette base contient deux tables, l'une contenant des dresseurs (joueurs), l'autre des pokémons. Le contenu de ces tables est librement inspiré du jeu PokémonGo.

La table des dresseurs enregistre les joueurs avec leurs identifiants (pseudo, adresse email et empreintes MD5 du mot de passe), leurs latitude et longitude (en degrés décimaux) et la date de leur dernière connexion (date et heure).

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
1	pseudo 🔑	varchar(32)	utf8_general_ci		Non	Aucun(e)		
2	email	varchar(64)	utf8_general_ci		Non	Aucun(e)		
3	motDePasse	varchar(32)	utf8_general_ci		Non	Aucun(e)		
4	latitude	double			Non	Aucun(e)		
5	longitude	double			Non	Aucun(e)		
6	derniereConnexion	datetime			Non	Aucun(e)		

La table des pokémons liste les pokémons avec leur numéro (ID), leur espèce, leurs latitude et longitude, leur visibilité, la date de leur ajout dans la table et le pseudo de leur propriétaire éventuel (pseudo d'un dresseur ou sauvage si non attribué).

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
1	id 🔑	int(11)			Non	Aucun(e)		AUTO_INCREMENT
2	espece	varchar(32)	utf8_general_ci		Non	Aucun(e)		
3	latitude	double			Non	Aucun(e)		
4	longitude	double			Non	Aucun(e)		
5	visible	tinyint(1)			Non	Aucun(e)		
6	creation	datetime			Non	Aucun(e)		
7	proprietaire	varchar(32)	utf8_general_ci		Non	Aucun(e)		

L'accès à une base de données avec JDBC se fait en quatre étapes :

- Connexion à la base de données
- Préparation d'une requête
- Envoie de la requête
- Récupération et exploitation du résultat



Se connecter à une base de données

La première étape consiste à se connecter à la base de données. JDBC fournit une classe bien pratique pour cela, la classe `DriverManager`. Pour établir la connexion, il suffit de construire un objet `DriverManager` en précisant l'adresse complète de la base, un identifiant et un mot de passe.

Dans notre cas, l'adresse complète est constituée du préfixe `jdbc:mysql://`, suivi du nom DNS du serveur `nemrod.ens2m.fr`, puis du numéro du port :`3306`, et enfin du nom de la base `/tp_jdbc`, ce qui donne :

```
Connection connexion = DriverManager.getConnection("jdbc:mysql://nemrod.ens2m.fr:3306/tp_jdbc?serverTimezone=UTC", "etudiant", "YTDTvbj9TR3CDYCMp");
```

Une fois la connexion établie, vous pouvez lancer une ou plusieurs requêtes. Il faut ensuite fermer la connexion avec :

```
connexion.close();
```



Consulter une table – SELECT

Nous allons maintenant voir comment réaliser une requête de type `SELECT`. Il faut d'abord préparer la requête à l'aide d'une connexion précédemment ouverte. Par exemple, si vous voulez récupérer la liste des pseudos des dresseurs avec leurs coordonnées, vous pouvez écrire la requête suivante :

```
PreparedStatement requete = connexion.prepareStatement("SELECT _pseudo, _latitude, _longitude FROM _dresseurs");
```

Il suffit ensuite d'exécuter la requête :

```
ResultSet resultat = requete.executeQuery();
```

Si la requête ne comporte pas d'erreur, on récupère le résultat dans un objet `ResultSet`. La méthode `next` permet d'accéder aux lignes successives. Cette méthode renvoie `false` si il n'y a plus de ligne à traiter. On récupère ensuite les valeurs des différents champs à l'aide des méthodes `getString`, `getDouble`, `getBoolean` ou encore `getDate` en précisant le nom du champ désiré.

```
while (resultat.next()) {  
    String pseudo = resultat.getString("pseudo");  
    double latitude = resultat.getDouble("latitude");  
    double longitude = resultat.getDouble("longitude");  
    System.out.println(pseudo + " _(_" + latitude + ";_ " + longitude + ")");  
}
```

Une fois les résultats traités, vous devez fermer la requête.

```
requete.close();
```

- 1** Ouvrez le projet Netbeans TP_JDBC et exécutez le fichier TestSelect.java.
- 2** Dupliquez TestSelect (coper/coller dans l'arborescente du projet) puis modifiez la copie pour lister les pokémons de la table des pokémons avec leurs ID, espèce, latitude et longitude. Le nom de la table des pokémons est pokemons.

La commande WHERE dans une requête SQL permet d'extraire les lignes d'une base de données qui respectent une condition. Par exemple, pour lister les dresseurs dont la longitude est supérieure à 5,993°, on peut écrire :

```
PreparedStatement requete = connexion.prepareStatement ("SELECT_pseudo,_latitude,_longitude_FROM_dresseurs_
WHERE_longitude_>_5.993");
```

- 3** Modifiez le programme précédent pour récupérer la liste des pokémons attribués à un dresseur connu par son pseudo.

Remarque

Pour mettre au point, une requête SELECT sans avoir à personnaliser le code de traitement du résultat, vous pouvez utiliser la méthode maison OutilsJDBC.afficherResultSet(resultat) qui affiche tous les résultats dans la console quels qu'ils soient.

- 4** Dupliquez TestSelectAll et écrivez un programme listant les pokémons de la table des pokémons
- 5** Modifiez le programme pour lister uniquement les pokémons dont la latitude et la longitude est à moins de 0.001° du point (47.250°, 5.992°).
- 6** Écrivez un programme permettant de compter le nombre de pokémons visibles à l'aide de la fonction COUNT.

III Modifier une donnée – UPDATE

La préparation d'une requête UPDATE est similaire à celle d'une requête SELECT. Par exemple, la requête suivante vous permet de modifier les coordonnées d'un joueur dans la table des dresseurs.

```
PreparedStatement requete = connexion.prepareStatement ("UPDATE_dresseurs_SET_latitude=_?,_longitude=_?
WHERE_pseudo=_?");
```

On remarque ici la présence de points d'interrogation qui permettent de préciser les valeurs dans un second temps sans changer la requête.

```
requete.setDouble(1, 47.251);
requete.setDouble(2, 5.994);
requete.setString(3, "jessie");
```

Remarque

Les numéros désignent les points d'interrogation dans l'ordre d'apparition dans la requête. Attention, contrairement aux indices de Java, les indices de SQL commencent à 1 !

Une fois les données explicitées, vous pouvez exécuter la requête mais comme cette requête va modifier la base, il faut utiliser la méthode executeUpdate et non executeQuery.

```
requete.executeUpdate();
```

- 1** Ouvrez et testez `TestUpdate.java`
- 2** Écrivez un programme similaire permettant de déplacer un pokémon.
- 3** Vérifiez le fonctionnement de votre programme à l'aide du programme `TestSelectAll`.
- 4** Écrivez un programme similaire permettant de modifier le propriétaire d'un pokémon dans la table des pokémons.

IV Ajouter des données – INSERT

L'utilisation d'une requête INSERT suit la même logique que les exemples précédents. L'exemple suivant vous montre comment préparer et exécuter une requête permettant d'ajouter un joueur dans la table des dresseurs.

```
PreparedStatement requete = connexion.prepareStatement("INSERT INTO dresseurs VALUES (?,?,?,?,?,?)");
requete.setString(1, "pierre");
requete.setString(2, "pierre@ens2m.org");
requete.setString(3, OutilsJDBC.MD5("1234"));
requete.setDouble(4, 47.250221);
requete.setDouble(5, 5.992052);
requete.executeUpdate();
```

La méthode maison `OutilsJDBC.MD5("1234")` permet de calculer l'empreinte du mot de passe 1234 en MD5 dans la requête pour ne pas le transmettre et le stocker en clair.

La fonction `NOW()` est une commande SQL qui renvoie la date et l'heure du serveur.

Remarque

Dans la table des dresseurs, il n'est pas possible d'ajouter un joueur portant un pseudo déjà présent car le pseudo est la clé primaire de la table des dresseurs.

- 1** Ouvrez `TestInsert.java`, changez les données puis testez.
- 2** Écrivez un programme similaire permettant d'ajouter un pokémon sauvage dans la table des pokémons.

V Supprimer des données – DELETE

Pour supprimer des données avec une requête DELETE, la procédure est identique. Par exemple, pour supprimer un joueur connu par son nom, vous pouvez écrire :

```
PreparedStatement requete = connexion.prepareStatement("DELETE FROM dresseurs WHERE pseudo=?");
requete.setString(1, "pierre");
requete.executeUpdate();
```

Attention !

Un requête DELETE mal faite peut vider tout ou partie de la table ! Testez votre requête préalablement en remplaçant DELETE par SELECT *.

- 1** Ouvrez et testez `TestDelete.java`
- 2** Écrivez un programme similaire permettant de supprimer un pokémon connu par son id dans la table des pokémons.
- 3** Écrivez un programme permettant de supprimer les pokémons en dehors de la carte (les longitudes et latitudes maximales et minimales sont disponibles dans le fichier `Carte.java` du package `pokemon`

VI

Lier deux tables – JOIN

Il est possible de lier deux tables en effectuant un produit cartésien. Le résultat de l'opération est une nouvelle table.

```
PreparedStatement requete = connexion.prepareStatement ("SELECT * FROM dresseurs, pokemons");
```

- 1** *Dupliquez TestSelectAll et ajoutez la table des pokemons en plus de la table des dresseurs comme ci-dessus. Qu'observez vous ?*

Pour éviter de générer inutilement des tables gigantesques, il est possible d'ajouter une condition pour lier plusieurs tables entre elles, c'est une jointure. La plus courante est la jointure interne qui retourne les enregistrements quand la condition est vrai dans les deux tables.

```
PreparedStatement requete = connexion.prepareStatement ("SELECT * FROM dresseurs INNER JOIN pokemons ON  
dresseurs.pseudo=pokemons.proprietaire");
```

- 2** *Écrivez un programme permettant de récupérer à l'aide d'une jointure la liste des couples pseudo/espece*
- 3** *Modifiez le programme pour récupérer le nombre de pokémons possédés par chaque dresseur à l'aide des commandes COUNT et GROUP BY.*
- 4** *Écrivez un programme permettant de lister les couples pokémons/dresseurs qui ont une distance relative inférieure à 0.001° en latitude et en longitude rangés par ordre alphabétique des pseudos à l'aide de la commande ORDER BY.*

VII

Récupération des clés générées

Lors de l'insertion d'un pokémon dans la table des pokémon, son ID est généré automatiquement par la base de données. JDBC donne un moyen d'obtenir la valeur de la clé générée aussitôt la requête d'insertion effectuée.

```
PreparedStatement requete = connexion.prepareStatement ("INSERT INTO pokemons VALUES (0,?, ?, ?, ?, ?, NOW(), ?)", Statement.RETURN_GENERATED_KEYS);  
...  
requete.executeUpdate();  
  
ResultSet resultat = requete.getGeneratedKeys();  
if (resultat.next()) {  
    System.out.println(resultat.getInt(1));  
}
```

Héritage

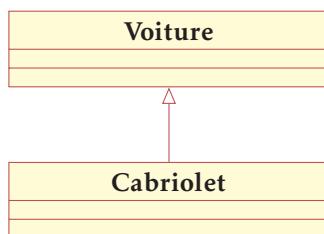
Compétences

- Etre capable de définir une classe dérivée d'une classe existante.

L'héritage est un concept fondamental de la programmation orientée objet qui permet la réutilisation d'une classe existante et de mettre en place le polymorphisme.

L'héritage offre la possibilité de définir une classe sur la base d'une classe déjà existante. La nouvelle classe, appelée *classe dérivée*, possède les mêmes attributs et les mêmes méthodes que la classe de base et peut être étendue en ajoutant de nouveaux attributs et de nouvelles méthodes ou en redéfinissant des méthodes de la classe de base. La redéfinition d'une méthode est appelée *surcharge*.

La relation entre la classe dérivée et la classe de base peut être exprimée par le verbe être au sens où, par exemple, un cabriolet *est une* voiture avec un toit escamotable. Ainsi dans une simulation automobile, on pourrait définir une classe cabriolet sur la base d'une classe voiture à laquelle on ajouterait un toit escamotable. Cette relation est représentée par le diagramme UML suivant :



Soit une classe Point définie de la manière suivante :

```
public class Point {  
  
    private double x;  
    private double y;  
  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public void setAbscisse(double x) {  
        this.x = x;  
    }  
  
    public double distance(Point p) {  
        return Math.sqrt((this.x - p.x) * (this.x - p.x)  
                        + (this.y - p.y) * (this.y - p.y));  
    }  
  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
}
```

On souhaite définir un point coloré qui *est un* point auquel on ajoute la notion de couleur. Pour cela, on utilise le mot-clé `extends` dans la définition de la classe.

```
public class PointColore extends Point {  
  
    private String couleur;  
  
    public PointColore(double x, double y, String couleur) {  
        super(x, y);  
        this.couleur = couleur;  
    }  
}
```

En général, le constructeur de la classe dérivée utilise le constructeur de la classe de base. Dans ce cas, une syntaxe particulière est employée : dans la définition du constructeur de la classe dérivée, la première instruction est obligatoirement `super(...)` avec les arguments destinés au constructeur de la classe de base .

Avec cette définition, la classe `PointColore` dispose des trois attributs (`abscisse`, `ordonnée`, `couleur`) et des méthodes de la classe `Point` (`setAbscisse`, `distance` et `afficher`). On peut écrire par exemple :

```
PointColore a = new PointColore(3, 7, "Bleu");  
PointColore b = new PointColore(10, 5, "Rouge");  
  
a.setAbscisse(0);  
System.out.println("Point_a:" + a);  
System.out.println("Point_b:" + b);  
System.out.println("Distance_entre_a_et_b=" + a.distance(b));
```

Qui donnera le résultat suivant dans la console :

```
Point a : (0.0, 7.0)  
Point b : (10.0, 5.0)  
Distance entre a et b = 10.198039027185569
```

On constate que les méthodes `setAbscisse` et `distance` conviennent parfaitement à la nouvelle classe mais que la méthode `toString` doit être modifiée. On peut également vouloir définir de nouvelles méthodes qui n'existent pas dans la classe `Point` comme par exemple `estDeLaMemeCouleur`. Pour cela, on ajoute tout simplement les méthodes à la classe `PointColore`. Le mot réservé `super` permet d'appeler une méthode de la classe de base afin d'éviter de récrire le code.

```
public class PointColore extends Point {  
  
    private String couleur;  
  
    public PointColore(double x, double y, String couleur) {  
        super(x, y);  
        this.couleur = couleur;  
    }  
  
    public boolean estDeLaMemeCouleur(PointColore p) {  
        return (this.couleur.equals(p.couleur));  
    }  
  
    public String toString() {  
        return super.toString() + " " + this.couleur;  
    }  
}
```

L'exécution du code précédent donnera cette fois le résultat :

```
Point a : (0.0, 7.0) Bleu  
Point b : (10.0, 5.0) Rouge  
Distance entre a et b = 10.198039027185569
```

Travaux pratiques

La préparation des exercices est disponible dans le projet NetBeans TP_Heritage.

Exercice 1

Soit la classe Individu définie ci-dessous et dans le projet.

```
public class Individu {  
  
    private String nom;  
    protected String prenom;  
    private int age;  
  
    /** Constructeur a partir de donnees elementaires */  
    public Individu(String nom, String prenom, int age) { ... }  
  
    /** Constructeur par defaut */  
    public Individu() { ... }  
  
    /** Accesseur de nom */  
    public String getNom() { ... }  
  
    /** Mutateur de nom */  
    public void setNom(String nom) { ... }  
  
    /** Accesseur de prenom */  
    public String getPrenom() { ... }  
  
    /** Mutateur de prenom */  
    public void setPrenom(String prenom) { ... }  
  
    /** Accesseur d'age */  
    public int getAge() { ... }  
  
    /** Mutateur d'age */  
    public void setAge(int age) { ... }  
  
    /** Affichage complet dans la console */  
    public void afficher() { ... }  
  
    /** Conversion en chaine pour un affichage compact */  
    public String toString() { ... }  
  
    /** Test d'egalite entre individus */  
    public boolean equals(Object o) { ... }  
}
```

- 1** Créer une classe TestIndividu contenant une méthode main (à l'aide de la commande File->New File->Java Main Class) puis écrire un programme de test construisant un individu et affichant ses caractéristiques dans la console.

On souhaite définir une classe Etudiant. Tout étudiant est un individu avec comme informations supplémentaires un numéro d'étudiant, une adresse email et le nombre d'inscriptions déjà effectuées dans l'établissement. Les méthodes de la classe Etudiant doivent être au moins les mêmes que celles de la classe Individu (constructeur, afficher, acquérir, age, etc.) mais on souhaite en plus disposer de méthodes permettant de connaître le numéro et l'email de l'étudiant. L'email de l'étudiant est généré automatiquement à partir de son nom et de son prénom.

- 2** Décrire la relation entre les classes Individu et Etudiant à l'aide d'un diagramme UML.

- 3** Définir la classe Etudiant.

On souhaite ensuite définir une classe annuaire répertoriant des individus. Un annuaire est une liste d'individus avec des fonctionnalités spécifiques.

- 4** Décrire la relation entre les classes `Annuaire` et `ArrayList<Individu>` à l'aide d'un diagramme UML.
- 5** Définir une classe `Annuaire` permettant d'afficher tous les individus, d'ajouter un nouvel individu, de connaître le nombre total de personnes, de savoir si un individu est dans l'annuaire et de calculer l'âge moyen des personnes.
- 6** Définir une classe `TestAnnuaire` contenant une méthode `main` pour tester la classe `Annuaire`. Essayer d'ajouter également des étudiants, qu'observe-t-on ?

Exercice 2

On s'intéresse aux polynômes de degré deux à coefficients complexes. Une classe `PolynomeNormalise` est fournie. Cette classe utilise la classe `Complexe` et implante les polynômes de la forme :

$$P(z) = z^2 - sz + p \quad (3.1)$$

s et p étant respectivement la somme et le produit des racines.

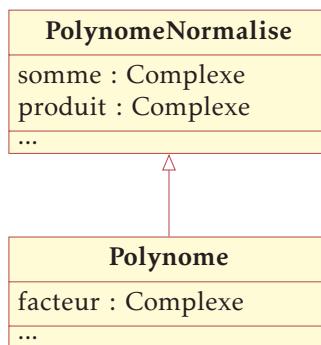
La classe `PolynomeNormalise` comporte quelques méthodes : constructeur, afficher, valeur, racines, égalité...

- 1** Tester la classe `PolynomeNormalise` à l'aide de la classe `TestPolynome`.

On s'intéresse maintenant aux polynômes de la forme :

$$P(z) = r(z^2 - sz + p) \quad (3.2)$$

avec r un complexe. En d'autres termes, un polynôme est un polynôme normalisé multiplié par un facteur r .



- 2** Définir une classe `Polynome` dérivant de `PolynomeNormalise`. Avec un polynome, on devra pouvoir utiliser les méthodes : constructeur, afficher, valeur, racines, égalité.

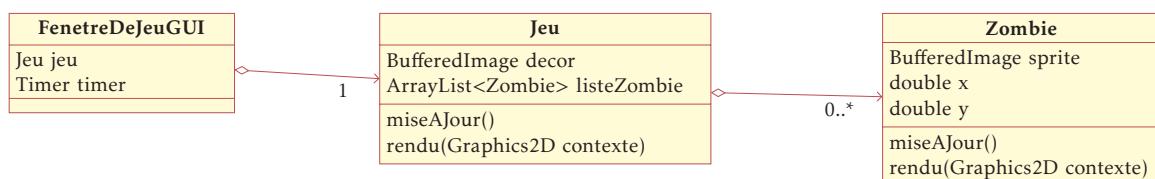
Exercice 3

Dans cet exercice, nous allons illustrer l'utilisation de l'héritage dans de cadre d'une petite animation inspirée du jeu *Plants vs Zombies*.



Le package `PlantsVsZombies` contient 4 classes :

- La classe `FenetreDeJeuGUI` permet d'ouvrir la fenêtre pour afficher l'écran du jeu. Elle contient un timer qui appelle tous les 40 ms la mise à jour puis le rendu (affichage) du jeu.
- La classe `Jeu` contient l'image du décor (jardin) et la liste des zombies qui se déplacent dans le jeu. La méthode `miseAJour` demande successivement à chaque zombie de calculer sa nouvelle position. De même, la méthode `rendu` commence par dessiner de décor puis demande à chaque zombie de faire son rendu.
- La classe `Zombie` contient entre autre une image, les coordonnées `x` et `y` du zombie dans le décor et les méthodes `miseAJour` et `rendu`.



1 Tester l'animation en exécutant la classe `FenetreDeJeuGUI`.

On souhaite ajouter des zombies avec des caractéristiques et des comportements différents comme par exemple un `PogoZombie` qui est un zombie se déplaçant sur un bâton sauteur.

2 Décrire la relation entre les classes `PogoZombie` et `Zombie` en complétant le diagramme UML.

3 Définir une classe `PogoZombie` qui se déplace par petits bonds (on pourra utiliser la valeur absolue d'un cosinus pour simplifier les calculs).

4 Compléter le code associé au bouton pour ajouter un `PogoZombie` au jeu de la même manière qu'un zombie.

Interfaces

Compétences

- Etre capable de définir une classe implémentant une interface standard (comme Comparable ou ActionListener).

Java propose un mécanisme similaire à l'héritage permettant de définir un modèle de classe : les interfaces. Une interface permet d'imposer un comportement à des classes en définissant les méthodes qui doivent impérativement être implémentées.

Une interface est équivalente à une classe abstraite qui ne contient aucun attribut et uniquement le prototype des méthodes. Toute classe qui implémente l'interface doit définir à minima l'ensemble des méthodes de l'interface.

Par exemple, on peut définir une interface `Affichable` imposant à toutes classes qui l'implémentent de définir la méthode `afficher`.

```
public interface Affichable {  
    public void afficher();  
}
```

Pour définir une classe qui implémente une interface, on utilise le mot réservé `implements`. Avec l'exemple du point, on peut écrire :

```
public class Point implements Affichable {  
    ...  
    public void afficher() {  
        System.out.println("Abscisse = " + this.x);  
        System.out.println("Ordonnée = " + this.y);  
    }  
    ...  
}
```

Remarque

Contrairement à l'héritage, une classe peut implémenter plusieurs interfaces.

En Java, la plupart des classes génériques imposent d'implémenter une interface. C'est par exemple le cas de la classe `Collections` qui contient des algorithmes de tris et de recherche d'extremum. Pour pouvoir utiliser la méthode `sort`, les objets doivent implémenter la méthode `compareTo`.

Par exemple, si on veut classer une liste de points de gauche à droite, on peut écrire :

```
ArrayList<Point> liste = new ArrayList<Point>();  
liste.add(new Point(8, 9));  
liste.add(new Point(19, 7));  
liste.add(new Point(1, 11));  
Collections.sort(liste);  
System.out.println(liste);
```

à condition que la classe Point implémente l'interface Comparable :

```
public class Point implements Comparable {  
    ...  
    public int compareTo(Object obj) {  
        Point autre = (Point) obj;  
        if (this.x < autre.x) {  
            return -1;  
        } else if (this.x > autre.x) {  
            return 1;  
        } else {  
            return 0;  
        }  
    }  
    ...  
}
```

Travaux pratiques

La préparation des exercices est disponible dans le projet NetBeans TP_Interfaces.

Exercice 1

Soit la classe Individu définie ci-dessous et dans le projet NetBeans.

```
public class Individu {  
  
    private String nom;  
    protected String prenom;  
    private int age;  
  
    /** Constructeur à partir de données élémentaires */  
    public Individu(String nom, String prenom, int age) { ... }  
  
    /** Constructeur par défaut */  
    public Individu() { ... }  
  
    /** Accesseur de nom */  
    public String getNom() { ... }  
  
    /** Mutateur de nom */  
    public void setNom(String nom) { ... }  
  
    /** Accesseur de prénom */  
    public String getPrenom() { ... }  
  
    /** Mutateur de prénom */  
    public void setPrenom(String prenom) { ... }  
  
    /** Accesseur d'âge */  
    public int getAge() { ... }  
  
    /** Mutateur d'âge */  
    public void setAge(int age) { ... }  
  
    /** Affichage complet dans la console */  
    public void afficher() { ... }  
  
    /** Conversion en chaîne pour un affichage compact */  
    public String toString() { ... }  
  
    /** Test d'égalité entre individus */  
    public boolean equals(Object o) { ... }  
}
```

- 1** Ajouter une méthode compareTo respectant l'interface Comparable à la classe Individu permettant de comparer un individu avec un autre par ordre alphabétique des noms.

On a également défini une classe annuaire comme une liste d'individus (voir la leçon sur l'héritage).

```
public class Annuaire extends ArrayList<Individu> {  
  
    public Annuaire() {  
        super();  
    }  
}
```

On souhaite pouvoir trier les individus d'un annuaire par ordre alphabétique des noms.

- 2** Ajouter une méthode à la classe Annuaire permettant de trier les individus par ordre alphabétique.

Exercice 2

La classe Timer permet d'appeler une méthode à intervalles réguliers :

```
Tache tache = new Tache();
Timer timer = new Timer(1000, tache); // appel de la tache tous les 1000 ms
timer.start();
```

Pour que cela fonctionne, l'objet tache doit implémenter la méthode actionPerformed de l'interface ActionListener.

```
public interface ActionListener {
    public void actionPerformed(ActionEvent e);
}
```

1 Définir une classe Tache permettant d'afficher "Hello world!" toutes les secondes dans la console.

Exercice 3

Pour qu'une application soit pilotable avec le clavier (comme un jeu par exemple), il faut qu'elle reçoive et traite les évènements arrivant à sa fenêtre active (qui a le focus). Pour cela, on doit créer un écouteur clavier (*listener*) qui est un objet destiné à recevoir et traiter les évènements provoqués par l'appui et le relâchement des touches du clavier.

La classe TestEcouteurs contient un programme qui crée une fenêtre graphique (JFrame) et lui ajoute un écouteur pour intercepter les évènements provenant du clavier.

```
public class TestEcouteurs {
    public static void main(String[] args) {
        JFrame fenetre = new JFrame();
        fenetre.setSize(500, 400);
        fenetre.setVisible(true);
        fenetre.addKeyListener(new EcouteurClavier());
    }
}
```

Pour que cet ajout soit possible, la classe EcouteurClavier doit implémenter l'interface KeyListener qui contient trois méthodes.

```
public interface KeyListener {
    public void keyPressed(KeyEvent event);
    public void keyTyped(KeyEvent event);
    public void keyReleased(KeyEvent event);
}
```

Les trois méthodes (keyPressed, keyTyped et keyReleased) sont appelées lorsqu'un KeyEvent arrive.

On peut connaître la touche qui a été appuyée ou relâchée à l'aide de la méthode event.getKeyCode() ainsi que le caractère correspondant avec la méthode event.getKeyChar().

1 Définir une classe EcouteurClavier permettant d'afficher le code de chaque touche appuyée.

Compétences

- être capable de dessiner des formes simples (lignes, rectangles, ellipses) ou d'écrire du texte dans une fenêtre graphique.
- connaître la structure d'une image matricielle et être capable de la dessiner dans une fenêtre graphique

L'API Java 2D offre des fonctionnalités graphiques, textuelles et d'imagerie pour les programmes Java. Elle permet d'effectuer facilement les tâches suivantes :

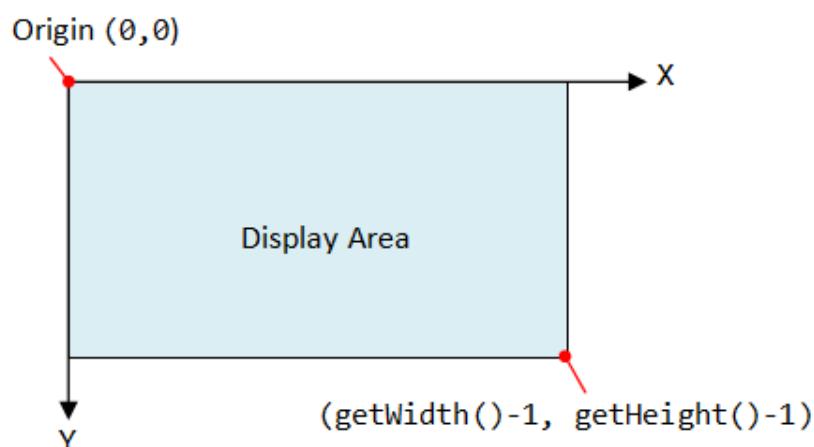
- Dessiner des lignes, des rectangles et toute autre forme géométrique,
- Remplir ces formes avec des couleurs ou des gradients et des textures solides,
- Dessiner du texte avec des options pour un contrôle fin sur la police et le processus de rendu,
- Appliquer des transformations géométriques ou des opérations telles que l'union, l'intersection sur des formes définies ci-dessus,
- Dessiner des images, en appliquant des transformations géométriques ou des opérations de filtrage.

Cette leçon présente quelques possibilités de l'API Java 2D. Une documentation complète est disponible sur <https://docs.oracle.com/javase/tutorial/2d/index.html>



Dessiner avec la classe Graphics2D

Pour dessiner sur différents périphériques, Java 2D utilise des classes spécifiques appelées *contextes graphiques*. Un contexte correspond à une surface de rendu graphique avec le repérage suivant :



L'API Java 2D comprend le contexte graphique `Graphics2D`, qui étend la classe `Graphics` pour donner accès à des fonctions graphiques améliorées. La classe `Graphics2D` contient des méthodes de dessin et des attributs définissant la couleur des tracés, la police de caractères, etc.

1

Dessiner des formes simples

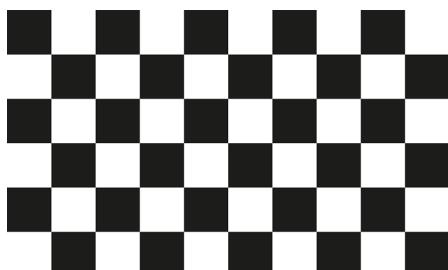
L'exemple ci-dessous dessine une ligne rouge, un rectangle vert et un ellipse bleue pleine dans le contexte graphique d'une fenêtre. La classe `FenetreGraphique` est une classe « maison » permettant d'ouvrir une fenêtre graphique et d'actualiser le dessin une fois tous les tracés réalisés.

```
FenetreGraphique fenetre = new  
FenetreGraphique("Hello_World!", 300, 250);  
  
Graphics2D contexte = fenetre.getGraphics2D();  
  
contexte.setColor(Color.RED); contexte.drawLine(10, 10, 60, 160);  
  
contexte.setColor(Color.GREEN); contexte.drawRect(100, 10, 50, 30);  
  
contexte.setColor(Color.BLUE); contexte.fillOval(200, 10, 20, 10);  
  
fenetre.actualiser();
```

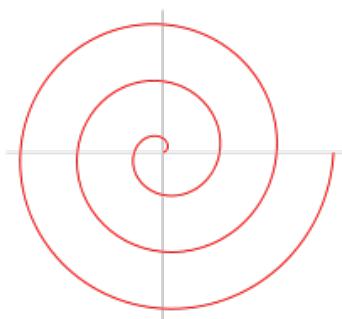
Exercice 1

1 Faire quelques essais en exécutant le fichier `TestDraw`.

2 Écrire une méthode dessinant un damier dans un contexte graphique



3 Écrire une méthode dessinant une courbe spirale de d'Archimède ($\rho = a\theta$) dans un contexte graphique.



2

Gestion des couleurs

L'espace de couleur par défaut pour l'API Java 2D est sRGB, c'est-à-dire qu'une couleur est définie par la synthèse additive des trois composantes primaires rouge, vert et bleu. Chaque composante dans l'espace sRGB est une valeur entière comprise entre 0 et 255.

Un certain nombre de couleur sont déjà définies par défaut comme `Color.RED`, `Color.ORANGE`, `Color.WHITE`, etc. Il est possible de créer une couleur personnalisée en utilisant le constructeur de la classe `Color`.

```
Color rouge = new Color(255,0,0);
Color violet = new Color(255,0,255);
Color gris = new Color(128,128,128);
Color noir = new Color(0,0,0);
```

Inversement, on peut connaître les composantes d'une couleur à l'aide des accesseurs de la classe `Color`.

```
int rouge = couleur.getRed();
int vert = couleur.getGreen();
int bleu = couleur.getBlue();
```

Exercice 2

- Écrire une méthode dessinant un rectangle dégradé horizontal dans un contexte graphique.



3

Ecrire du texte

La méthode `drawString` permet d'écrire du texte dans le contexte avec la couleur et la police de caractères choisies.

```
contexte.setColor(new Color(124,200,30));
contexte.setFont(new Font("Times_New_Roman", Font.PLAIN, 50));
contexte.drawString("Hello_world!", 0, 125);
```

La classe `Font` permet de définir une nouvelle police de caractères à partir de son nom, de son style (normal : `Font.PLAIN`, italique : `Font.ITALIC`, gras : `Font.BOLD`, etc.) et de sa taille.

Exercice 3

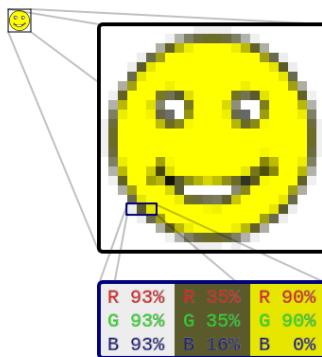
- Faire quelques essais en exécutant le fichier `TestDrawString`.

- Écrire une méthode dessinant une chaîne de caractère plusieurs fois au hasard avec des tailles et des couleurs aléatoires. On pourra utiliser la méthode `Math.random()` qui renvoie un nombre aléatoire entre 0.0 (inclus) et 1.0 (exclus).



II Gestion des images

L'API Java 2D permet également de dessiner des images matricielles. Une image matricielle (bitmap) est constituée d'une matrice de points colorés appelés pixels. Chaque pixel possède une couleur qui lui est propre et est considérée comme un point. Une image matricielle est donc une juxtaposition de pixels de couleurs formant, dans leur ensemble, une image.



1

Stocker et dessiner des images

La classe `BufferedImage` permet de stocker des images matricielles qui peuvent être ensuite dessinées dans un contexte graphique.

```
BufferedImage image = ImageIO.read(new  
File("nyancat.png"));  
contexte.drawImage(image,150,100,null);
```

Exercice 4

- 1** Faire quelques essais en exécutant le fichier `TestDrawImage`.
- 2** Modifier le fichier `TestDrawImage` pour faire défiler le nyan cat de gauche à droite dans la fenêtre.
Pour adapter la vitesse de rafraîchissement, on peut préciser un délai de 10 ms dans la méthode `fenetre.actualiser(10)`.
- 3** Modifier le fichier `estDrawImage` pour faire rebondir le nyan cat sur les bords droit et gauche de la fenêtre.
- 4** Modifier le fichier `estDrawImage` pour faire rebondir le nyan cat sur tous les bords de la fenêtre.

Transformer les images

La classe `AffineTransform` de l'API Java 2D permet d'appliquer une transformation affine (translation, rotation, échelle, etc.) à une image avant de la dessiner dans le contexte. Par exemple, le code ci-dessous agrandit une image d'un facteur 4 en X et -2 en Y et la place aux coordonnées (30,150). Le coefficient négatif sur Y a pour effet de retourner l'image (miroir horizontal).

```
AffineTransform transform = new AffineTransform();
transform.translate(30, 150);
transform.scale(4.0, -2.0);

contexte.drawImage(image, transform, null);
```

Le code suivant tourne l'image de $-\pi/4$ et la place aux coordonnées (50;50).

```
transform = new AffineTransform();
transform.translate(50, 50);
transform.rotate(-Math.PI / 4);

contexte.drawImage(image, transform, null);
```

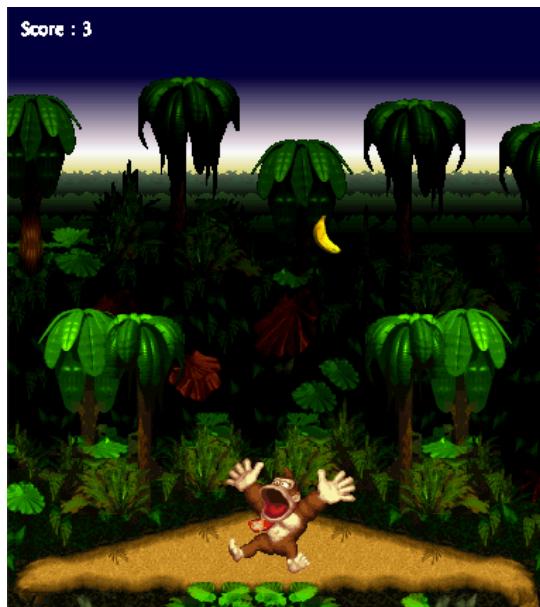
Au besoin, il est possible de préciser le centre de rotation dans le repère de l'image, par exemple pour la faire tourner autour du centre de l'image comme dans l'exemple ci-dessous.

```
transform = new AffineTransform();
transform.translate(200, 50);
transform.rotate(-Math.PI / 4, image.getWidth() / 2, image.getHeight() / 2);

contexte.drawImage(image, transform, null);
```

Réalisation d'un mini jeu vidéo avec Swing et Java 2D

Ce tutoriel montre comment réaliser un mini jeu vidéo en utilisant les bibliothèques *Swing* et *Java 2D*. Il suppose que vous ayez déjà suivi les leçons sur les interfaces graphiques et sur Java 2D (disponibles sur Moodle).



L'objectif est de réaliser un jeu où Donkey Kong doit attraper un maximum de bananes tombant des arbres. La partie se termine quand une banane lui échappe. Le product backlog est le suivant :

- En tant que joueur je veux voir les bananes tomber du haut de l'écran afin de prévoir son point de chute.
- En tant que joueur je veux que la partie s'arrête quand une banane touche le sol.
- En tant que joueur je veux déplacer mon avatar de droite à gauche avec les flèches du clavier afin de pouvoir attraper les bananes.
- En tant que joueur je veux marquer un point quand une banane touche mon avatar.



Fenêtre de jeu

La première étape consiste à ouvrir une fenêtre dans laquelle on pourra réaliser l'affichage du jeu. Vous trouverez dans le projet TP_JeuVideo une classe FenetreDeJeu qui servira de base à votre jeu vidéo.

```
public class FenetreDeJeu extends JFrame {

    private BufferedImage framebuffer;
    private Graphics2D contexte;
    private JLabel jLabel1;

    public FenetreDeJeu() {
        // initialisation de la fenêtre
        this.setSize(380, 430);
        this.setResizable(false);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this(jLabel1 = new JLabel();
        this(jLabel1.setPreferredSize(new java.awt.Dimension(380, 430));
        this.setContentPane(this(jLabel1));
        this.pack();

        // Creation du buffer pour l'affichage du jeu et recuperation du contexte graphique
        this.framebuffer = new BufferedImage(this(jLabel1.getWidth(), this(jLabel1.getHeight(), BufferedImage.TYPE_INT_ARGB);
        this(jLabel1.setIcon(new ImageIcon(framebuffer));
        this.contexte = this.framebuffer.createGraphics();
    }

    public static void main(String[] args) {
        FenetreDeJeu fenetre = new FenetreDeJeu();
        fenetre.setVisible(true);
    }
}
```

Cette classe hérite de `JForm` et définit un constructeur ainsi qu'une méthode statique `main` pour la rendre exécutable.

Elle contient un composant `JLabel` qui permet de rendre l'affichage persistant. La première partie du constructeur permet de dimensionner la fenêtre et d'associer un composant `JLabel` de mêmes dimensions à la fenêtre.

La seconde partie du constructeur a pour objectif d'associer un `framebuffer` (tampon graphique) au `JLabel` puis de récupérer un contexte graphique qui permettra de dessiner la scène du jeu à l'aide de la bibliothèque Java 2D.

- 1 Vérifiez que la classe `FenetreDeJeu` s'exécute bien (un `Clean and Build Project` peut s'avérer nécessaire).

II

Boucle de jeu

Afin de créer l'illusion d'un mouvement continu, un jeu vidéo doit être actualisé au moins 25 fois par seconde comme un dessin animé. Cette actualisation nécessite notamment de prendre en compte les commandes du joueur pour calculer le mouvement de son avatar puis de dessiner la scène correspondante. Ainsi deux étapes se répètent tant que le jeu n'est pas fini :

- L'étape de calcul du mouvement des personnages d'un jeu est appelée *mise à jour* ou *update*.
- L'étape de la synthèse de la scène d'un jeu est appelée *rendu* ou *rendering*.

Cet enchaînement est appelée la boucle de jeu.

Nous allons d'abord créer une classe `Jeu` qui contiendra tous les éléments du jeu (décor, objets, avatar, règles, etc.). La classe `Jeu` doit posséder les méthodes `miseAJour` et `rendu` qui seront appelées dans la boucle de jeu ainsi qu'une méthode booléenne `estTermine` pour savoir si la partie est terminée.

```
public class Jeu {  
    ...  
  
    public void rendu(Graphics2D contexte) {  
        // 1. Rendu du décor  
        // 2. Rendu des sprites  
        // 3. Rendu des textes  
    }  
  
    public void miseAJour() {  
        // 1. Mise à jour de l'avatar en fonction des commandes des joueurs  
        // 2. Mise à jour des autres éléments (objets, monstres, etc.)  
        // 3. Gérer les interactions (collisions et autres règles)  
    }  
  
    public boolean estTermine() {  
        // Renvoie vrai si la partie est terminée (gagnée ou perdue)  
        return false;  
    }  
}
```

- 1 Créez une nouvelle classe `Jeu` dans le projet comme spécifié ci-dessus puis ajoutez un jeu comme attribut de la fenêtre de jeu.

Il faut ensuite appeler les méthodes `miseAJour` et `rendu` du jeu 25 fois par seconde depuis notre fenêtre de jeu. Pour réaliser cet enchaînement, une solution est d'ajouter un timer à la classe `FenetreDeJeu` ce qui permet d'appeler une méthode à intervalles de temps réguliers.

Dans le constructeur de `FenetreDeJeu`, on crée un nouveau timer cadencé à 40 ms et associé à la fenêtre (d'où `this` en argument). Du coup, la classe `FenetreDeJeu` doit implémenter l'interface `ActionListener` (voir la leçon sur les interfaces) et définir la méthode `actionPerformed` qui sera appelée tous les 40 ms.

La méthode `actionPerformed` appelle successivement les méthodes `miseAJour` et `rendu` du jeu puis demande au composant `JLabel` de se redessiner pour afficher les dessins réalisés d'un seul coup (double buffering). Elle appelle enfin la méthode `estTermine` afin d'arrêter le timer si la fin du jeu est déclarée.

```
public class FenetreDeJeu extends JFrame implements ActionListener {

    private BufferedImage framebuffer;
    private Graphics2D contexte;
    private JLabel jLabel1;
    private Jeu jeu;
    private Timer timer;

    public FenetreDeJeu() {
        // initialisation de la fenetre
        this.setSize(380, 430);
        this.setResizable(false);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.jLabel1 = new JLabel();
        this.jLabel1.setPreferredSize(new java.awt.Dimension(380, 430));
        this.setContentPane(this.jLabel1);
        this.pack();

        // Creation du buffer pour l'affichage du jeu et recuperation du contexte graphique
        this.framebuffer = new BufferedImage(this.jLabel1.getWidth(), this.jLabel1.getHeight(), BufferedImage.TYPE_INT_ARGB);
        this.jLabel1.setIcon(new ImageIcon(framebuffer));
        this.contexte = this.framebuffer.createGraphics();

        // Creation du jeu
        this.jeu = new Jeu();

        // Creation du Timer qui appelle this.actionPerformed() tous les 40 ms
        this.timer = new Timer(40, this);
        this.timer.start();
    }

    // Methode appelee par le timer et qui effectue la boucle de jeu
    @Override
    public void actionPerformed(ActionEvent e) {
        this.jeu.miseAJour();
        this.jeu.rendu(contexte);
        this.jLabel1.repaint();
        if (this.jeu.estTermine()) {
            this.timer.stop();
        }
    }

    public static void main(String[] args) {
        FenetreDeJeu fenetre = new FenetreDeJeu();
        fenetre.setVisible(true);
    }
}
```

- 2 Ajoutez ces nouveaux éléments à la classe `FenetreDeJeu` et vérifiez le fonctionnement du timer en ajoutant l'affichage d'un message dans la console dans la méthode `rendu` de la classe `Jeu`.

III

Rendu du jeu

La phase de rendu a pour objectif de dessiner la scène du jeu à un instant donné. A chaque boucle, la méthode `rendu` de la classe `Jeu` doit redessiner l'intégralité de la scène.

Le rendu se fait couche par couche de l'arrière plan au premier plan. Par exemple, pour rendre une scène de *Super Mario Bros*, on dessine dans l'ordre :

- l'arrière plan (ciel, nuages, collines),
- les objets fixes (plateformes, herbes, tuyaux),
- les monstres (champignons, tortues, plantes carnivores),
- l'avatar (Mario),
- les textes (score, niveau).



Dans notre cas, nous allons pour l'instant dessiner le décor et le score du joueur.

- 1 Ajoutez à la classe `Jeu` un attribut `BufferedImage` nommé `decor` et un attribut entier nommé `score`.
- 2 Ajoutez dans le constructeur de la classe `Jeu` le chargement de l'image `jungle.png` et l'initialisation du score à 0.

```
public Jeu() {  
    try {  
        this.decor = ImageIO.read(getClass().getResource("../resources/jungle.png"));  
    } catch (IOException ex) {  
        Logger.getLogger(Jeu.class.getName()).log(Level.SEVERE, null, ex);  
    }  
    this.score = 0;  
}
```

- 3 Ajoutez dans la méthode `rendu` de la classe `Jeu` le dessin du décor et l'affichage du score.

```
public void rendu(Graphics2D contexte) {  
    contexte.drawImage(this.decor, 0, 0, null);  
    contexte.drawString("Score : " + score, 10, 20);  
}
```

A l'exécution, la fenêtre du jeu doit maintenant afficher un décor de jungle avec le score en surimpression en haut à gauche. Tout est prêt pour ajouter des bananes !

Remarque

Si il y a une erreur lors de la lecture de l'image (commande `ImageIO.read(...)`), il faut reconstruire complètement le projet avec la commande `Clean and Build Project` dans le menu `Run`.

IV

Mise à jour du jeu

Dans cette partie, nous allons ajouter des bananes tombant du ciel. Pour animer un objet, sa position doit être recalculée à chaque boucle de jeu dans l'étape de mise à jour.

De façon générale, l'étape de mise à jour du jeu doit :

- faire évoluer l'avatar en fonction des commandes des joueurs,
- faire évoluer les autres éléments (objets, monstres, etc.),
- gérer les interactions (collisions et autres règles),

Pour l'instant, nous allons nous concentrer sur les bananes ! Une classe Banane complète est fournie dans le projet.

```
public class Banane {  
  
    protected BufferedImage sprite;  
    protected double x, y;  
  
    public Banane() {  
        try {  
            this.sprite = ImageIO.read(getClass().getResource("../resources/banane.png"));  
        } catch (IOException ex) {  
            Logger.getLogger(Banane.class.getName()).log(Level.SEVERE, null, ex);  
        }  
        lancer();  
    }  
  
    public void miseAJour() {  
        y = y + 5;  
    }  
  
    public void rendu(Graphics2D contexte) {  
        contexte.drawImage(this.sprite, (int) x, (int) y, null);  
    }  
  
    public void lancer() {  
        this.x = 15 + Math.random() * 330;  
        this.y = -27;  
    }  
  
    public double getX() {  
        return x;  
    }  
  
    public double getY() {  
        return y;  
    }  
  
    public double getLargeur() {  
        return sprite.getHeight();  
    }  
  
    public double getHauteur() {  
        return sprite.getWidth();  
    }  
}
```

Comme tous les éléments d'un jeu, la classe Banane contient les méthodes `rendu` et `miseAJour`.

La méthode `rendu` dessine le sprite de la banane aux coordonnées `x` et `y`. Un sprite est une petite image en partie transparente qui représente un élément animé du jeu (objet, un monstre, avatar).

La méthode `miseAJour` augmente l'ordonnée de la banane de 5 pixels à chaque appel donc si la boucle du jeu est exécutée 25 fois par seconde, la banane va se déplacer vers le bas de 5 pixels tous les 40 ms.

Enfin, la méthode `lancer` permet de replacer la banane en haut de la scène à une abscisse aléatoire.

Nous allons maintenant ajouter une banane à la classe Jeu.

- 1** Ajoutez un attribut Banane nommé uneBanane à la classe Jeu.
- 2** Ajoutez dans le constructeur de la classe Jeu une ligne pour créer une banane .
- 3** Ajoutez le rendu de la banane dans la méthode rendu de la classe Jeu.
- 4** Ajoutez la mise à jour de la banane dans la méthode miseAJour de la classe Jeu.

A l'exécution, une banane doit maintenant tomber du haut de la fenêtre avant de disparaître. Pour éviter que la banane ne sorte de la fenêtre, nous devons arrêter le jeu quand la banane touche le sol.

- 5** Ajoutez un test dans la méthode estTermine de la classe Jeu renvoyant vrai si l'ordonnée de la banane est supérieure à 400 pixels.

```
public boolean estTermine() {  
    return this.uneBanane.getY() > 400;  
}
```

Cette fois, la fenêtre du jeu doit se figer quand la banane touche le sol : la partie est terminée !



Gestion des commandes du joueur

Nous allons maintenant voir comment faire évoluer Donkey Kong dans la scène du jeu. Pour commencer, nous allons créer une classe Avatar sur le modèle de la classe Banane.

- 1** Créez une classe Avatar identique à la classe Banane (on pourra utiliser la commande Copy avec un clic droit sur la classe Banane puis la commande Paste->Refactor Copy avec un clic droit sur le package Jeu).
- 2** Ajoutez à la classe Jeu un attribut Avatar et les commandes nécessaires dans le constructeur et dans les méthodes rendu et miseAJour.

Si vous lancez la fenêtre de jeu, Donkey Kong doit tomber du ciel ! N'oubliez pas de remplacer l'image banane.png par donkeyKong.png dans le constructeur de la classe Avatar.

- 3** Modifiez le constructeur de la classe Avatar pour placer Donkey Kong aux coordonnées (170;320) et supprimez la modification de l'ordonnée dans la méthode miseAJour.

Donkey Kong est maintenant fixe en bas de la scène du jeu et nous allons voir comment le faire bouger de droite à gauche à l'aide des flèches du clavier.

Pour récupérer les événements liés au clavier, on utilise un écouteur clavier (voir la leçon sur les interfaces graphiques). Il faut pour cela que la classe FenetreDeJeu implémente l'interface KeyListener et définisse les méthodes keyTyped, keyPressed et keyReleased.

```
public class FenetreDeJeu extends JFrame implements ActionListener, KeyListener {  
    ...  
  
    public FenetreDeJeu() {  
        ...  
  
        // Ajout d'un écouteur clavier  
        this.addKeyListener(this);  
    }  
  
    ...  
  
    @Override  
    public void keyTyped(KeyEvent evt) {  
    }  
  
    @Override  
    public void keyPressed(KeyEvent evt) {  
        if (evt.getKeyCode() == evt.VK_RIGHT) {  
            this.jeu.getAvatar().setDroite(true);  
        }  
        if (evt.getKeyCode() == evt.VK_LEFT) {  
            this.jeu.getAvatar().setGauche(true);  
        }  
    }  
  
    @Override  
    public void keyReleased(KeyEvent evt) {  
        if (evt.getKeyCode() == evt.VK_RIGHT) {  
            this.jeu.getAvatar().setDroite(false);  
        }  
        if (evt.getKeyCode() == evt.VK_LEFT) {  
            this.jeu.getAvatar().setGauche(false);  
        }  
    }  
}
```

4 Modifiez la classe FenetreDeJeu en conséquence.

5 Ajoutez le sélecteur getAvatar à la classe Jeu et les modificateurs setDroite et setGauche à la classe Avatar.

Remarque

On ne modifie pas directement les coordonnées de l'avatar par un simple setAbscisse car il faut respecter ses règles de déplacement! A la place, nous indiquons à l'avatar que le joueur souhaite l'envoyer vers la droite ou vers la gauche. La méthode miseAJour de l'avatar prendra ensuite en compte cette demande en respectant ses règles de déplacement et une seule fois par boucle de jeu.

Il reste maintenant à implanter les règles de déplacement de l'avatar.

```
public class Avatar {  
  
    protected BufferedImage sprite;  
    protected double x, y;  
    private boolean gauche, droite;  
  
    public Avatar() {  
        try {  
            this.sprite = ImageIO.read(getClass().getResource("../resources/donkeyKong.png"));  
        } catch (IOException ex) {  
            Logger.getLogger(Avatar.class.getName()).log(Level.SEVERE, null, ex);  
        }  
        this.x = 170;  
        this.y = 320;  
        this.gauche = false;  
        this.droite = false;  
    }  
  
    public void miseAJour() {  
        if (this.gauche) {  
            x -= 5;  
        }  
        if (this.droite) {  
            x += 5;  
        }  
        if (x > 380 - sprite.getWidth()) { // collision avec le bord droit de la scène  
            x = 380 - sprite.getWidth();  
        }  
        if (x < 0) { // collision avec le bord gauche de la scène  
            x = 0;  
        }  
    }  
  
    public void rendu(Graphics2D contexte) {  
        contexte.drawImage(this.sprite, (int) x, (int) y, null);  
    }  
  
    public void setGauche(boolean gauche) {  
        this.gauche = gauche;  
    }  
  
    public void setDroite(boolean droite) {  
        this.droite = droite;  
    }  
  
    public double getX() {  
        return x;  
    }  
  
    public double getY() {  
        return y;  
    }  
  
    public double getLargeur() {  
        return sprite.getHeight();  
    }  
  
    public double getHauteur() {  
        return sprite.getWidth();  
    }  
}
```

6 Modifiez la classe Avatar en conséquence.

Donkey Kong peut maintenant se déplacer de droite à gauche quand on appuie sur les flèches du clavier et sans sortir de la scène. Dernière étape, Donkey Kong doit attraper les bananes !

Pour que Donkey Kong attrape une banane, il faut détecter si il entre en contact avec la banane, en d'autres termes, il faut détecter une collision.

Il existe plusieurs techniques pour détecter les collisions allant du simple calcul de la distance entre deux points, à l'utilisation de hitboxes (boîtes de collisions) comme illustré ci-dessous.



Pour notre jeu, nous allons détecter la collision entre les deux rectangles entourant les sprites (comme dans le jeu *Gradius* à gauche). Dans cette configuration, il en réalité plus simple de détecter l'absence de collision qui se réduit à quatre tests. La méthode suivante renvoie vrai si il y a une collision entre l'avatar et une banane dans la classe *Jeu*.

```
public boolean collisionEntreAvatarEtBanane() {
    if ((uneBanane.getX() >= avatar.getX() + avatar.getLargeur()) // trop à droite
        || (uneBanane.getX() + uneBanane.getLargeur() <= avatar.getX()) // trop à gauche
        || (uneBanane.getY() >= avatar.getY() + avatar.getHauteur()) // trop en bas
        || (uneBanane.getY() + uneBanane.getHauteur() <= avatar.getY())) { // trop en haut
        return false;
    } else {
        return true;
    }
}
```

Il suffit ensuite d'utiliser ce test de collision dans la méthode `miseAJour` de la classe *Jeu* pour, le cas échéant, augmenter le score et relancer la banane en haut de la scène.

1 Modifiez la classe *Jeu* en conséquence.

Normalement, Donkey Kong peut enfin attraper les bananes et votre mini jeu est terminé !

Vous avez sans doute remarqué une certaine similitude entre les classes Avatar et Banane. En effet, tous les éléments d'un jeu possèdent des propriétés communes : ils ont tous une position dans la scène et une image qui les représente, ils peuvent entrer en collision les uns les autres, etc. En revanche, ils ont des comportements différents, par exemple ils sont soit pilotés par le jeu soit par les joueurs.

Une technique intéressante pour simplifier la gestion de tous les éléments d'un jeu est d'utiliser l'héritage et le polymorphisme. Une leçon consacrée à l'héritage est disponible sur Moodle.

Une autre technique bien connue pour réaliser de grands décors dans un jeu est le *tuilage* ou *tile mapping*. Un tutoriel dédié est disponible sur Moodle.

Pour plus d'informations sur l'animation de sprites, vous pouvez consulter le tutoriel d'Alexandre Laurent disponible à l'adresse <https://alexandre-laurent.developpez.com/tutoriels/programmation-jeux/animations-2D/>

Pour plus d'informations sur les collisions, vous pouvez consulter le tutoriel de Fvirtman disponible à l'adresse <https://jeux.developpez.com/tutoriels/theorie-des-collisions/>

Tile mapping

Le *tile mapping* (ou tuilage) est une technique très utilisée dans les jeux vidéos en 2D. Le principe est de créer les décors du jeu en associant de petites images appelées tuiles (*tiles* en anglais). On retrouve cette technique dans des jeux en vue de côté comme *Super Mario Bros*, dans des jeux en vue de dessus comme *Pac-man*, ou encore dans des jeux en perspective isométrique comme *Civilization II*.

Ce tutoriel montre comment réaliser un tile mapping avec l'API Java 2D. Pour suivre ce tutoriel, on s'appuiera sur le projet NetBeans TutorielTileMapping qui contient quelques classes formant une base de jeu (voir tutoriel sur la réalisation d'une boucle de jeu). Le projet contient notamment une classe Carte qui aura pour objectif de dessiner le décor du jeu.



FIGURE 7.1 – Illustration du principe du tile mapping (extrait du tutoriel de Fvirtman disponible sur le site developpez.com).



Définition d'une carte

Une carte est définie par sa largeur et sa hauteur en nombre de tuiles ainsi que par la taille en pixels d'une tuile. Une première version de la classe Carte est fournie dans le projet.

```
public class Carte {  
  
    private int largeur = 12;  
    private int hauteur = 9;  
    private int tailleTuile = 32;  
  
    private BufferedImage uneTuile;  
  
    public Carte() {  
        try {  
            BufferedImage tileset = ImageIO.read(getClass().getResource("images/tileSetMinecraft32x32.png"));  
            uneTuile = tileset.getSubimage(0, 0, tailleTuile, tailleTuile);  
        } catch (IOException ex) {  
            Logger.getLogger(Carte.class.getName()).log(Level.SEVERE, null, ex);  
        }  
    }  
  
    public void miseAJour() {  
    }  
}
```

```

    }

    public void rendu(Graphics2D contexte) {
        contexte.drawImage(uneTuile, 0, 0, null);
    }
}

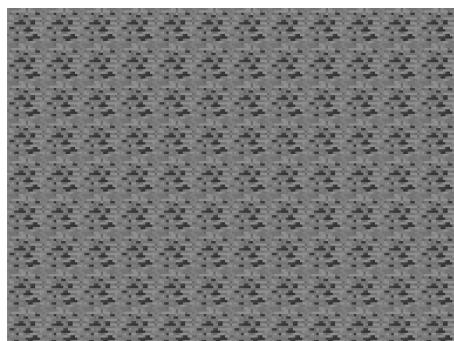
```

Dans le constructeur, on charge une image spéciale, le *tileset*, qui contient toutes les tuiles possibles. On récupère ensuite la tuile située en haut à gauche à l'aide de la commande `getSubimage`.



Comme toutes les entités d'un jeu vidéo, la classe `Carte` contient les méthodes `miseAJour` et `rendu` qui seront appelées à chaque boucle de jeu. La méthode `rendu` a pour mission de dessiner le décor du jeu dans le contexte graphique passé en argument. Dans cette première version, on dessine simplement la tuile récupérée aux coordonnées `(0;0)` à l'aide de la commande `drawImage`.

- 1** Tester la classe `Carte` fournie en exécutant la classe `FenetreDeJeu`.
- 2** Changer les coordonnées de `getSubimage` pour récupérer la troisième tuile de la deuxième ligne (pierre avec du charbon).
- 3** A l'aide de deux boucles pour imbriquées, modifier la méthode `rendu` pour réaliser un tuilage (damier) correspondant à la largeur et à la hauteur de la carte en dessinant la tuile récupérée autant de fois que nécessaire.



II Préparation des tuiles

On souhaite maintenant stocker toutes les tuiles du tileset dans un tableau de façon à pouvoir y accéder à l'aide d'un numéro sans se préoccuper des coordonnées.

On commence par ajouter un tableau de `BufferedImage` aux attributs de la classes pour stocker les tuiles :

```
private BufferedImage[] tuiles;
```

Le tileset fourni dans le projet contient 176 tuiles de 32×32 pixels répartis sur 16 colonnes et 11 lignes. Il faut donc créer un tableau de 176 images dans le constructeur la carte.

```
tuiles = new BufferedImage[176];
```

On remplit ensuite le tableau avec toutes les tuiles de gauche à droite et de haut en bas avec une simple boucle pour :

```
for (int i = 0; i < tuiles.length; i++) {  
    int x = (i % 16) * tailleTuile;  
    int y = (i / 16) * tailleTuile;  
    tuiles[i] = tileset.getSubimage(x, y, tailleTuile, tailleTuile);  
}
```

- 1 Modifier le constructeur de la classe `Carte` afin de charger toutes les tuiles dans le tableau.
- 2 Modifier la méthode `rendu` pour afficher le tuilage avec la tuile numéro 3 (terre).

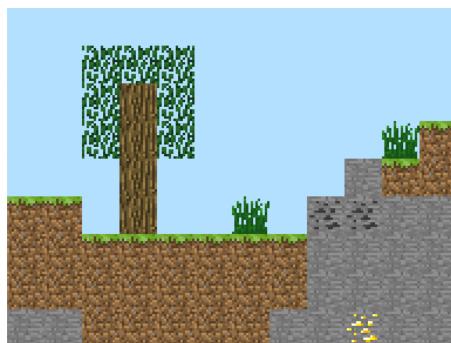
III Réalisation du tile mapping

Le principe du tile mapping repose sur un tableau à deux dimensions qui stocke le numéro des tuiles à afficher dans chaque case. Ce tableau permet donc de représenter le décor du jeu sous une forme plus compacte. Pour dessiner le décor, il suffit de parcourir le tableau et de dessiner chaque tuile à sa place dans la fenêtre.

Pour commencer, on peut définir un tableau directement dans la classe `Carte` :

```
private int[][] decor = {  
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  
    { 0, 0, 26, 26, 26, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  
    { 0, 0, 26, 20, 26, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  
    { 0, 0, 26, 20, 26, 0, 0, 0, 0, 0, 0, 0, 57, 2},  
    { 0, 0, 0, 20, 0, 0, 0, 0, 0, 0, 1, 2, 3},  
    { 2, 2, 0, 20, 0, 0, 57, 0, 18, 18, 1, 1},  
    { 3, 3, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1},  
    { 3, 3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1},  
    { 1, 1, 3, 3, 3, 3, 3, 1, 1, 16, 1, 1}
```

Ce qui correspond au décor suivant :



- 1 Ajouter le tableau à la classe `Carte` et modifier la méthode `rendu` pour afficher le décor en consultant le numéro de chaque tuile dans le tableau.

Remarque

Le tableau doit avoir le même nombre de colonne que la largeur de la carte et le même nombre de ligne que la hauteur de la carte.

IV

Stockage d'une carte dans un fichier

La réalisation d'un grand décor peut être fastidieux si on doit le faire directement sous la forme d'un tableau Java. Il est préférable de stocker les informations d'une carte dans un fichier texte puis de recréer le tableau automatiquement à partir de ce fichier.

Cette méthode permet notamment d'utiliser un logiciel pour dessiner un décor de façon interactive. Le projet Netbeans fournit le petit éditeur nommé *FruitEditor* de Nico Poblete. On propose de commencer par réaliser un fichier contenant une carte avec cet éditeur.

- 1 Exécuter la classe *Main* contenue dans le package *FruitEditor*.
- 2 Créer une nouvelle carte (*File->New*) en précisant un nom, une largeur de 30 tuiles, une hauteur de 20 tuiles et une taille de tuile de 32x32 pixels.
- 3 Charger le tileset *tileSetMinecraft32x32.png* à l'aide d'un clic droit dans le panneau de gauche (*Open Tileset*).
- 4 Dessiner un décor puis enregistrer la carte dans un fichier texte (*File->Save as*).

Le fichier texte produit par *FruitEditor* est de la forme suivante (pour l'exemple précédent) :

```
Exemple 1
images/tileSetMinecraft32x32.png
12 9 32 32
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 26 26 26 0 0 0 0 0 0 0 0
0 0 26 20 26 0 0 0 0 0 0 0 0
0 0 26 20 26 0 0 0 0 0 0 57 2
0 0 0 20 0 0 0 0 0 0 1 2 3
2 2 0 20 0 0 57 0 18 18 1 1
3 3 2 2 2 2 2 2 1 1 1 1
3 3 3 3 3 3 3 3 1 1 1 1
1 1 3 3 3 3 3 1 1 16 1 1
```

- La première ligne contient le nom de la carte utile par exemple pour identifier le décor ou le niveau du jeu.
- La deuxième ligne renseigne sur le nom et le chemin du fichier contenant les tuiles (tileset).
- La troisième ligne contient dans l'ordre : la largeur de la carte, la hauteur de la carte, la taille d'une tuile en pixel (largeur et hauteur).
- Les lignes suivantes contiennent les numéros des tuiles pour chaque case du tableau *decor*.

- 5 Ajouter un nouveau constructeur à la classe *Carte* permettant de construire une carte à partir des informations stockées dans un fichier texte.

Remarque

Pour la lecture des fichiers textes avec Java, consulter la leçon consacrée à ce sujet dans le polycopié de première année.