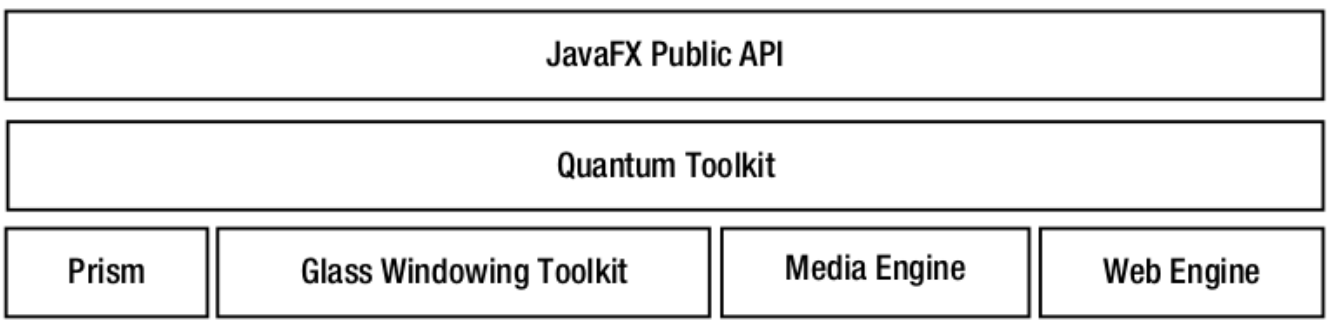
Tartalomjegyzék

Mi is az a JavaFX?

- Java GUI-s alkalmazásokhoz egy ideális választás
- Tekinthető a Swing leszármazottjának
- GUI építés kétféle módon
  - Java kód írásával
  - FXML leíró segítségével (XML alapú)

A JavaFX komponensei



A GUI felépítéséhez egy úgynevezett **Scene Graph** hozunk létre. Ez a gráf vizuális elemeket tartalmaz, amiket **Node**-oknak nevezünk (`javafx.scene.Node`). Ezeket a `Node``-okat a Scene Graph **hierarchikus** elrendezésben tárolja. Fontos, hogy ez a hierarchia egy fát alkot, azaz nincs benne kör. Ez azért tilos, mert például ha két tároló egymást tárolná direkt vagy indirekt az értelmetlen lenne. Továbbá a gráfhoz minden elemet csak egyszer adhatunk hozzá. A Scene Graph-ot a **publikus JavaFX API** segítségével tudjuk felépíteni. Egy node lehet például egy egyszerű UI vezérlő, például egy gomb.

A **Prism** egy hardver-gyorsított grafikus csővezeték, ami rendereli a Scene Graph-ot.

A **Glass Windowing Toolkit** az operációs rendszertől függően natív módon ablakozási feladatokat lát el. Ezen felül ez a komponens felelős az eseménykezelő sorok (event queue) kezeléséért. JavaFX-ben az események a fő threadben kerülnek kezelésre, amely JavaFX-ben az úgynevezett **Application Thread**. Fontos, hogy a Scene Graph-ot csak ezen a fő szálon keresztül módosíthatjuk.

A **Media Engine** meglepő módon a média lejátszához ad segítséget. Audio és Videó lejátszási lehetőségeket biztosít a felhasználó számára.

JavaFX-es alkalmazásainkban webes tartalmat is megjeleníthetünk, melynek felelőse a **web engine** (WebKit alapú).

A **Quantum toolkit** a 4 alacsony szint felett egy absztrakciós szint, illetve a 4 alacsony szint közötti koordinációért is ő a felelős.

Telepítés

Többféleképpen konfigolhatjuk a rendszerünket, hogy a JavaFX-et működésre bírjuk. Mivel a 8-as Oracle JDK-ban benne van a JavaFX fejlesztéshez szükséges összes eszköz, ezért talán az a legegyszerűbb, így elsősorban ezt fogom bemutatni. A gyakorlaton az Eclipse IDE-t használjuk, így ennek belövését írom le itt részletesen

Közös előkészületek

Bármelyik IDE-t is választod mindenképpen legyen 8-as Java a gépen, illetve legyen beállítva a JAVA_HOME, illetve legyen hozzáadva a PATH változóhoz a JDK bin könyvtára. Ha 8-as OpenJDK-d van akkor abban sajnos nincs benne a jfxrt (java fx runtime), szóval oda kell még egy ilyen

- `sudo apt-get install openjfx`

JavaFX - Eclipse

Amennyiben a JavaFx-et Eclipse-ből szeretnéd használni akkor kövesd a következő utasításokat:

- Töltsd le a legfrissebb Eclipse-et
- Help/Install new software: http://download.eclipse.org/efxclipse/updates-nightly/site -> e(fx)clipse install telepítése
- Ha ment az eclipse akkor restart és utána már a jfxrt-nek is benne kell lennie JRE System Library-k között
- SceneBuilder: https://docs.oracle.com/javafx/scenebuilder/1/use_java_ides/sb-with-eclipse.htm

JavaFX - IntelliJ

- Csináljunk egy új projektet: File -> New -> Project
- Válasszuk ki, hogy JavaFX alkalmazást szeretnénk

Amennyiben a Közös előkészületek szekcióban leírtaknak megfelelően van openjdk + openjfx telepítve a gépedre, akkor nincs is más teendőd, mint elnevezni a projektet és kész. A JDK verzióra ügyelj a projekt létrehozásakor, mert különben nem fog működni.

Java 9 és utána

A JavaFX saját útra tért, így kicsit másképpen kell összelőni a rendszert. Mivel a hivatalos honlap bő leírást ad, ezért itt külön nem írom le a lépéseket.

https://openjfx.io/openjfx-docs/

Hello World JavaFX-ben

Csontváz

Minden JavaFX alkalmazasnak az Application osztályból kell származnia, ami a javafx.application csomagban található.

Tehát minden alkalmazás fő osztálya valahogy így néz ki:

```
package hu.alkfejl;
import javafx.application.Application;

public class HelloFX extends Application {
  // ide jön a további kód
}
```

FONTOS: Ez a kód ilyen formában még nem lesz futtatható. Az Application osztály egy absztrakt osztály, melynek van egy `start(Stage stage)` absztrakt metódusa. Az Application osztályban a következő módon van deklarálva a metódus:

```
public abstract void start(Stage stage) throws java.lang.Exception
```

Ezt nekünk kell kifejtenünk. Ez lesz a fő belépési pont. Ezzel bővítve a kódot a következőt kapjuk:

```
package hu.alkfejl;
import javafx.application.Application;
import javafx.stage.Stage;

public class HelloFX extends Application {
```

```
    @Override
    public void start(Stage stage) {
      // Ide jön, hogy induláskor mi történjen
    }
}
```

A start metódus a JavaFX alkalmazás elindításakor fog meghívódni. Vegyük észre, hogy paraméterként egy `Stage`-et kap ez a metódus. Ez lesz az úgynevezett **primary stage**. A stage-re tekintsünk úgy, mint egy komplett ablakra (címsor, minimize, maximize, bezárás gombokkal együtt, és maga az ablak tartalma). Az ablak tartalma egy scene lesz, amit majd kicsit később hozzá is fogunk adni. A primary stage mindig létrejön, de ezen felül majd mi magunk is hozhatunk létre további stage-eket (ablakokat). A fenti alkalmazást már le tudjuk fodítani, de történni nem fog semmi. Nézzük meg mi kell ahhoz, hogy valami láthatót is csináljunk.

Tehát a primary stage-ünk tekinthető egy scene tárolójának. A JavaFX által létrehozott primary stage-nek alapból nincs megadva scene-je, azaz üres ez a konténer (az ablakon belül nincs semmi tartalom). Később készítünk egy saját scene-t amit megadunk majd a primary stage-nek. Egyelőre állítsuk be a primary stage címsorát, hogy mutassa a 'Hello JavaFX' szöveget. Ahhoz, hogy a stage-et kirajzolja (renderelje a rendszer) meg kell hívnunk a `show()` metódusát.

```
package hu.alkfejl;
import javafx.application.Application;
import javafx.stage.Stage;

public class HelloFX extends Application {

    @Override
    public void start(Stage stage) {
      stage.setTitle("Hello JavaFX");

      stage.show();
    }
}
```

A fenti kód teljesen jól működik, ám a szemfülesek észrevehetik, hogy nincsen main metódusunk. JavaFX alkalmazés esetében nem is kell, hogy legyen main metódusunk. Ha ez valamiért mégis szükséges számunkra (például a parancssori argumentumok miatt) akkor módosítsuk a programunkat a következőképpen.

```
package hu.alkfejl;
import javafx.application.Application;
import javafx.stage.Stage;

public class HelloFX extends Application {

    public static void main(String[] args) {
      Application.launch(args);
    }
```

```
    @Override
    public void start(Stage stage) {
      stage.setTitle("Hello JavaFX");

      stage.show();
    }
}
```

Az Application statikus launch metódusa csinál néhány munkálatot a háttérben, majd meghívja a start metódust. Ilyen módon jóformán becsomagoltuk a JavaFX alkalmazásunkat.

FONTOS: Néhány IDE esetén problémát okozhat a main nélküli osztály, így mi mindig használni fogjuk.

```
TIPP: A launch metódus nem tér addig vissza, amedding minden ablakot be nem
zárunk, vagy meg nem hívjuk a Platform.exit() metódust.
```

Scene hozzáadása

Ahogy azt említettük korábban a primary stage-ünk tartalmazhat egy `Scene`-t, ami pedig a grafikai elemeket tartalmazza egy fa-struktúrában. Ennek a fa hierarchiának a legfelső pontja az úgynevezett **root node**. Egy `Scene`-nek rendelkeznie kell egy root node-al. A következő példában VBox-ot használunk majd root-ként.

```
TIPP: Bármelyik elem ami a javafx.scene.Parent osztályból származik
használható root node-ként. Tipikusan a tárolók és elrendezés panelek ilyen
elemek. Például: VBox, HBox, Pane, GridPane, TitlePane, FlowPane.
```

Íme egy példa, amiben már scene-t is adunk a stage-hez:

```
package hu.alkfejl;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class HelloFX extends Application {

    public static void main(String[] args) {
      Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
      stage.setTitle("Hello JavaFX");

      VBox root = new VBox();
      Text msg = new Text("Hello JavaFX from the Scene");
```

```
        root.getChildren().add(msg);

        Scene scene = new Scene(root, 300, 50);

        stage.setScene(scene);
        stage.show();
    }
}
```

Vegyük sorra az új kódot.

Ahhoz, hogy a scene root node-ját be tudjuk állítani létrehozunk egy VBox objektumot, amely egy vertikális elrendezést biztosító tároló. Ez azt jelenti, hogy a VBox-hoz hozzáadott gyermek node-ok egymás alatt fognak megjelenni.

```
VBox root = new VBox();
```

Ehhez a tárolóhoz most egyetlen gyereket adunk hozzá, mégpedig egy Text típusú objektumot.

```
Text msg = new Text("Hello JavaFX from the Scene");
```

Ez egy egyszerű szöveges tartalmat definiál JavaFX-ben. A Text osztály azon konstruktorát használtuk, melynek egy String-et adhatunk meg.

Ezután ténylegesen hozzá is adjuk a létrehozott Text objektumot a tárolóhoz:

```
root.getChildren().add(msg);
```

A `getChildren()` metódussal lekérhető az adott tároló összes gyereke, ami egy `ObservableList<Node>` objektumot ad vissza, azaz a gyerekeket egy listában érhetjük el. A fenti példában egyszerűen ehhez a listához adjuk hozzá az új elemet, azaz a Text típusú msg objektumot.

```
TIPP: Minden Parent-ből származó elemnek van getChildren() metódusa, mivel
a Parent-nek lehetnek gyerekei...
```

Az ObservableList egy olyan lista interface, mely biztosítja azt, hogy a lista változásakor a feliratkozók értesítést kapjanak. Ez azért fontos, mert ha futás közben dinamikusan változtatjuk egy Parent gyerekeit (pl. hozzáadunk egy új gyereket), akkor valószínűleg újra ki kell rajzolnia a rendszernek.

Miután elkészítettük a root node-unkat és adtunk hozzá tartalmat, létrehozzuk a scene-t:

```
Scene scene = new Scene(root, 300, 50);
```

A Scene több konstruktorral is rendelkezik, de a root node-ot meg kell adnunk mindenképpen (ami Parent típusú kell legyen). A példában megadjuk a scene méretét is (szélesség és magasság).

Ezután a primary stage-nek meg kell mondanunk, hogy az imént létrehozott scene-t használja, így a következő utasítást is megadjuk:

```
stage.setScene(scene);
```

A Hello JavaFX bővítése

Még nagyon keveset láttunk a JavaFX-ből. A következő lépés, hogy hozzáadunk egy gombot, amit ha megnyom a felhasználó, akkor az alkalmazás kilép.

Amikor a felhasználó megnyomja a gombot, akkor egy `ActionEvent` esemény keletkezik. Amennyiben kezelni szeretnénk a keletkező ActionEvent-et, akkor hozzá kell adnunk egy ActionEvent handler-t (kezelőt), mert a gomb `setOnAction` metódusa ezt várja, egészen pontosan egy `EventHandler<ActionEvent>` paramétert. A saját eseménykezelőhöz ezért a `EventHandler<ActionEvent>` interfészből kell származtatni egy saját osztályt, és a `void handle(ActionEvent e)` metódust kell megvalósítani, ami az esemény bekövetkeztekor fog meghívódni. Erre egy példa a következő:

```
class MyEventHandler implements EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent e) {
        Platform.exit();
    }
}
```

És a használata:

```
Button exit = new Button("Exit");
exit.setOnAction(new MyEventHandler());
```

Mivel a különböző eseményekhez különböző eseménykezelőt használunk, és általában egy eseménykezelő osztályt csak egy helyen használunk, ezért ,,felesleges" az osztályt létrehozni, ezt megvalósíthatjuk anonymous osztállyal is.

```
Button exit = new Button("Exit");
exit.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent e) {
      Platform.exit();
    }
```

```
});
```

Tovább egyszerűesítve a dolgot használhatunk lambda kifejezést is.

```
Button exit = new Button("Exit");
exit.setOnAction(e -> {
    Platform.exit();
});
```

A gombon kívül létrehozhatunk egy TextField-et is, amely egy egyszerű szöveges beviteli mező. A TextField-hez hozzáadunk egy eseménykezelőt, mely figyeli a billentyű felengedéseket és a korábban létrehozott Text típusú msg szövegét változtatja meg.

```
Text msg = new Text();
TextField name = new TextField();

name.setOnKeyReleased(e -> {
    msg.setText("Hi " + name.getText());
});
```

Az új részek után a teljes program valahogy így néz ki:

```
package hu.alkfejl;
import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class HelloFX extends Application {

    public static void main(String[] args) {
      Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
      stage.setTitle("Hello JavaFX");

      VBox root = new VBox();
      Text msg = new Text();
      TextField name = new TextField();
```

```
        Button exit = new Button("Exit");
        exit.setOnAction(e -> {
          Platform.exit();
        });

        name.setOnKeyReleased(e -> {
          msg.setText("Hi " + name.getText());
        });

        root.getChildren().addAll(name, msg, exit);

        Scene scene = new Scene(root, 500, 300);
        stage.setScene(scene);
        stage.show();
    }
}
```

Egy JavaFX Application életciklusa

Két fő szál jön létre egy JavaFX alkalmazás futtatásakor (a `launch()` metódus hatására):

- JavaFX Launcher
- JavaFX Application Thread

A JavaFX Runtime az `Application` osztály következő metódusait hívja meg annak életciklusa folyamán (sorrendben):

- Paraméter nélküli konstruktor (Ilyennek lennie kell)
- `init()`
- `start()`
- `stop()`

A JavaFX Application Thread példányosít egy objektumot az Application leszármazottjából a `launch()` hívás hatására. Ezután meghívódik az `init()` metódus, ami az Application osztályban egy üres metódus, de kedvünkre felüldefiniálhatjuk a fő osztályunkban. Stage és Scene létrehozása az `init`-en belül nem lehetséges. Ezután meghívódik a `start()` metódus és a `launch` metódus a futás befejezésére vár. Amikor a futását befejezte az app, akkor meghívódik a `stop()` metódus, ami ugyanúgy egy üres metódus az `Application` osztályban, de felüldefiniálhatjuk. A fentiek közül az init a Launcher Thread által hívott, a többi az App Thread által.

Scene-ek menedzselése

Jelen fejezet megmutatja, hogy hogyan tudunk több `Scene` között váltani. Ez azért fontos, mert egy komplexebb alkalmazásban biztosan több Scene-t fogunk használni az egyes tartalmak megjelenítéséhez.

Például: Chat alkalmazás

- Partner választó ablak
- Beszélgetés a partnerrel

A következő példa megmutatja, hogy hogyan tudunk két egyszerű scene-t létrehozni és közöttük navigálni.

```
package hu.alkfejl;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;


public class Main extends Application {
        private Stage mainWindow;
        private Scene scene1, scene2;

        @Override
        public void start(Stage primaryStage) {
                mainWindow = primaryStage;

                constructScene1();
                constructScene2();

                mainWindow.setScene(scene1);
                mainWindow.setTitle("Multiple scenes");
                mainWindow.show();

        }

        private void constructScene1() {
                Label lb = new Label("Scene 1");
                Button btn = new Button("Go to Scene 2");
                btn.setOnAction(e -> {
                        mainWindow.setScene(scene2);
                });

                //Some layout
                VBox root = new VBox();
                root.getChildren().addAll(lb, btn);

                scene1 = new Scene(root, 300, 300);
        }

        private void constructScene2() {
                Label lb = new Label("Scene 2");
                Button btn = new Button("Go to Scene 1");
                btn.setOnAction(e -> {
                        mainWindow.setScene(scene1);
                });

                //Some layout
                VBox root = new VBox();
                root.getChildren().addAll(lb, btn);
```

```
                scene2 = new Scene(root, 300, 300);
        }

        public static void main(String[] args) {
                launch(args);
        }
}
```

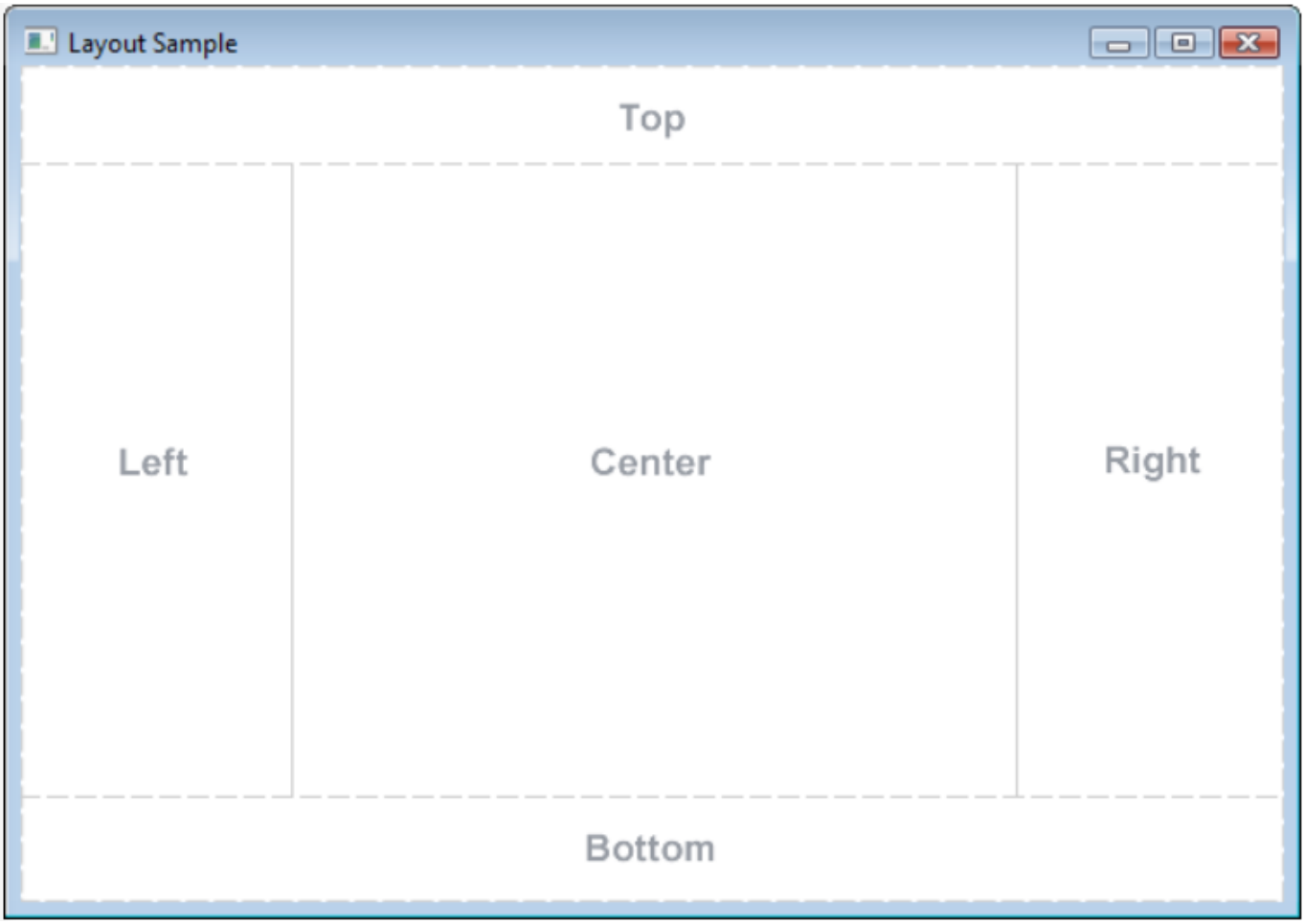
A `start()` metódus elején eltároljuk egy adattagban a primaryStage-re a referenciát, hogy ezen metóduson kívül is elérhessük a fő ablakunkat. Ezután egyszerűen megkonstruáljuk a két `Scene`-t, amelyekre egy-egy `Label` (egyszerű szöveges UI elem), illetve egy-egy `Button` kerül. A gombokhoz hozzáadtunk egy-egy eseménykezelőt is, melyek rendre a másik `Scene`-t állítják be a `mainWindow`-nak, mint aktuális Scene. Komplexebb alkalmazásoknál érdemes lehet készítenünk egy `SceneManager` osztályt, melynek segítségével végezzük el az összes ilyen tevékenységet.

Elrendezések

JavaFX-ben a UI elemeknek rendre megmondhatjuk, hogy milyen pozícióban jelenjenek meg. Ennél egy jobb megoldás lehet, ha layout-okat használunk, mivel átméretezéskor a rendszer automatikus méretezi és pozicionálja a layout által tartalmazott UI elemeket is. A beépített elrendezések mind a `javafx.scene.layout.Pane` osztályból származnak (ezért nevük általában ...Pane-re végződik).

BorderPane

A `BorderPane` az alábbi képen látható elrendezést biztosít:



A `BorderPane` a legtöbb alkalmazásban előfordul, mivel remek lehetőséget biztosít a felsőbb szintű komponensek elrendezéséhez. Pl: Fenti toolbar, baloldali navigációs nézet, alulra valamilyen status bar kerülhet, középen a fő tartalom jeleníthető meg.

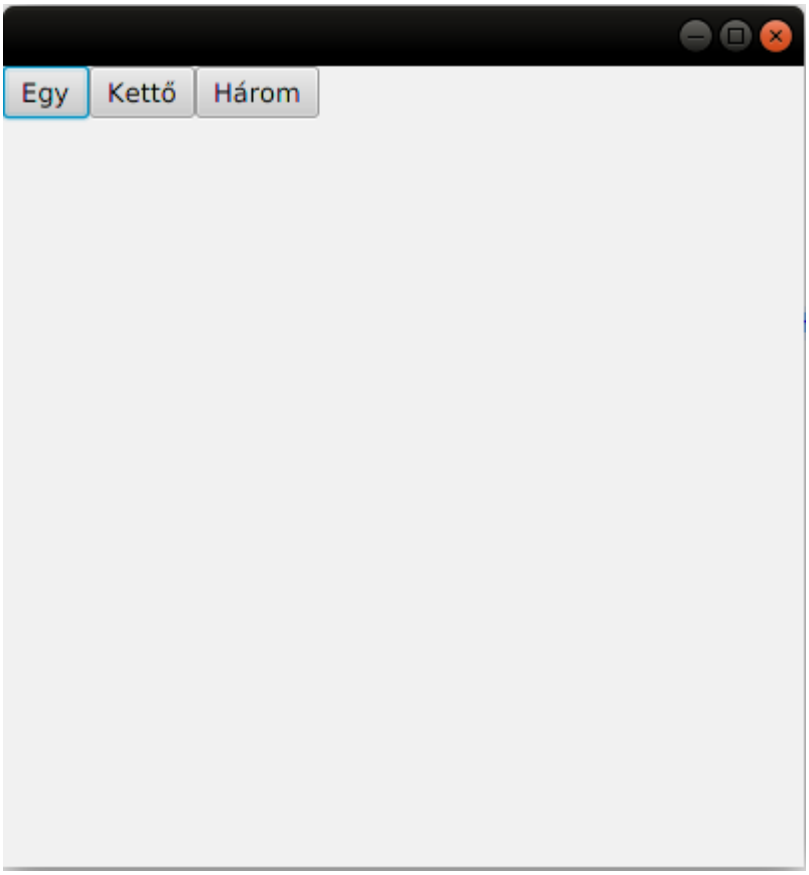
A különböző régiókat akárhogyan méretezhetjük. Ha például nincs szükség a jobboldali részre, akkor annak mérete lehet 0 is. Ehhez egyszerűen nem kell definiálni azt a régiót, amire nincs szükség.

```
@Override
        public void start(Stage primaryStage) {

                BorderPane root = new BorderPane();
                root.setTop(new Label("Fent"));
                root.setLeft(new Label("Bal"));
                root.setRight(new Label("Jobb"));
                root.setBottom(new Label("Lent"));
                root.setCenter(new Label("Közép"));

                Scene scene = new Scene(root, 300, 300);

                primaryStage.setScene(scene);
                primaryStage.setTitle("BorderPane demo");
                primaryStage.show();

        }
```

HBox

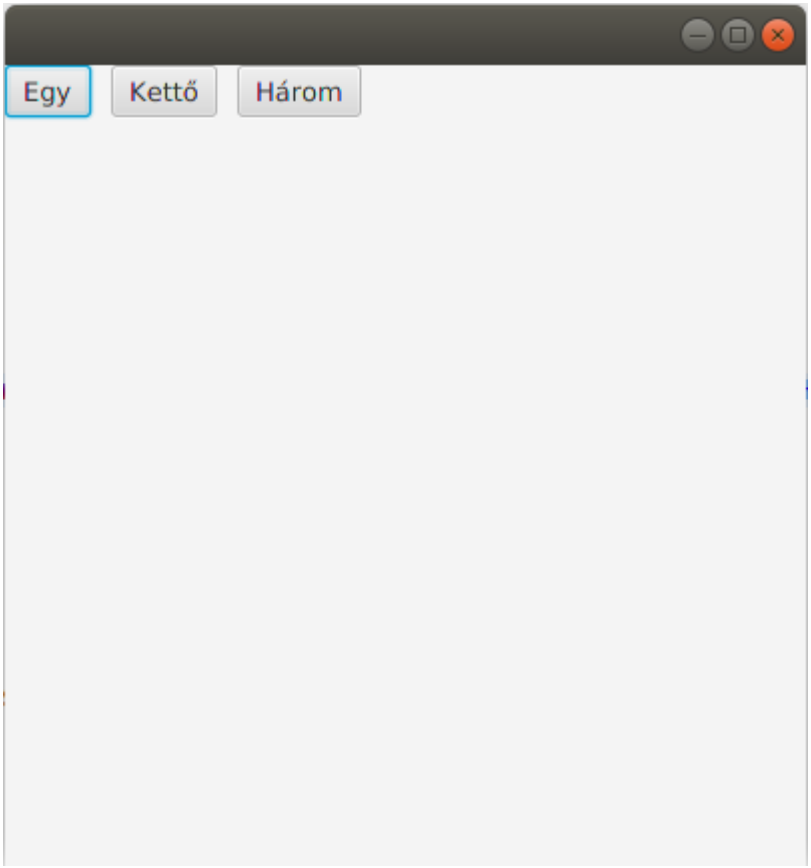
A `HBox` egy egyszerű horizontális elrendezést tesz lehetővé, ahol az elemek egymás mellé kerülnek. A `HBox`-on belül az elemek közötti távolságot a `Spacing`-el adhatjuk meg.

```
HBox root = new HBox();
root.getChildren().addAll(new Button("Egy"), new Button("Kettő"), new
Button("Három"));
```



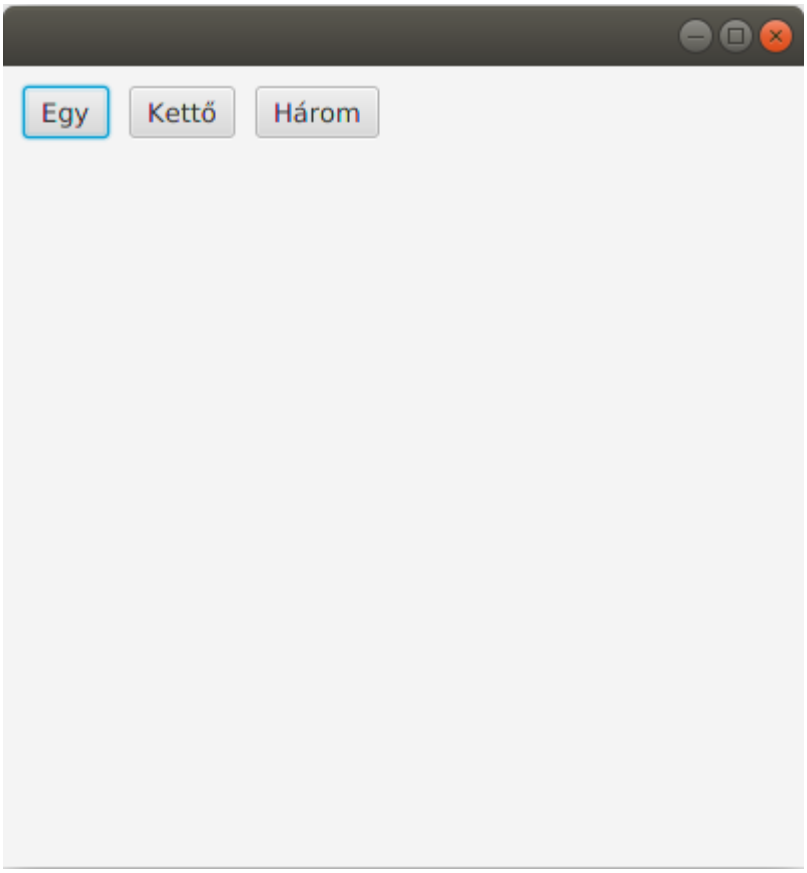
Ugyanez 10-es spacing-el:

```
HBox root = new HBox();
root.setSpacing(10);
root.getChildren().addAll(new Button("Egy"), new Button("Kettő"), new
Button("Három"));
```



Ha padding-et is alkalmazunk rá:

```
root.setPadding(new Insets(10,10,10,10));
```

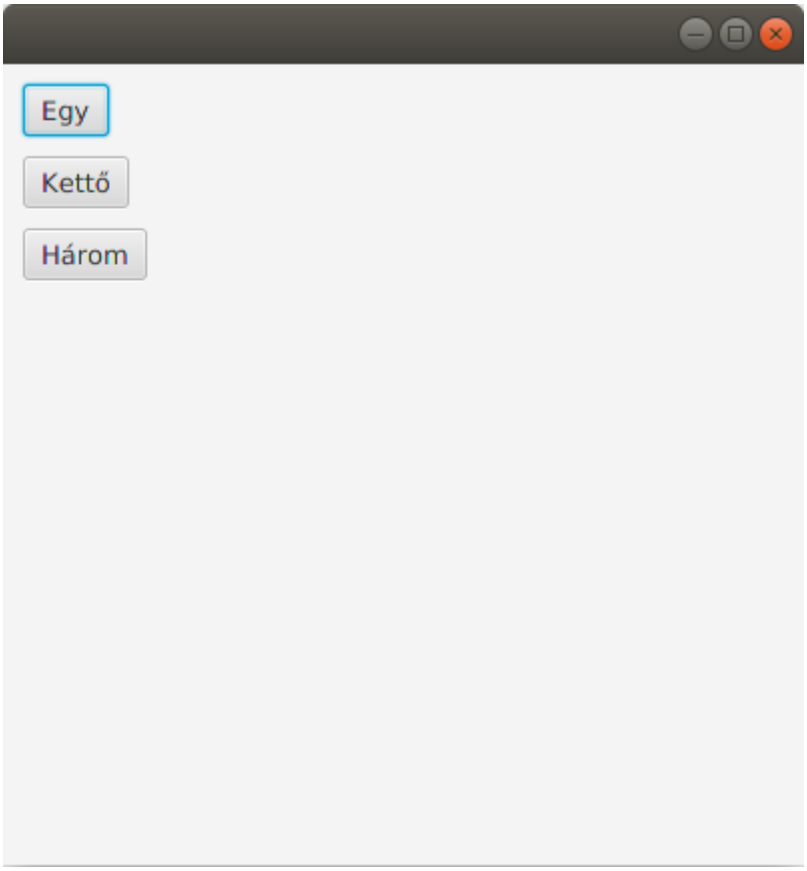


VBox

A `VBox` egy egyszerű vertikális elrendezést tesz lehetővé, ahol az elemek egymás alá kerülnek. A `VBox`-on belül az elemek közötti távolságot a `Spacing`-el adhatjuk meg.

A példakód már spacing és padding alkalmazását is mutatja.

```
  @Override
  public void start(Stage primaryStage) {
    try {
      VBox root = new VBox();
      root.setSpacing(10);
      root.setPadding(new Insets(10, 10, 10, 10));
      root.getChildren().addAll(new Button("Egy"), new Button("Kettő"), new
Button("Három"));
      Scene scene = new Scene(root, 400, 400);
      primaryStage.setScene(scene);
      primaryStage.show();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```



StackPane

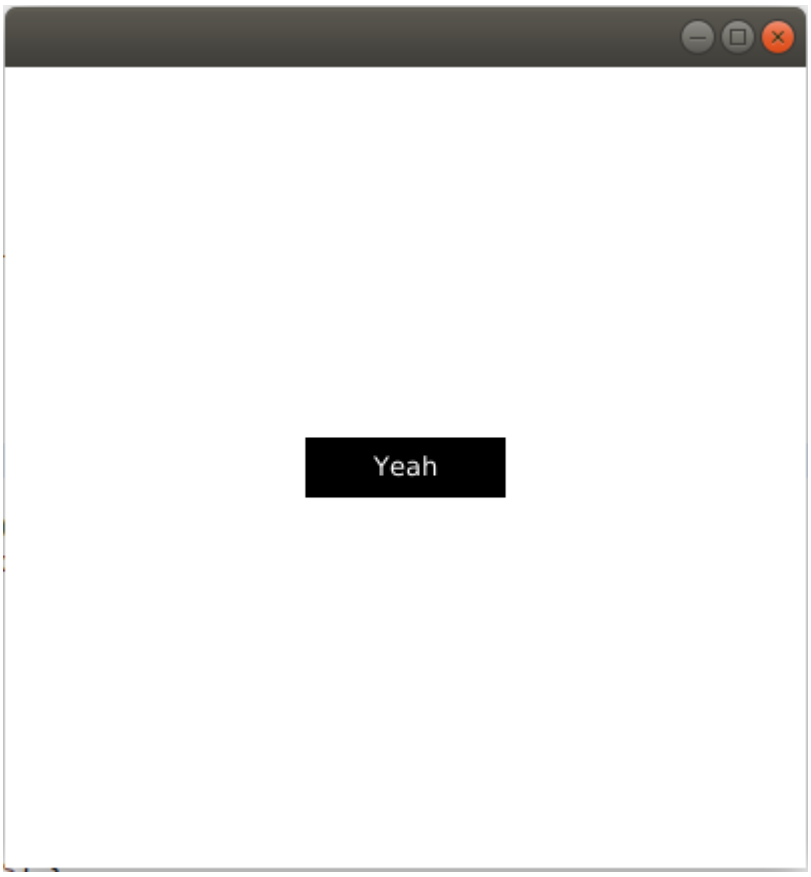
A `StackPane`, ahogy neve is mutatja, egy stack-et használ. A hozzáadott elemek egy stack-be kerülnek és így egymáson jelennek meg. Jól használható akkor amikor például egy képre szeretnénk szöveget írni.

Amennyiben megváltoztatjuk az elemek pozicionálását, akkor az az összes belerakott elemre hatással lesz.

Lássunk egy példát:

```
@Override
public void start(Stage primaryStage) {
  try {
    StackPane root = new StackPane();
    Rectangle rect = new Rectangle(100.0, 30.0);

    Text text = new Text("Yeah");
    text.setFill(Color.WHITE);

    root.getChildren().addAll(rect, text);
    Scene scene = new Scene(root, 400, 400);
    primaryStage.setScene(scene);
    primaryStage.show();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Ennek az eredménye a következő:



A példában a `javafx.scene.shape.Rectangle` osztályt használtuk egy téglalap létrehozásához. Az alap szín fekete, így ezt ezzel rajzolja ki a rendszer. Ezután létrehoztunk egy `Text` típusú objektumot, melynek a színét (`setFill`) fehérre állítottuk be.

TIPP: A StackPane alapértelmezetten középre igazítja a benne lévő elemeket.

GridPane

Rácsos elrendezést lehetővé tevő elrendezés manager. A cellatartalmak `Node` típust esznek meg. Az elrendezést kedvünkre alakíthatjuk a span-ek (sor vagy oszlop összevonások) használatával.

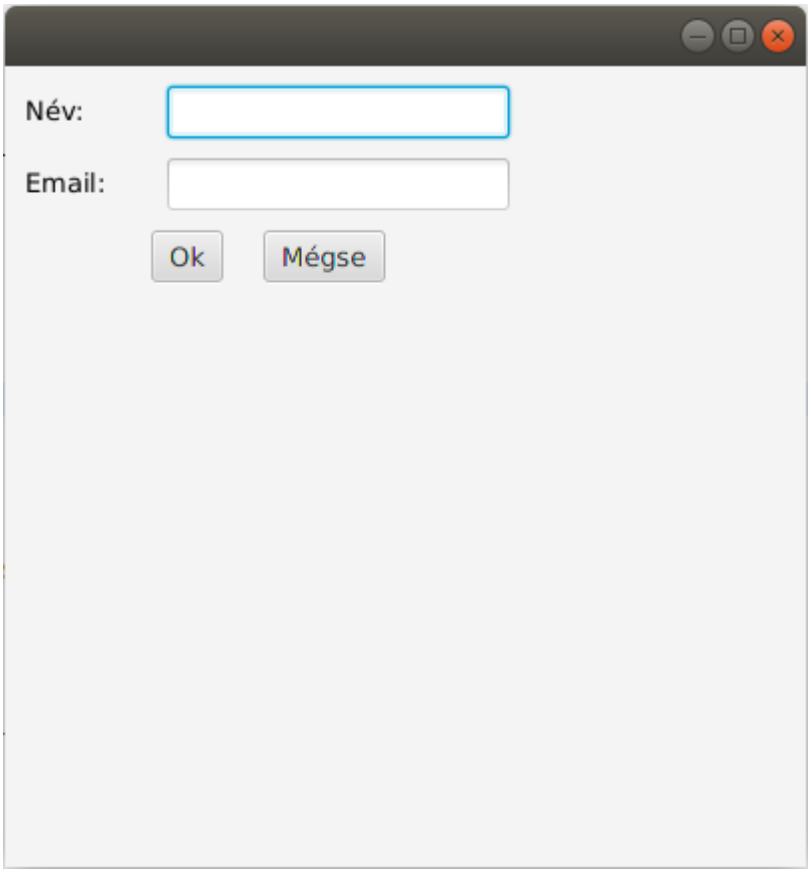
TIPP: Jól használható például formok létrehozásakor.

A cellák közötti helyet a HGap és VGap adja meg.

Egy egyszerű form összerakása valahogy így néz ki:

```
@Override
  public void start(Stage primaryStage) {
    try {
      GridPane root = new GridPane();
      root.setVgap(10); //függőleges helyköz a cellák között
      root.setHgap(30); //vízszintes helyköz a cellák között

      //első sor hozzáadása
      Text name = new Text("Név:");
      root.add(name, 0, 0); //add(Node child, int columnIndex, int
rowIndex)
      TextField nameTextField = new TextField();
      root.add(nameTextField, 1, 0);

      //második sor hozzáadása
      Text email = new Text("Email:");
      root.add(email, 0, 1);
      TextField emailTextField = new TextField();
      root.add(emailTextField, 1, 1);

      // A két gombot belerakjuk egy HBox-ba
      Button ok = new Button("Ok");
      Button cancel = new Button("Mégse");
      HBox hb = new HBox();
      hb.setAlignment(Pos.CENTER); // A hbox-on belül középre igazítottak
az elemek
      hb.setSpacing(20); // az elemek közötti távolság
      hb.getChildren().addAll(ok, cancel); // gombok a hbox-hoz
      root.add(hb, 0, 2, 2, 1); //hbox a grid-hez -> add(Node child, int
columnIndex, int rowIndex, int colspan, int rowspan)

      root.setPadding(new Insets(10, 10, 10, 10)); // gridre egy padding
      Scene scene = new Scene(root, 400, 400);
      primaryStage.setScene(scene);
      primaryStage.show();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Ennek eredménye:



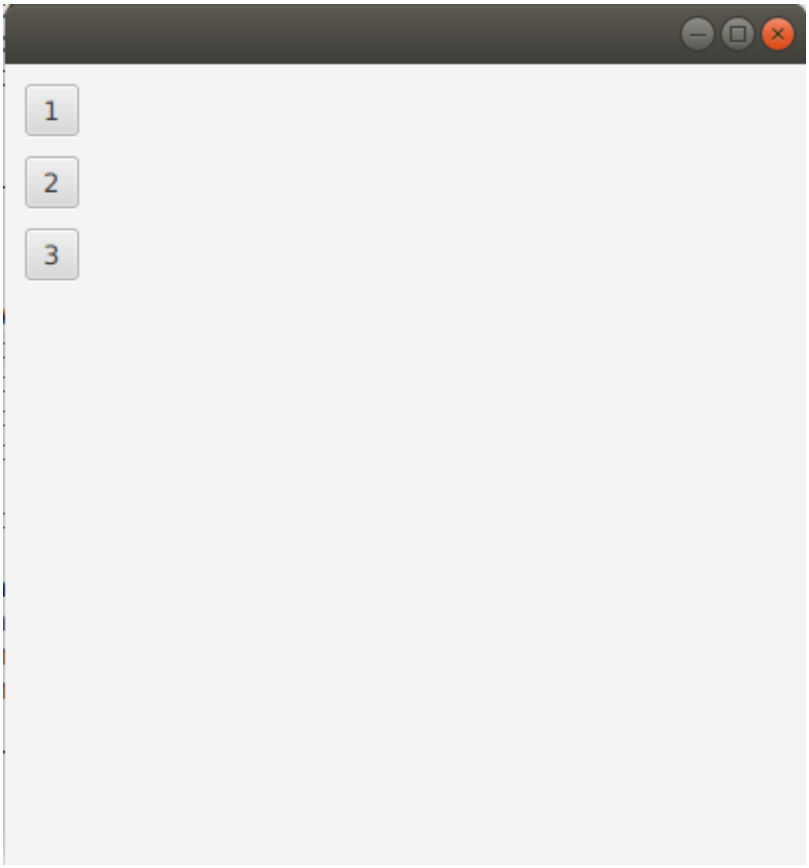
FlowPane

A `FlowPane` az elemek egymás után helyezi el. Alapértelmezetten horizontálisan rakja őket egymás után, ameddig a szélesség azt engedi, utána új sort kezd. Megadható, hogy sor- vagy oszlopfolytonosan dolgozzon (`setOrientation()`).

Lássunk egy példát:

```
    @Override
    public void start(Stage primaryStage) {
      try {
        FlowPane root = new FlowPane();
        root.setPadding(new Insets(5, 0, 5, 0)); //Flowpane paddingje
        root.setVgap(10); //Elemek közötti vertikális hely
        root.setHgap(10); // horizontálisan a spacing
        root.setOrientation(Orientation.VERTICAL); //Orientáció beállítása

        root.getChildren().addAll(new Button("1"), new Button("2"), new
Button("3"));

        Scene scene = new Scene(root,400,400);
        primaryStage.setScene(scene);
        primaryStage.show();
      } catch(Exception e) {
```

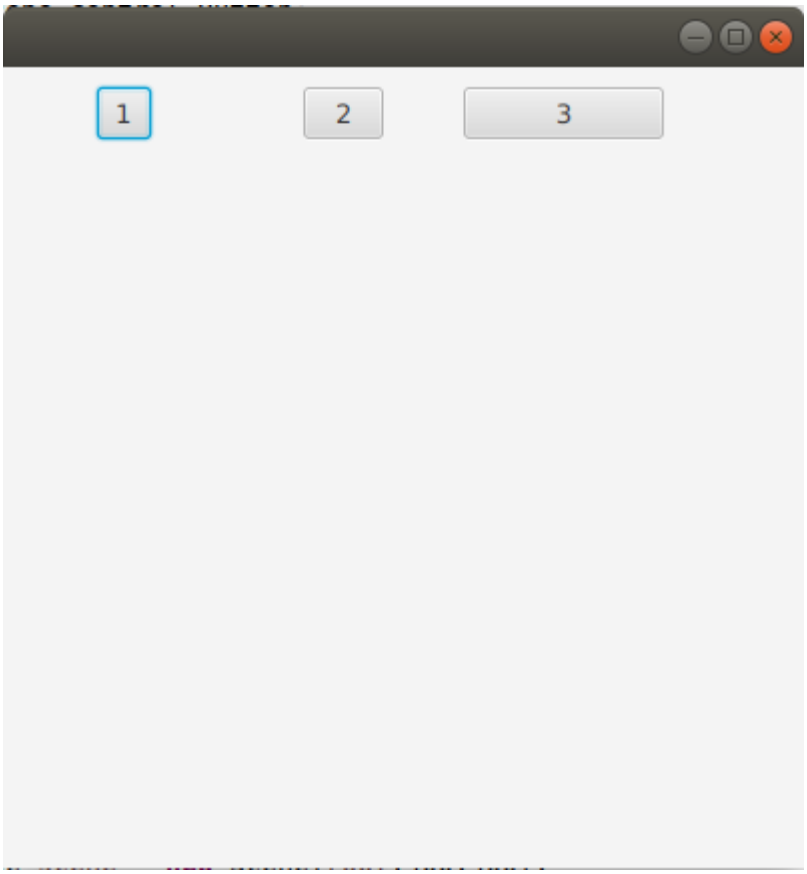
```
        e.printStackTrace();
      }
    }
```



TilePane

A `TilePane` nagyon hasonló a FlowPane-hez, de egy rácsos elrendezést biztosít az egymás után bepakolt elemekre. Fontos, hogy a cellák mérete ugyanakkora lesz.

```
TilePane root = new TilePane();
root.setPadding(new Insets(10));
root.setVgap(10);
root.setHgap(10);

Button btn1 = new Button("1");
btn1.setPrefWidth(20);

Button btn2 = new Button("2");
btn2.setPrefWidth(40);

Button btn3 = new Button("3");
btn3.setPrefWidth(100);

root.getChildren().addAll(btn1, btn2, btn3);
```



Amennyiben a példában kicseréljük a `TilePane`-t `FlowPane`-re, akkor az elemek közötti nagy 'rés' megszűnik, mert a `FlowPane` nem foglalkozik azzal hogy rácsos legyen az elrendezés, csak pakolja egymás után az elemeket.

AnchorPane

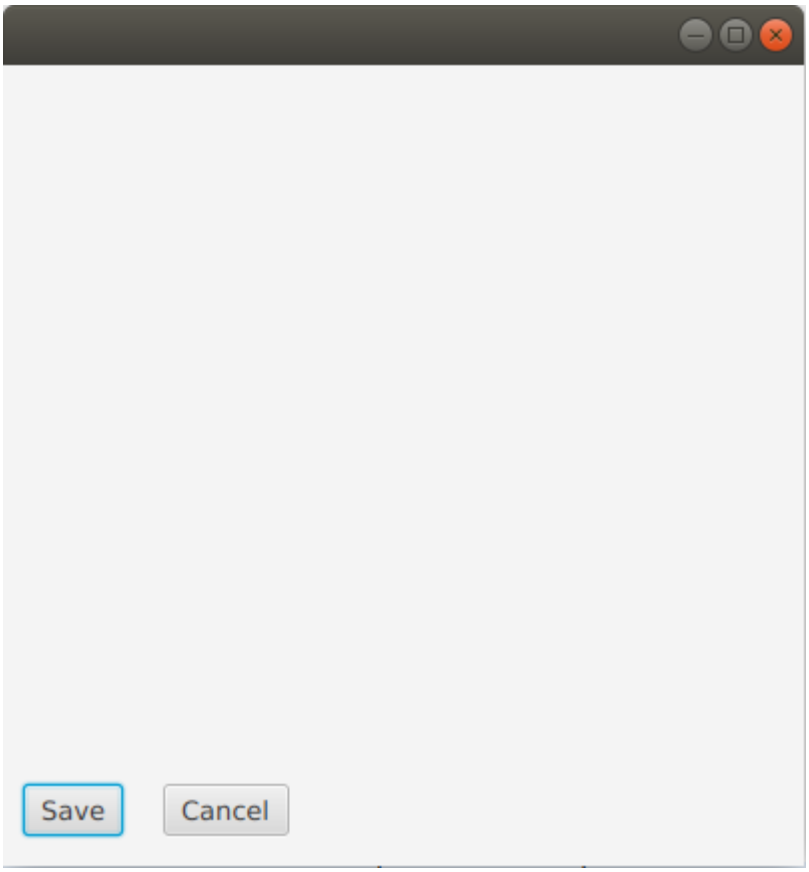
Az `AnchorPane` akkor jöhet jól, ha az ablak négy széléhez viszonyítva valamilyen elemeket fixen akarunk tartani. Például egy `HBox`-ban lévő gombokat alulra szeretnénk mindig rakni. Ez az átméretezéskor is megmarad. Ehhez az kell, hogy a `HBox`-ra egy horgonyt (anchor-t) állítsunk be.

Lássuk a példát:

```
...
HBox hb = new HBox();
Button buttonSave = new Button("Save");
Button buttonCancel = new Button("Cancel");
hb.setSpacing(20);
hb.setPadding(new Insets(10));
hb.getChildren().addAll(buttonSave, buttonCancel);

AnchorPane root = new AnchorPane();
root.getChildren().add(hb);
AnchorPane.setBottomAnchor(hb, 5.0);
Scene scene = new Scene(root,400,400);
...
```

Fontos, hogy egy node-ra több anchor-t is alkalmazhatunk. Ha például a fenti példában a `HBox`-ot nem csak lentre szeretnénk horgonyoztatni, hanem jobb oldalra is akkor az `AnchorPane.setRightAnchor(hb, 10)` hívást is meg kell ejtenünk.

A fenti példa eredménye:



További források

```
https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm
```

Dailógus ablakok

Alert

A JDK 8u40-val a JavaFX-be bekerültek a dialógus ablakokat megvalósító API osztályok. Az egyik ilyen osztály az `Alert`.

Lássunk is egy példát:

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Cím");
alert.setHeaderText("Az ablak tartalom felső header része");
alert.setContentText("Részletesebb leírás a header text alatt");

alert.showAndWait();
```

Az `Alert` többféle konstruktorral rendelkezik, de mindegyiknél megtalálható az `AlertType` paraméter, mellyel megadhatjuk, hogy milyen típusú dialógus ablakot szeretnénk létrehozni. A következő értékek adhatóak meg:

- `AlertType.INFORMATION`
- `AlertType.WARNING`
- `AlertType.ERROR`
- `AlertType.CONFIRMATION`
- `AlertType.NONE`

Ezek maguktól értetődnek, de próbáljuk ki mindegyiket, hogy lássuk a különbségeket. A `NONE` csak egy csupasz dialógus ablakot ad nekünk.

Egy `Alert`-nek 3 különböző szövegét állíthatjuk be, melyet a fenti példa is mutat. A dialógus ablaknak van egy címe, egy `header`-je (kb egy összegző szöveg), és egy részletesebb leírása (a `contentText`).

Ahogy azt korábban a `Stage`-nél láttuk, meg kell hívnunk a `show()` metódust, mivel addig a pontig nem látszik az ablak. A dialógus ablakainkat általában modálisként szeretjük tálalni a felhasználó elé, tehát nem akarjuk, hogy a rendszerben bármi mást tudjon csinálni a felhasználó ameddig a dialógusablakról nem gondoskodott megfelelően. Ehhez a legegyszerűbb, ha a fent is használt `showAndWait()` metódust használjuk. Fontos, hogy a `showAndWait()` vissza is tér egy `Optional<ButtonType>` eredménnyel, amelyből megtudhatjuk, hogy a felhasználó melyik gombot nyomta meg, ami fontos lehet például egy `CONFIRMATION` típusú dialógusnál.

Nézzünk egy példát:

```
package application;

import java.util.Optional;

import javafx.application.Application;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.ButtonType;
import javafx.stage.Stage;

public class Main extends Application {
  @Override
  public void start(Stage primaryStage) {

    primaryStage.setOnCloseRequest(e -> {
      Alert confirm = new Alert(AlertType.CONFIRMATION, null,
ButtonType.YES, ButtonType.NO);
      confirm.setTitle("Are you sure you want to exit?");
      confirm.setHeaderText("Are you sure you want to exit?");

      Optional<ButtonType> answer = confirm.showAndWait();

      if (answer.get() == ButtonType.NO) {
```

```
            e.consume();
          }
        });

        primaryStage.setWidth(400);
        primaryStage.setHeight(400);
        primaryStage.show();

      }

      public static void main(String[] args) {
        launch(args);
      }
    }
```

A példából több mindent is tanulhatunk, egyrészt az `Alert`-nek van egy olyan konstruktora is, ami a következőképpen néz ki:

```
Alert(AlertType alertType, String contentText, ButtonType... buttons)
```

Így egyszerűen megadhatjuk a `contentText`-et, illetve megadhatjuk, hogy milyen gombokat szeretnénk a dialógusra rakni. Jelen helyzetben egy Yes és egy No feliratú gombokat helyezünk el. Amennyiben ezeket megadjuk, akkor az alapértelmezett (OK és Cancel) gombok nem lesznek rápakolva az ablakra.

A dialógus akkor jelenik meg, ha megpróbáljuk bezárni a fő ablakunkat. Ehhez az eseménykezelőt a `setOnCloseRequest(...)` metódus segítségével állíthatjuk be.

A `showAndWait()` visszatérését eltároljuk egy lokális változóban, majd megállapítjuk, hogy a `NO` gombot nyomta-e meg a felhasználó. Ha mégsem akar kilépni az alkalmazásból, akkor az eseményt elkapjuk és nem küldjük tovább (Ezt teszi a `consume()` hívás). Ez azt jelenti, hogy a kilépési szándékunkat (az eseményt) nem továbbítjuk.

A fenti példában jól látható, hogy mi magunk választhatjuk ki, hogy milyen gombok jelenjenek meg a felületen. Ez rendben is van, de mi van akkor, ha magyar feliratú gombokat szeretnénk? A rendszer alapból angol feliratú gombokat biztosít. Szerencsére a JavaFX API erre is ad lehetőséget. Tekintsük a következő kódrészletet:

```
ButtonType buttonTypeNo = new ButtonType("Nem", ButtonData.NO);
```

Ilyen módon létrehozhatunk egy új gombtípust, melynek a felirata a 'Nem' szöveg, illetve viselkedését tekintve megegyezik a `ButtonType.NO` funkcionalitásával. Ezeket a viselkedéseket a `ButtonData` adja meg, mint azt a példa is mutatja. Érdemes lehet a dokumentációt megtekinteni az összes lehetséges variánsért.

További beállítási lehetőségként azt is megadhatjuk, hogy az ablak milyen elemekkel rendelkezzen. Ilyen például a stílusa.

```
dialog.initStyle(StageStyle.UTILITY);
```

Ebben az esetben az ablak csak a bezáró gombbal fog rendelkezni és minimize, maximize gombokkal nem.

Fontos lehet a szülőt beállítani: `dialog.initOwner(parentWindow);`

Illetve megadhatjuk a modalitás típusát is: `dialog.initModality(Modality.APPLICATION_MODAL);` Ennek eredményeképpen az ablakunk a teljes alkalmazásra nézve lesz modális nem pedig csak a szülőre nézve.

További dialógus típusok

Az `Alert`-en kívül a JavaFX biztosít még néhány további beépített dialógust. Ezeket nem fogjuk részletesen megnézni. Használatuk nagyban hasonlít az `Alert`-hoz. Ezek a következőek:

- `TextInputDialog`: egyszerű szöveges input is található a dialógusablakon, melynek tartalmát a `showAndWait()` adja vissza eredményül.
- `ChoiceDialog`: Egy legördülő mezőt tartalmazó dialógus ablak. Amennyiben a felhasználó ad meg értéket, akkor az elkérhető a `showAndWait()` visszatérési értékétől.
- `Dialog`: A lehető legáltalánosabb dialógus ablak, melynek minden részét egyedire szabhatunk ahogy csak akarunk.