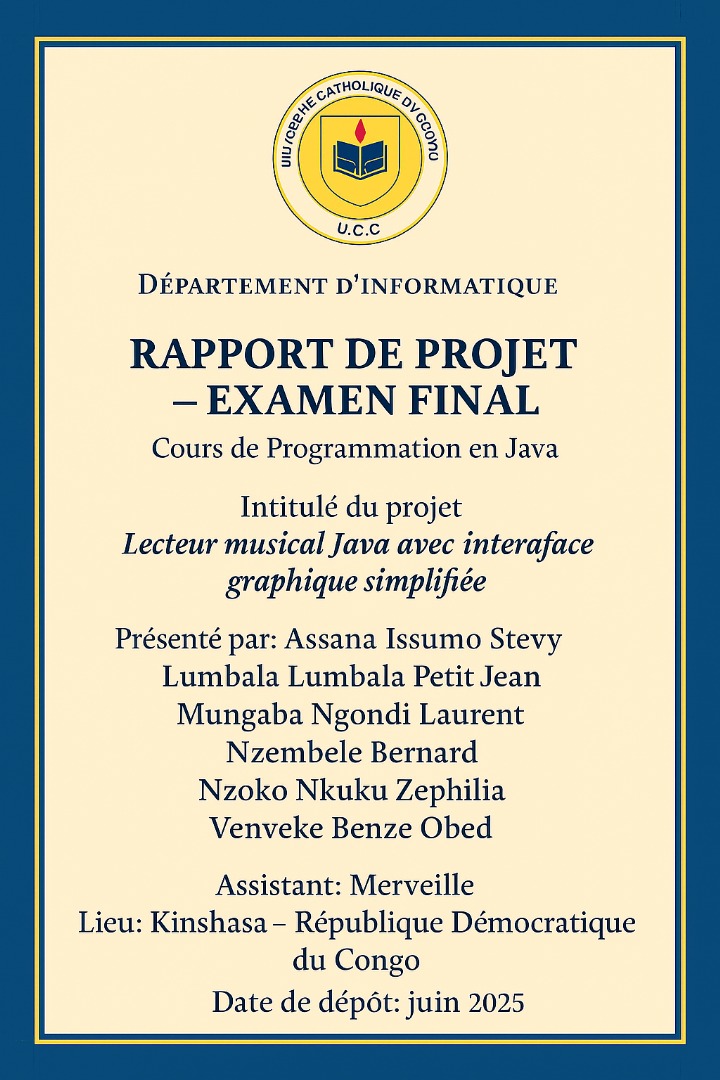
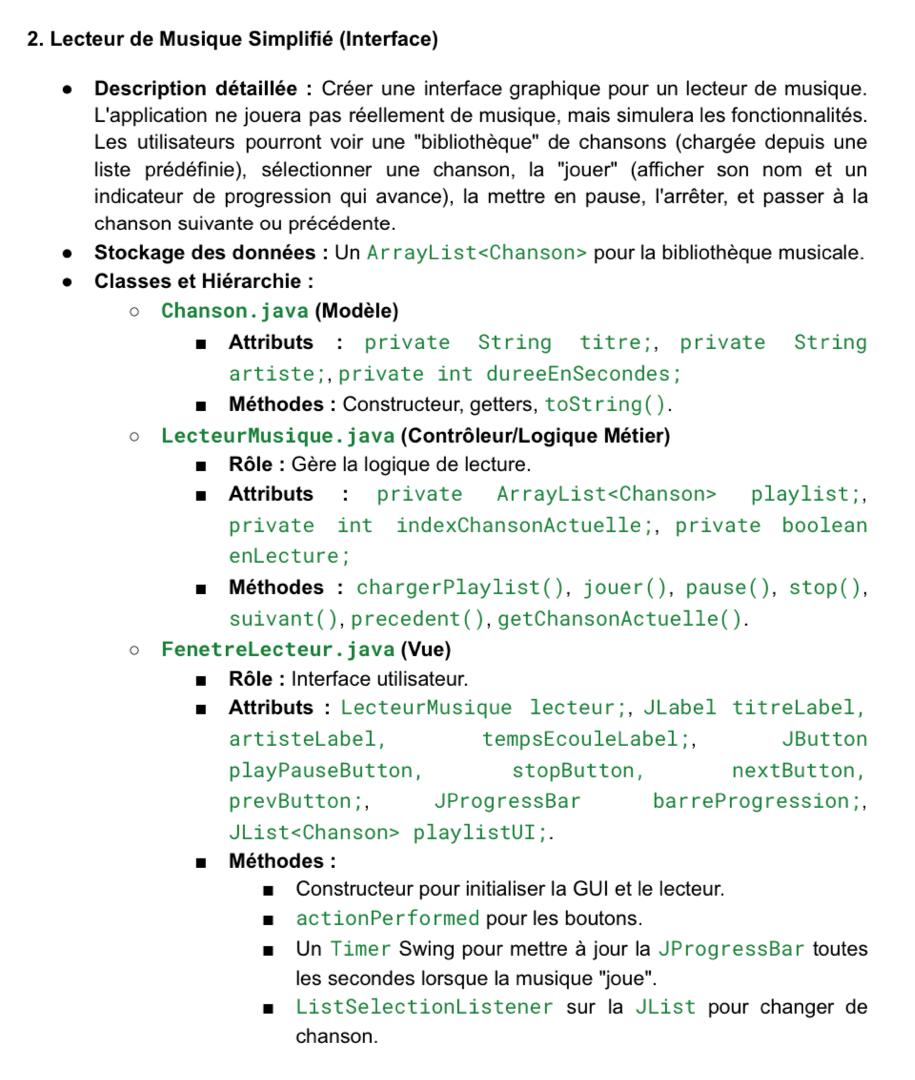
****

****

**Objectif du projet**

Créer un **lecteur de musique visuel immersif** en Java, avec pour particularité :

* Une **interface néon/glow personnalisable**
* Un système de navigation fluide (playlist, albums…)
* Des **boutons interactifs avec effets visuels**
* une **barre de progression animée**
* un **visualiseur simulé**, donnant l’illusion d’un égaliseur audio
* une structure de code en **MVC (Modèle / Vue / Contrôleur)**

L’approche combine **design graphique, gestion de données et logique Java Swing**, tout en respectant l’esthétique Glow.

**Architecture logique (MVC)**

Le projet a été bâti autour de 3 classes principales :

| **Classe** | **Rôle** |
| --- | --- |
| Chanson.java | Représente chaque chanson (titre, artiste, durée) |
| LecteurMusique.java | Gère la playlist + les actions suivantes/précédentes |
| FenetreLecteurV6.java | Gère toute l’interface graphique et les interactions utilisateur |

Ce modèle permet une séparation claire du code :

* La classe Chanson manipule des **données**
* Le LecteurMusique applique la **logique de contrôle**
* Et la FenetreLecteurV6 affiche le **visuel** et gère les clics utilisateurs

**Organisation du dossier du projet**

/src

/com

/leteurplayer

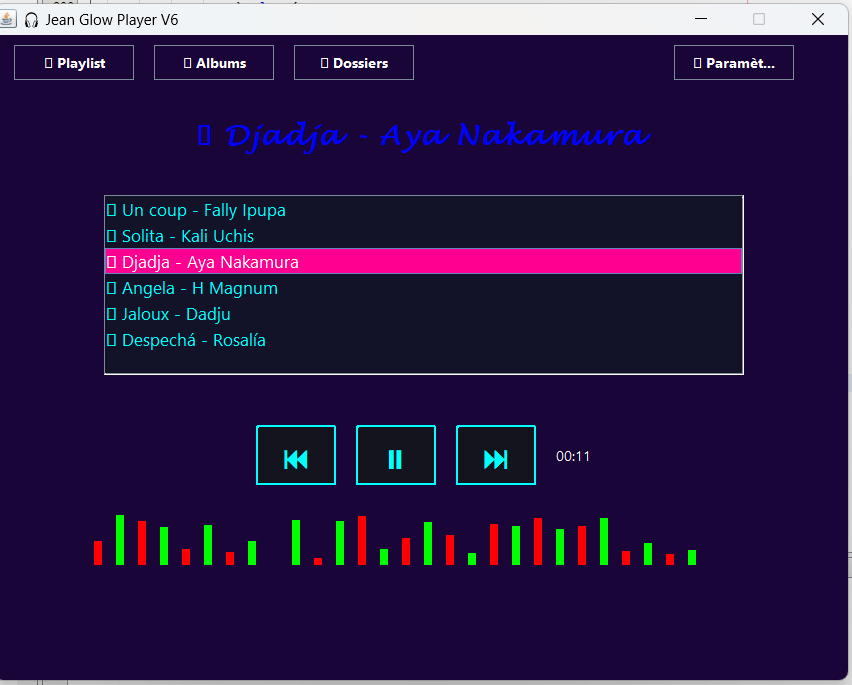
├── Chanson.java

├── LecteurMusique.java

└── FenetreLecteurV6.java

**Illustration du lecteur musique simplifie — vue complète**

Et voici une illustration du lecteur Musique simplifie :



Chapitre suivant e : **Chanson.java** + explication ligne par ligne + mini illustration de la structure

**Chapitre 2 — La classe Chanson.java**

**Rôle de la classe**

La classe Chanson représente une chanson individuelle dans ton lecteur. C’est un **modèle de données** qui stocke :

* Le **titre** de la chanson
* Le **nom de l'artiste**
* Sa **durée en secondes**

Chaque chanson est un objet Java, comme une brique de ta playlist.

**⌨️ Syntaxe Java complète**

Java

Public class Chanson {

private String titre, artiste;

private int dureeEnSecondes;

public Chanson(String titre, String artiste, int duree) {

this.titre = titre;

this.artiste = artiste;

this.dureeEnSecondes = duree;

}

public String getTitre() { return titre; }

public String getArtiste() { return artiste; }

public int getDuree() { return dureeEnSecondes; }

@Override

public String toString() {

return "♫ " + titre + " - " + artiste;

}

}

**Explication ligne par ligne**

| **Ligne** | **Explication** |
| --- | --- |
| public class Chanson | Déclaration d'une classe publique nommée Chanson |
| private String titre, artiste; | Deux variables texte privées : le titre et l’artiste |
| private int dureeEnSecondes; | Un entier stockant la durée de la chanson |
| public Chanson(...) | Constructeur : appelé lors de la création d’une nouvelle chanson |
| getTitre(), getArtiste() | Méthodes d’accès (getters) pour lire les attributs |
| toString() | Méthode spéciale qui définit comment une chanson s’affiche en texte |

💡 Grâce à toString (), chaque chanson apparaît dans la liste comme :

plaintext

♫ Un coup - Fally Ipupa

**Illustration : représentation conceptuelle d'une Chanson**

*[Image générée : une boîte contenant “titre”, “artiste”, “durée” avec une étiquette “♫ titre - artiste”]* → Je te fournirai cette illustration dans le bloc visuel final

# Chapitre 3 — La classe LecteurMusique.java

## Rôle de cette classe

C’est ton **cerveau de gestion musicale** : il contient la playlist, suit la chanson en cours, et contrôle les actions “suivant” ou “précédent”. Il ne s’occupe ni d’affichage graphique, ni d’animation — juste de **logique métier** (le “C” du MVC).

### ⌨️ Code complet

java

public class LecteurMusique {

private ArrayList<Chanson> playlist = new ArrayList<>();

private int indexActuel = -1;

public void ajouterChanson(Chanson chanson) {

playlist.add(chanson);

if (indexActuel == -1) indexActuel = 0;

}

public Chanson getChansonActuelle() {

if (indexActuel >= 0 && indexActuel < playlist.size())

return playlist.get(indexActuel);

return null;

}

public void suivant() {

if (playlist.size() > 0)

indexActuel = (indexActuel + 1) % playlist.size();

}

public void precedent() {

if (playlist.size() > 0)

indexActuel = (indexActuel - 1 + playlist.size()) % playlist.size();

}

public void setIndexActuel(int index) {

if (index >= 0 && index < playlist.size()) indexActuel = index;

}

}

## 🧾 Explication détaillée ligne par ligne

| **Élément** | **Explication** |
| --- | --- |
| ArrayList<Chanson> playlist | Liste dynamique contenant les objets Chanson |
| int indexActuel | Position actuelle de lecture dans la playlist |
| ajouterChanson() | Ajoute une nouvelle chanson à la fin, et initialise indexActuel si besoin |
| getChansonActuelle() | Retourne la chanson jouée actuellement (ou null si aucune) |
| suivant() | Avance au morceau suivant. % permet de boucler à zéro |
| precedent() | Revient au morceau précédent. La formule avec + playlist.size() évite les indices négatifs |
| setIndexActuel(int i) | Permet de définir manuellement la chanson sélectionnée |

### Illustration conceptuel

### Prochain chapitre en cours : **Chapitre 4 — FenetreLecteur.java** visuel.

# Chapitre 4 — La classe FenetreLecteurV6.java : Interface graphique et interactivité

## 🧠 Rôle de la classe

Cette classe crée la **fenêtre principale de l’application**. Elle s’occupe de :

* Construire tous les composants Swing (boutons, listes, labels…)
* Appliquer les **styles néon/glow** à chaque élément
* Connecter les actions utilisateur aux fonctions du lecteur (lecture, pause…)
* Mettre en place les timers pour le **simulateur** et le **visualiseur**

Elle donne **forme et vie** à ton lecteur.

## Composition générale de la classe

| **Partie** | **Fonction** |
| --- | --- |
| Constructeur FenetreLecteurV6() | Initialise tous les composants, organise les panels, configure styles |
| Méthode createTopButton() | Crée des boutons de navigation uniformisés (Playlist, Albums…) |
| Méthode createModernNavButton() | Crée les boutons ⏮ ⏯ ⏭ avec effet Glow |
| Timer simulateur | Fait avancer la barre de progression toutes les secondes |
| Timer equalizerTimer | Anime le visualiseur (rectangles) à intervalles réguliers |

### Exemple : Bouton de navigation Playlist

java

btnPlaylist = createTopButton("🎵 Playlist", 20);

Et la méthode correspondante :

java

private JButton createTopButton(String text, int x) {

JButton btn = new JButton(text);

btn.setBounds(x, 10, 120, 35);

btn.setFont(new Font("Segoe UI", Font.BOLD, 14));

btn.setForeground(Color.white);

btn.setBackground(themes[themeIndex]); // Couleur dynamique selon thème

btn.setCursor(new Cursor(Cursor.HAND\_CURSOR));

return btn;

}

🔎

* (x, y, w, h) : position absolue du bouton
* setFont(...) : change la police, la taille, le style
* setForeground() : couleur du texte
* setBackground() : couleur de fond
* Cursor.HAND\_CURSOR : curseur en forme de main au survol

### 🎶 Liste des chansons (playlist)

java

modele = new DefaultListModel<>();

liste = new JList<>(modele);

liste.setBackground(new Color(18,18,40)); // Fond très foncé

liste.setForeground(Color.cyan); // Texte néon

liste.setSelectionBackground(Color.decode("#ff0090")); // Rose clair au clic

🧠 La playlist est ajoutée dans un JScrollPane, ce qui permet le défilement s il y a trop de chansons.

### ⏯ Bouton Lecture / Pause stylisé

java

JButton btnPlayPause = new JButton("⏯");

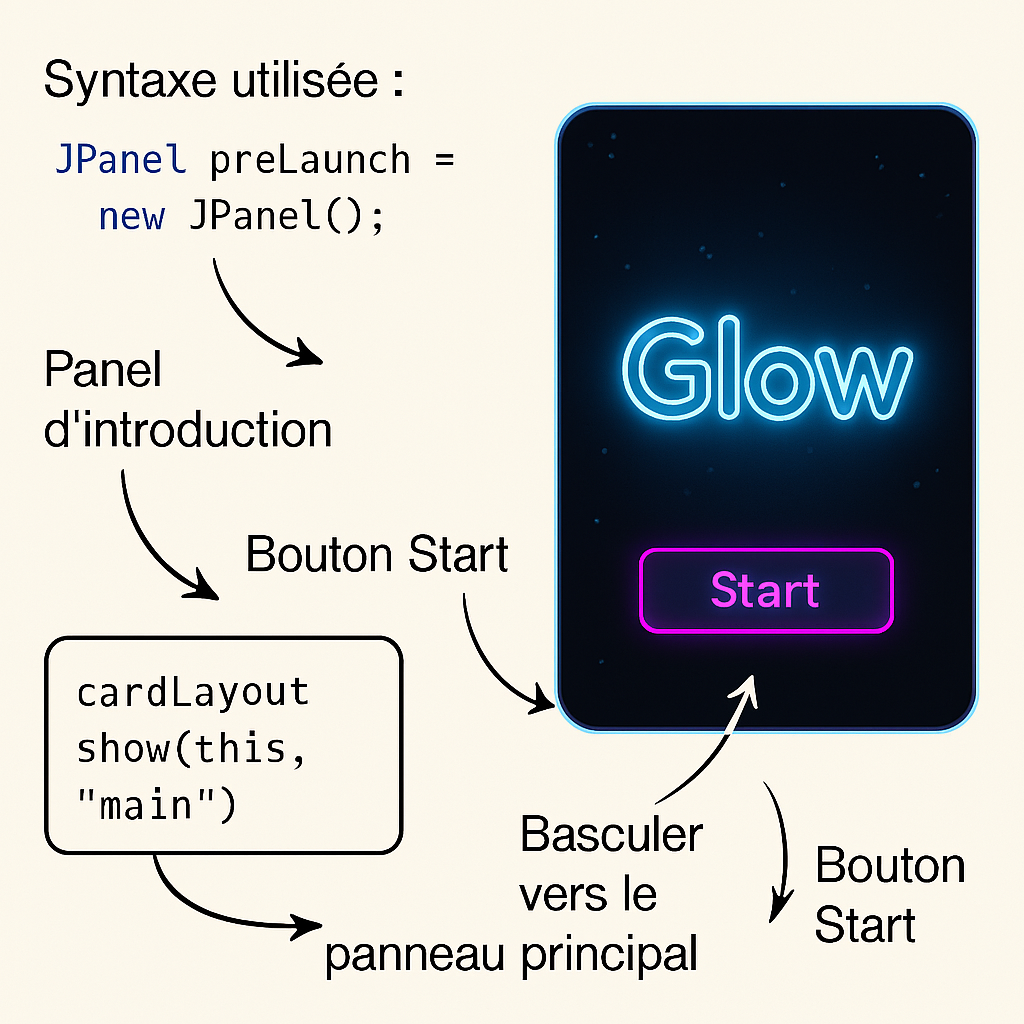
btnPlayPause.setFont(new Font("Segoe UI Symbol", Font.BOLD, 30));

btnPlayPause.setForeground(Color.cyan);

btnPlayPause.setBackground(new Color(20, 20, 30));

btnPlayPause.setBorder(BorderFactory.createLineBorder(Color.decode("#00ffff"), 2, true));

Cette bordure cyan + fond sombre = **effet Glow assuré**

****

Prochaines parties dans ce chapitre :

* **simulateur avec Timer → pour faire défiler le temps**
* **visualiseur animé (paintComponent)**
* **tableaux clairs des composants et couleurs**
* Et pour chaque, une image et une explication ligne par ligne

# Chapitre 4.1 — Bandeau supérieur (Navigation

## Code Java

Java

|  |
| --- |
| btnPlaylist = createTopButton("🎵 Playlist", 20);  btnAlbums = createTopButton("📀 Albums", 160);  btnDossiers = createTopButton("📂 Dossiers", 300);  btnParametres = createTopButton("⚙ Paramètres", 680); |

Et voici la méthode createTopButton() utilisée pour styliser chaque bouton :

Java

|  |
| --- |
| private JButton createTopButton(String texte, int x) {  JButton btn = new JButton(texte);  btn.setBounds(x, 10, 120, 35);  btn.setFont(new Font("Segoe UI", Font.BOLD, 14));  btn.setForeground(Color.white);  btn.setBackground(themes[themeIndex]); // couleur en fonction du thème actif  btn.setCursor(new Cursor(Cursor.HAND\_CURSOR));  return btn; |

}

## Explication ligne par ligne

| **Élément** | **Rôle** |
| --- | --- |
| new JButton(texte) | Crée un bouton avec une étiquette texte et une icône Unicode |
| setBounds(x, 10, 120, 35) | Positionne le bouton horizontalement à x, avec une taille fixe |
| setFont(...) | Définit la police en gras, 14 pts, look moderne |
| setForeground(Color.white) | Texte blanc pour une bonne lisibilité |
| setBackground(themes[themeIndex]) | Couleur de fond prise dans le tableau dynamique themes[] |
| setCursor(HAND\_CURSOR) | Curseur de souris se transforme en main au survol |

🧠 Grâce à cette méthode, tu peux **réutiliser le même style** pour plusieurs boutons sans répéter le code.

## Astuce UI

Java

|  |
| --- |
| btn.addMouseListener(new MouseAdapter() {  public void mouseEntered(MouseEvent e) {  btn.setBackground(Color.decode("#00ccff"));  }  public void mouseExited(MouseEvent e) {  btn.setBackground(themes[themeIndex]);  }  }); |

🎯 Cela donne une sensation d'interactivité très “Glow”, surtout combinée à une transition de couleur douce.

🪄 Prochaine section dans quelques instants : **4.2 — Playlist dynamique avec** JList**, gestion des couleurs et sélection**

# Chapitre 4.2 — Playlist dynamique (JList) avec couleurs Glow

## Illustration

## Code Java complet

Java

|  |
| --- |
| DefaultListModel<String> modele = new DefaultListModel<>();  JList<String> liste = new JList<>(modele);  liste.setBackground(new Color(18,18,40)); // Fond sombre  liste.setForeground(Color.cyan); // Texte bleu néon  liste.setSelectionBackground(Color.decode("#ff0090")); // Fond de sélection rose  liste.setSelectionForeground(Color.white); // Texte sélectionné en blanc |

Et pour ajouter les chansons :

Java

|  |
| --- |
| String[][] chansons = {  {"Un coup", "Fally Ipupa"}, {"Solita", "Kali Uchis"},  {"Djadja", "Aya Nakamura"}, {"Angela", "H Magnum"}, |
| {"Jaloux", "Dadju"}, {"Despechá", "Rosalía"}  };  for (String[] c : chansons) {  modele.addElement("♫ " + c[0] + " - " + c[1]);  } |

## Explication ligne par ligne

| **Syntaxe** | **Signification** |
| --- | --- |
| DefaultListModel<String> modele | Conteneur dynamique qui contient les éléments de la liste (JList) |
| new JList<>(modele) | Création de la liste avec les valeurs du modèle |
| setBackground(Color) | Détermine la couleur de fond de la liste |
| setForeground(Color) | Définit la couleur du texte non sélectionné |
| setSelectionBackground(Color) | Couleur de fond quand une ligne est sélectionnée |
| setSelectionForeground(Color) | Couleur du texte de la ligne sélectionnée |

## Résultat visuel attendu

* La playlist s’affiche avec une ambiance **Glow sombre**
* Lorsqu’on clique sur une chanson, elle se **surligne en rose néon**
* Grâce à l’utilisation de DefaultListModel, tu peux **ajouter ou supprimer des chansons dynamiquement**

Et n’oublie pas : cette JList est généralement placée dans un JScrollPane pour permettre le **défilement vertical** si ta playlist s’allonge :

java

JScrollPane scroll = new JScrollPane(liste);

scroll.setBounds(50, 140, 720, 170); // Positionnement dans ta fenêtre

Et ensuite, on enchaîne avec :

* le timer & la barre de progression
* le visualiseur graphique
* les tableaux récap + lexique



# Chapitre 4.3 — Boutons interactifs ⏮ ⏯ ⏭

## Rôle des boutons

Ces trois boutons permettent :

* ⏮ : revenir à la chanson précédente
* ⏯ : lire ou mettre en pause la lecture
* ⏭ : passer à la chanson suivante

## Code Java utilisé

java

private JButton createModernNavButton(String symbole) {

JButton btn = new JButton(symbole);

btn.setFont(new Font("Segoe UI Symbol", Font.BOLD, 30));

btn.setPreferredSize(new Dimension(80, 60));

btn.setForeground(Color.cyan);

btn.setBackground(new Color(20, 20, 30));

btn.setBorder(BorderFactory.createLineBorder(Color.decode("#00ffff"), 2, true));

// Effet de survol (hover)

btn.addMouseListener(new MouseAdapter() {

public void mouseEntered(MouseEvent evt) {

btn.setBackground(Color.decode("#00ccff"));

}

public void mouseExited(MouseEvent evt) {

btn.setBackground(new Color(20, 20, 30));

}

});

return btn;

}

## Détail ligne par ligne

| **Ligne** | **Explication** |
| --- | --- |
|  |  |
| new JButton(symbole) | Crée un bouton contenant un caractère Unicode (⏯, ⏮, ⏭) |
| setFont(...) | Police lisible et large pour mettre en valeur l’icône |
| setPreferredSize(new Dimension(80, 60)) | Taille standardisée pour tous les boutons |
| setForeground(Color.cyan) | Couleur du texte (icône) → teinte Glow |
| setBackground(new Color(20, 20, 30)) | Couleur de fond sombre, contraste parfait pour Glow |
| setBorder(...) | Bordure cyan lumineuse de 2px, avec coins arrondis |
| MouseListener | Ajoute un **effet de survol** → devient plus clair au passage de la souris |



Prochain chapitre à suivre dans quelques instants : **4.4 — Minuteur + barre de progression (avec simulation du temps en Java)**

# Chapitre 4.4 — Barre de progression + minuterie simulée

## Rôle

Comme tu ne lis pas encore de fichier audio réel, on simule la durée d’une chanson avec :

* Une **barre de progression** (JProgressBar) qui avance progressivement
* Un **label à gauche** indiquant le temps écoulé (00:00)
* Un **label à droite** pour la durée totale (03:00 ou autre)
* Un **Timer** Java déclenché toutes les 1000 ms pour incrémenter et animer le tout

## Code complet

java

JProgressBar barre = new JProgressBar(0, 100);

barre.setPreferredSize(new Dimension(300, 10));

barre.setForeground(Color.CYAN); // Couleur de remplissage

barre.setUI(new BasicProgressBarUI()); // Supprime le style d'origine

barre.setBorder(BorderFactory.createLineBorder(Color.WHITE));

JLabel lblDebut = new JLabel("00:00");

JLabel lblFin = new JLabel("03:00");

## Simulateur de temps

java

Timer simulateur = new Timer(1000, new ActionListener() {

int secondes = 0;

public void actionPerformed(ActionEvent e) {

secondes++;

barre.setValue((secondes \* 100) / 90); // linéaire sur 90 secondes

lblDebut.setText(String.format("00:%02d", secondes));

if (secondes >= 90) {

simulateur.stop();

barre.setValue(0);

lblDebut.setText("00:00");

}

}

});

simulateur.start();

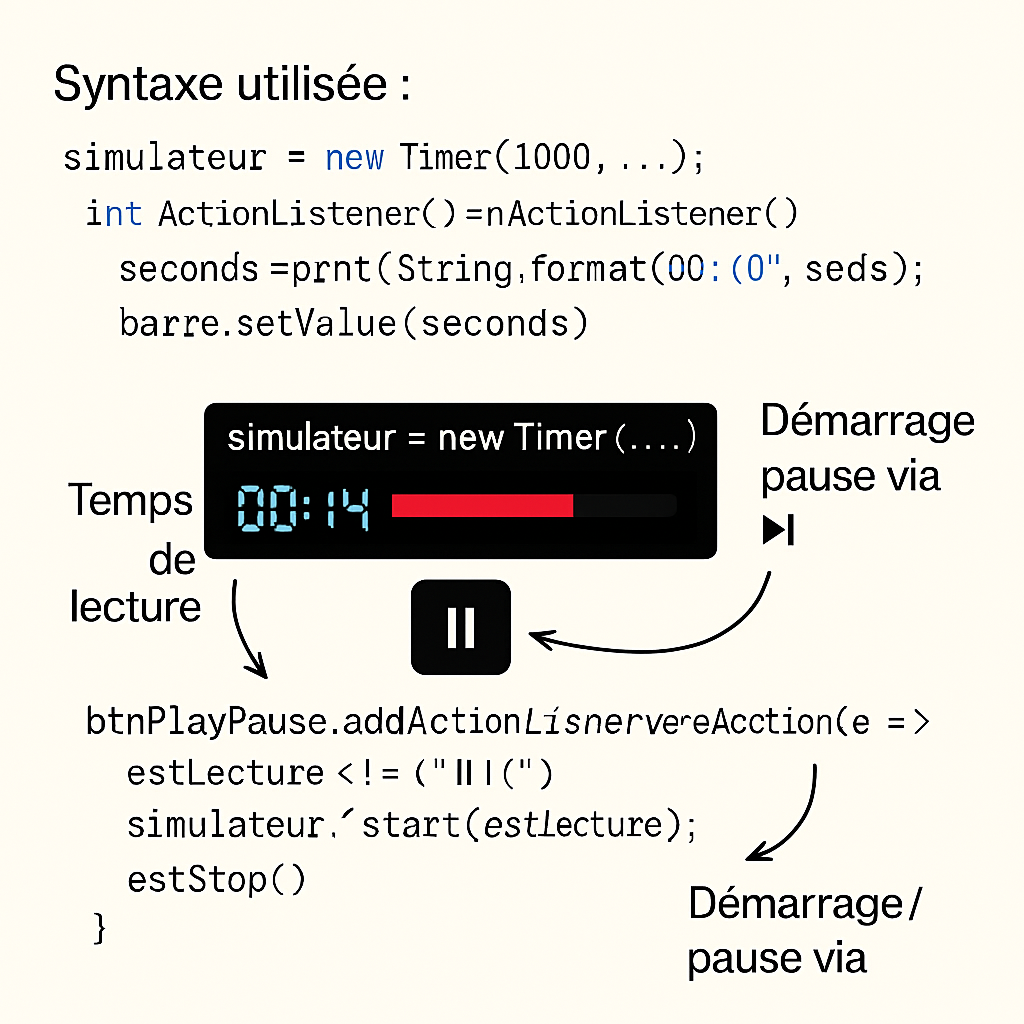
## Explication ligne par ligne

| **Élément Java** | **Rôle** |
| --- | --- |
| JProgressBar(0, 100) | Barre allant de 0 à 100 % |
| setPreferredSize(...) | Largeur & hauteur personnalisées |
| setForeground(Color.CYAN) | Couleur Glow de la barre |
| setUI(new BasicProgressBarUI()) | Empêche le look système par défaut |
| Timer(1000, ActionListener) | Exécute l'action chaque seconde (1000 ms) |
| secondes++ | Incrémente le temps simulé |
| setValue(...) | Actualise visuellement la progression |
| String.format(...) | Met en forme le temps écoulé (ex: "00:07") |

## Illustration à générer

Résultat :

* Simule le rythme d’écoute
* Ne dépend pas d’un vrai fichier audio
* Te permet d’ajouter un **effet réaliste et immersif** à l’interface



🪄 Prochaine étape déjà en cours de rédaction : **Chapitre 4.5 — Visualiseur animé avec** paintComponent()

# Chapitre 4.5 — Visualiseur animé (égaliseur graphique)

## Objectif

Donner une illusion de “son en mouvement” avec une animation graphique qui clignote et réagit pendant la lecture, même sans audio réel. C’est ce qui crée l’effet **“égaliseur lumineux**

Tu verras dans ton interface une série de **rectangles fins** qui changent de hauteur de façon aléatoire — simulant un battement ou une pulsation musicale.

## Code Java

java

JPanel visualiseur = new JPanel() {

protected void paintComponent(Graphics g) {

super.paintComponent(g);

Random r = new Random();

for (int i = 0; i < 28; i++) {

int h = r.nextInt(60);

g.setColor(i % 2 == 0 ? Color.RED : Color.GREEN);

g.fillRect(i \* 22, getHeight() - h, 8, h);

}

}

};

visualiseur.setPreferredSize(new Dimension(650, 100));

visualiseur.setBackground(new Color(10,10,20));

## Détail de chaque ligne

| **Ligne** | **Rôle** |
| --- | --- |
| new JPanel() {...} | Crée un panneau avec dessin personnalisé |
| paintComponent(Graphics g) | Redéfinit la méthode de dessin, appelée automatiquement |
| Random r = new Random() | Génère un nombre aléatoire à chaque passage |
| for (int i = 0; i < 28; i++) | Boucle pour créer 28 barres |
| int h = r.nextInt(60) | Détermine une hauteur aléatoire entre 0 et 60 pixels |
| setColor(...) | Alterne entre rouge et vert selon la position i |
| fillRect(x, y, largeur, hauteur) | Dessine une barre verticale à une position horizontale |
| setBackground(...) | Définit la couleur de fond du visualiseur |

java

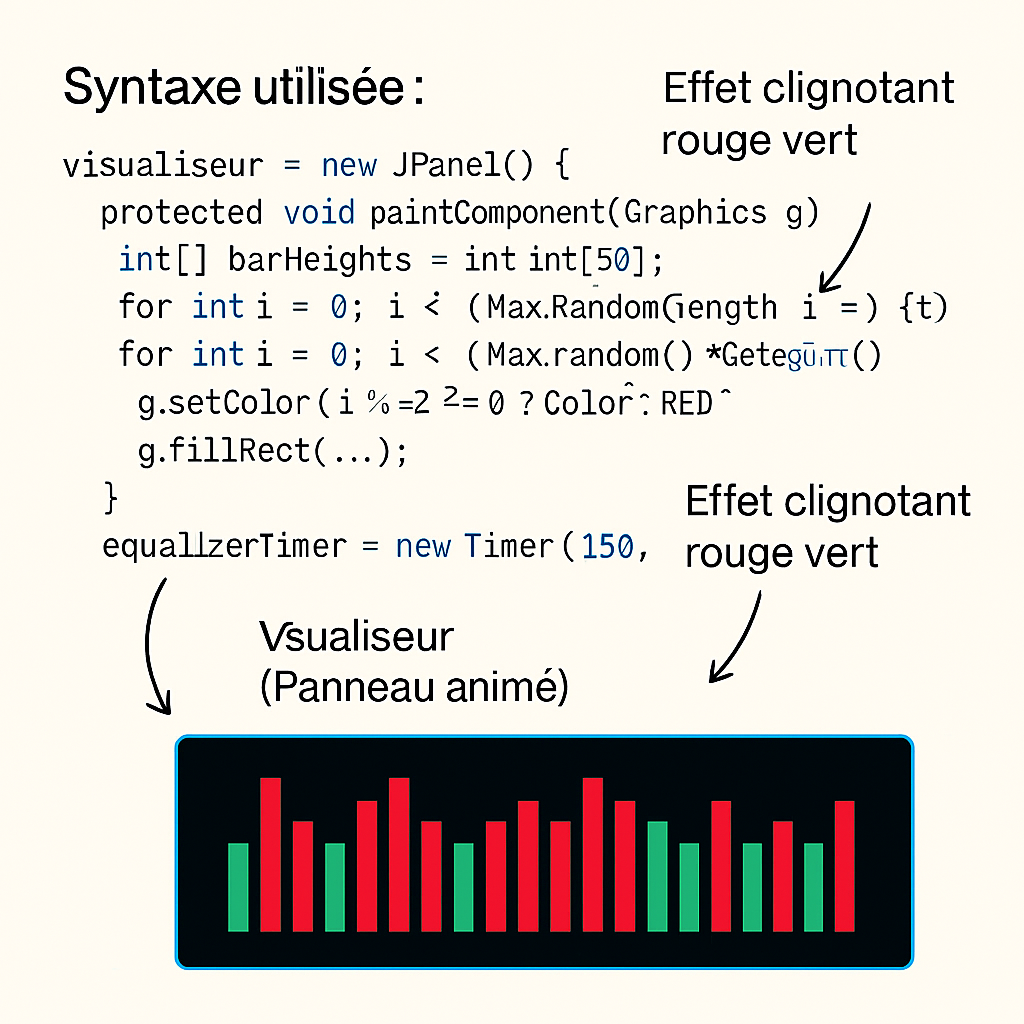
Timer equalizerTimer = new Timer(150, e -> {

if (estLecture) visualiseur.repaint();

});

equalizerTimer.start();

Ici, on déclenche un repaint() toutes les 150ms, ce qui redessine l’ensemble des barres avec des hauteurs différentes. Cela crée l’illusion d’un son qui évolue 🎶



L’interface devient **vivante et réactive**, même sans son réel. Et ton lecteur semble vibrer comme une vraie scène musicale.

# 4.6 — Thèmes personnalisables (Glow dynamique)

## Objectif Chapitre

Permettre à l’utilisateur de changer le **look entier de l’application** (couleurs du fond, boutons, bordures…) en un clic sur un bouton, pour créer une expérience visuelle **personnalisée et immersive**.

## Principe

* définir un **tableau de couleurs** : chaque élément du tableau représente un thème possible
* stocker un **index courant** (themeIndex) pour savoir quel thème est actif
* Quand on clique sur un bouton spécial (ex : “Changer de thème”), l’interface s’actualise

## Exemple de code

java

Color[] themes = {

new Color(40, 0, 60), // Violet foncé

new Color(0, 32, 96), // Bleu profond

new Color(10, 70, 10), // Vert froid

new Color(64, 0, 64) // Pourpre intense

};

int themeIndex = 0;

JButton btnChangerTheme = new JButton("🎨 Thème");

btnChangerTheme.addActionListener(e -> {

themeIndex = (themeIndex + 1) % themes.length;

mettreAJourThemes();

});

Et la méthode mettreAJourThemes() :

java

private void mettreAJourThemes() {

Color c = themes[themeIndex];

mainPanel.setBackground(c);

btnPlaylist.setBackground(c);

btnAlbums.setBackground(c);

btnDossiers.setBackground(c);

btnParametres.setBackground(c);

btnChangerTheme.setBackground(c);

}

## Explication mot par mot

| **Élément** | **Fonction** |
| --- | --- |
| Color[] themes | Tableau contenant 4 couleurs (chacune = un thème) |
| themeIndex = 0 | Commence au premier thème |
| (themeIndex + 1) % themes.length | Fait tourner les thèmes de façon cyclique |
| btn.addActionListener(...) | Déclenche une action quand on clique sur le bouton |
| mettreAJourThemes() | Applique la couleur active à tous les composants importants |

## Illustration

Visuel généré : 4 versions du lecteur Glow avec couleurs dominantes différentes : violet, bleu, vert, pourpre

Chaque style donne une **ambiance visuelle différente** — tu laisses l’utilisateur choisir son mood 🎛️

# Chapitre 4.7 — Organisation des panneaux et de l’affichage final

## Objectif

Tu veux une interface bien hiérarchisée qui regroupe :

* en haut : la navigation (Playlist, Albums, etc.)
* au centre : la playlist avec le titre en cours
* en bas : les boutons ⏮ ⏯ ⏭ + barre de progression + égaliseur animé

Pour cela, tu découpes ton interface en **panneaux Swing (**JPanel**)**, chacun avec ses responsabilités, puis tu les assembles dans mainPanel, puis dans ta fenêtre principale (JFrame).

## Exemple de structure

java

JPanel bandeauNavigation = new JPanel(null); // En haut

JPanel zonePlaylist = new JPanel(null); // Centre

JPanel barreEtBoutons = new JPanel(new FlowLayout()); // Bas (⏮ ⏯ ⏭)

JPanel visualiseur = new JPanel() { ... }; // Dessin animé

mainPanel = new JPanel(null);

mainPanel.setBackground(themes[themeIndex]);

mainPanel.setLayout(null);

Tu places chaque JPanel avec setBounds(...) pour une position précise, puis tu ajoutes tout au mainPanel :

java

mainPanel.add(bandeauNavigation);

mainPanel.add(zonePlaylist);

mainPanel.add(barreEtBoutons);

mainPanel.add(visualiseur);

Et enfin, dans la fenêtre Swing :

java

JFrame fenetre = new JFrame("JeanGlow Music Player");

fenetre.setContentPane(mainPanel);

fenetre.setSize(820, 580);

fenetre.setLocationRelativeTo(null);

fenetre.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

fenetre.setVisible(true);

## Explication ligne par ligne

| **Élément** | **Rôle** |
| --- | --- |
| new JPanel(null) | Positionnement **manuel** avec setBounds() (layout = null) |
| new FlowLayout() | Aligne les boutons ⏮ ⏯ ⏭ horizontalement |
| setBackground(themes[themeIndex]) | Applique la couleur du thème à tout le panneau principal |
| fenetre.setLocationRelativeTo(null) | Centre la fenêtre sur l’écran |

**Chapitre 4.8 — Vue globale du lecteur Glow final**

**Illustration**

*Image générée montrant l’ensemble du lecteur Glow dans sa version finale :*

* Navigation en haut (avec icônes)
* Titre central affiché
* Playlist à gauche, en liste verticale néon
* À droite : visualiseur animé en barres
* En bas : ⏮ ⏯ ⏭ + barre de progression + minuteur

Ce visuel est idéal pour ta **page de couverture** dans ton PDF ou comme poster pédagogique du projet.

**Résumé technique de l’interface Swing**

| **Composant Swing** | **Fonction** | **Personnalisation Glow** |
| --- | --- | --- |
| JButton | Navigation + Play/Pause | Couleurs dynamiques, bordures Glow, Unicode |
| JList | Playlist | Fond sombre + sélection néon |
| JLabel | Titre chanson, minuteur | Texte coloré (Magenta, Blanc…) |
| JProgressBar | Avancement visuel du temps | Couleur cyan Glow, barre stylisée |
| JPanel + paintComponent() | Visualiseur animé | Barres rouges et vertes en animation |
| Timer | Animation + simulation du temps | Tick toutes les X ms pour recréer le rythme |

**Lexique pédagogique**

| **Terme Java Swing** | **Signification simple** |
| --- | --- |
| JFrame | Fenêtre principale de l’application |
| JPanel | Conteneur pour organiser les éléments |
| setBounds(x, y, w, h) | Position absolue d’un composant (coordonnées) |
| setBackground() | Change la couleur de fond |
| addActionListener() | Ajoute une action au clic d’un bouton |
| paintComponent() | Permet de dessiner manuellement des formes |
| Timer | Répète une tâche à intervalles réguliers |

**Défis bonus — Pour aller plus loin**

Tu peux ajouter à l’avenir :

* Lecture réelle de .mp3 avec JLayer (voir jl1.0.1.jar)
* Thèmes personnalisables avec ColorChooser
* Reconnaissance vocale pour sélectionner des morceaux
* Visualisation synchronisée à l’audio réel avec FFT (Fast Fourier Transform)
* Animation de la fenêtre en cas de basses (Shake effect )