JavaFX: Interacting with the User

17

Outcomes:

By the end of this chapter you should be able to:

- *create applications that utilize pull-down* ***menus*** *and* ***context menus***;
- *create a* ***modal*** *dialogue by defining a secondary stage*;
- *write applications that offer choices* via ***combo boxes***, ***check boxes*** *and* ***radio buttons***;
- *use a stack pane to create a* ***card menu***;
- *enable user interaction by using the subclasses of the* `Dialog` *class*.

17.1 Introduction

One of the most common ways for an application to interact with the user is to provide a number of choices—just as we did with the text-based menus that we introduced you to in the first semester. With graphical applications, there are a number of ways of doing this, and in this chapter we present you with a variety of techniques you can use—we will show you how to create drop-down menus, pop-up menus (also called context menus), check boxes, radio buttons and combo boxes. We will also show you how to interact with the user via popup dialogue windows.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_17

17.2 Drop-Down Menus

A drop-down menu, or pull-down menu, is a very common way to offer choices to the user of a program. In Fig. 17.1. we see a very simple example. The program displays a flag consisting of three horizontal stripes—the colour of each stripe can be changed by means of the pull-down menus on the top bar.

As an example, Fig. 17.2 shows the choices offered by the first menu:

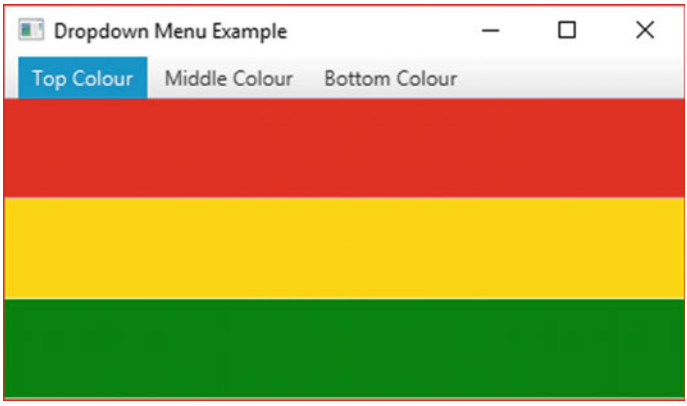


Fig. 17.1 An application with three pull-down menus

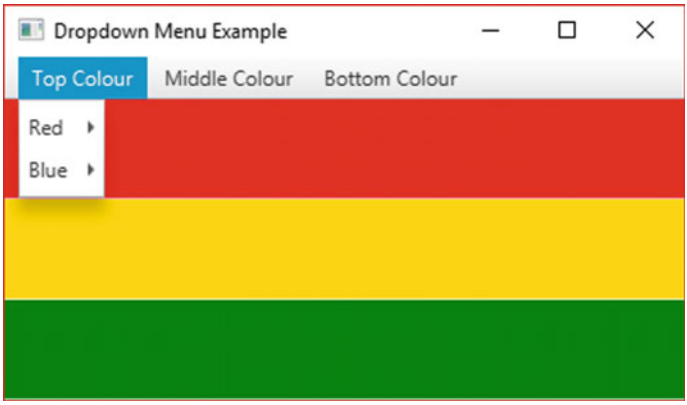


Fig. 17.2 The options on the "Top Colour" menu

The code for the `Flag` class is presented below:

Flag

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.layout.Background;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.geometry.Pos;

public class Flag extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 200;

        // create and configure a menu bar
        MenuBar bar = new MenuBar();
        bar.setMinHeight(25);

        // create drop-down menus
        Menu topStripeMenu = new Menu("Top Colour");
        Menu middleStripeMenu = new Menu("Middle Colour");
        Menu bottomStripeMenu = new Menu("Bottom Colour");

        // add the drop-down menus to the menu bar
        bar.getMenus().addAll(topStripeMenu, middleStripeMenu, bottomStripeMenu);

        // create menu items - two for each drop-down menu
        Menu red = new Menu("Red");
        Menu blue = new Menu("Blue");

        Menu gold = new Menu("Gold");
        Menu orange = new Menu("Orange");

        Menu green = new Menu("Green");
        Menu purple = new Menu("Purple");

        // add menu items to drop-down menus
        topStripeMenu.getItems().addAll(red, blue);
        middleStripeMenu.getItems().addAll(gold, orange);
        bottomStripeMenu.getItems().addAll(green, purple);


        // create the stripes
        Rectangle topStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);
        Rectangle middleStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);
        Rectangle bottomStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);

        // set initial colours
        topStripe.setFill(Color.RED);
        middleStripe.setFill(Color.GOLD);
        bottomStripe.setFill(Color.GREEN);

        // define the behaviour for each menu item
        red.setOnAction(e ->  topStripe.setFill(Color.RED));
        blue.setOnAction(e ->  topStripe.setFill(Color.BLUE));
        gold.setOnAction(e ->  middleStripe.setFill(Color.GOLD));
        orange.setOnAction(e ->  middleStripe.setFill(Color.ORANGE));
        green.setOnAction(e ->  bottomStripe.setFill(Color.GREEN));
        purple.setOnAction(e ->  bottomStripe.setFill(Color.PURPLE));

        // create VBox to hold the menu bar and the stripes
        VBox root = new VBox();
        root.setAlignment(Pos.TOP_LEFT);
        root.setBackground(Background.EMPTY);
        root.getChildren().addAll(bar, topStripe, middleStripe, bottomStripe);

        // create the scene and add the VBox
        Scene scene = new Scene(root, WIDTH, HEIGHT);

        // confgure the stage
        stage.setScene(scene);
        stage.setTitle("Dropdown Menu Example");
        stage.show();
    }

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

The first thing we do (after declaring constants for the width and height of the scene) is to create a menu bar:

```
MenuBar bar = new MenuBar();
bar.setMinHeight(25);
```

Next we create the three menus that will appear on the bar:

```
Menu topStripeMenu = new Menu("Top Colour");
Menu middleStripeMenu = new Menu("Middle Colour");
Menu bottomStripeMenu = new Menu("Bottom Colour");
```

Once you have created an instance of the `Menu` class, this object can hold other sub-menu items or can have an event handler attached to it which will enable it to respond to a mouse click (an `ActionEvent`).

In this case we will add two menu items for each stripe. First we create these items:

```
Menu red = new Menu("Red");
Menu blue = new Menu("Blue");

Menu gold = new Menu("Gold");
Menu orange = new Menu("Orange");

Menu green = new Menu("Green");
Menu purple = new Menu("Purple");
```

Then we add each one to the correct menu. The `getItems` method of the `Menu` class returns a list of items—and we add our items to this list with the `addAll` method.

```
topStripeMenu.getItems().addAll(red, blue);
middleStripeMenu.getItems().addAll(gold, orange);
bottomStripeMenu.getItems().addAll(green, purple);
```

The next thing is to declare and configure three rectangles for our stripes:

```
Rectangle topStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);
Rectangle middleStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);
Rectangle bottomStripe = new Rectangle(WIDTH, (HEIGHT-25)/3);

topStripe.setFill(Color.RED);
middleStripe.setFill(Color.GOLD);
bottomStripe.setFill(Color.GREEN);
```

We have arranged it so that the height of each stripe takes up one-third of the total height, less the height of the menu bar.

Now we add the event handlers, using the convenience method `setOnAction` that we have seen before:

```
red.setOnAction(e ->  topStripe.setFill(Color.RED));
blue.setOnAction(e ->  topStripe.setFill(Color.BLUE));
gold.setOnAction(e ->  middleStripe.setFill(Color.GOLD));
orange.setOnAction(e ->  middleStripe.setFill(Color.ORANGE));
green.setOnAction(e ->  bottomStripe.setFill(Color.GREEN));
purple.setOnAction(e ->  bottomStripe.setFill(Color.PURPLE));
```

All that remains is to create and configure a `VBox` to which we add the menu bar and the three stripes; then we add this to the scene, and finally we add the scene to the stage as usual:

```
VBox root = new VBox();
root.setAlignment(Pos.TOP_LEFT);
root.setBackground(Background.EMPTY);
root.getChildren().addAll(bar, topStripe, middleStripe, bottomStripe);

Scene scene = new Scene(root, WIDTH, HEIGHT);

stage.setScene(scene);
stage.setTitle("Dropdown Menu Example");
stage.show();
```

17.3 Context (Pop-Up) Menus

An alternative to a pull-down menu is a context menu, also referred to as a pop-up menu. A context menu is normally not available all the time, but pops up only when it is necessary, and then disappears. To demonstrate this we have created an application in which the menu is used simply to change the background colour of the graphic, and is invoked by pressing a button. This is demonstrated in Fig. 17.3.

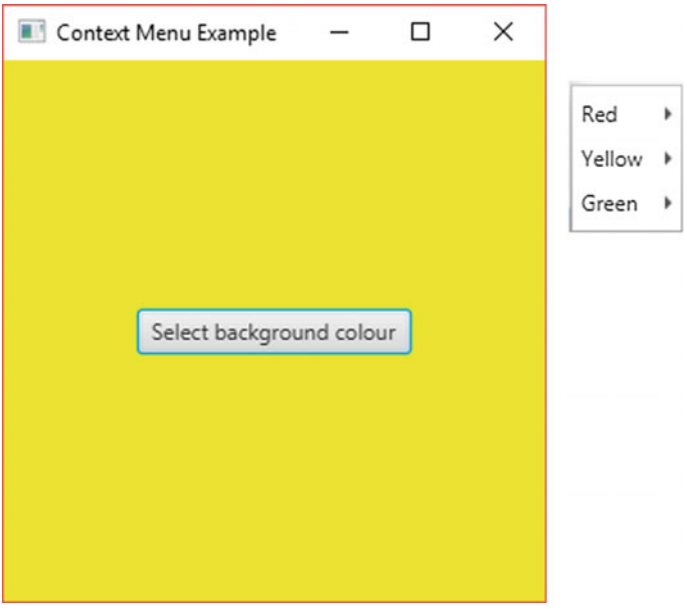


Fig. 17.3 A context menu

Here is the code for the `ContextMenuExample` class:

ContextMenuExample

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.geometry.Pos;
import javafx.geometry.Side;
import javafx.geometry.Insets;
import javafx.scene.control.Button;
import javafx.scene.control.ContextMenu;
import javafx.scene.control.Menu;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.FlowPane;

public class ContextMenuExample extends Application
{
   @Override
   public void start(Stage stage)
   {
     final double WIDTH = 300;
     final double HEIGHT = 300;

     // create a flow pane to be used as the root node
     FlowPane root = new FlowPane();
     root.setAlignment(Pos.CENTER);


     // create a button and add it to the flow pane
     Button button = new Button("Select background colour");
     root.getChildren().add(button);

     // create a context menu
     ContextMenu popup = new ContextMenu();

     // define the menu items
     Menu red = new Menu("Red");
     Menu yellow = new Menu("Yellow");
     Menu green = new Menu("Green");

     // add the menu items to the context menu
     popup.getItems().addAll(red, yellow, green);

     // add the event listeners: the background of the pane is changed and then the menu is closed
     red.setOnAction(e -> {
                                root.setBackground(new Background(new BackgroundFill(Color.RED,
                                                                  CornerRadii.EMPTY, Insets.EMPTY)));
                                popup.hide();
                            }
                    );

     yellow.setOnAction(e -> {
                                root.setBackground(new Background(new BackgroundFill(Color.YELLOW,
                                                                  CornerRadii.EMPTY, Insets.EMPTY)));
                                popup.hide();
                            }
                        );

     green.setOnAction(e ->  {
                                root.setBackground(new Background(new BackgroundFill(Color.GREEN,
                                                                  CornerRadii.EMPTY, Insets.EMPTY)));
                                popup.hide();
                            }
                        );

     // add the event listener to the button: the menu is shown when the button is pressed
     button.setOnAction(e -> popup.show(root, Side.RIGHT, 10, 10));

     // configure the scene and the stage
     Scene scene = new Scene(root, WIDTH, HEIGHT);
     stage.setScene(scene);
     stage.setTitle("Context Menu Example");
     stage.show();
   }

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

There is nothing very new here, and the code is mostly self-explanatory. But notice the code for the `setOnAction` method of the button:

```
button.setOnAction(e -> popup.show(root, Side.RIGHT, 10, 10));
```

The popup menu appears on the right side of the "anchor" node, offset by 10 pixels to the right and 10 from the top.

There is something else to notice about the menu that appears—it is **non-modal**. This means that the parent window is still accessible. This might be good for some applications, but in other applications you may want the application to be frozen until the context menu is hidden. In this case the menu is referred to as **modal**.

If we want a modal dialogue you can achieve this by creating a secondary stage. We have done this so that the popup menu has three buttons as shown in Fig. 17.4.

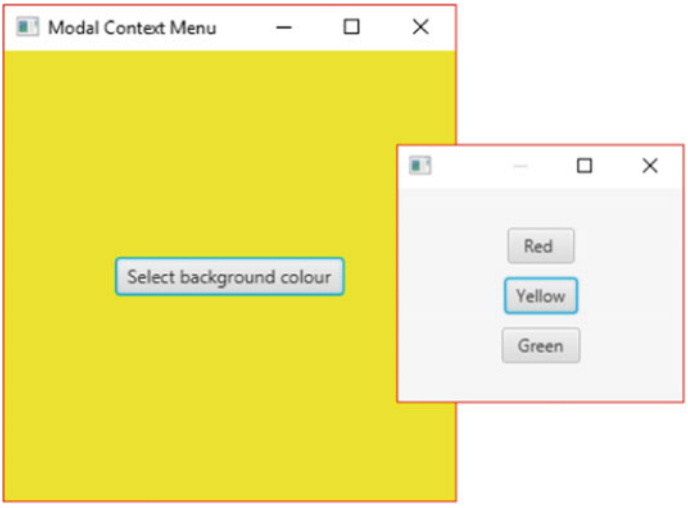


Fig. 17.4 A modal context menu

Here is the code:

ModalContextMenuExample

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.layout.FlowPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Modality;
import javafx.stage.Stage;

public class ModalContextMenuExample extends Application
{
    @Override
    public void start(Stage primaryStage)
    {
        final double WIDTH = 300;
        final double HEIGHT = 300;

        // create a flow pane to be used as the root node
        FlowPane root = new FlowPane();
        root.setAlignment(Pos.CENTER);

        // create a button and add it to the flow pane
        Button button = new Button("Select background colour");
        root.getChildren().add(button);

        // create buttons for the menu choices
        Button red = new Button(" Red  ");
        Button yellow = new Button("Yellow");
        Button green = new Button(" Green ");

        // create a VBox to hold the buttons
        VBox box = new VBox(10);
        box.setAlignment(Pos.CENTER);
        box.getChildren().addAll(red, yellow, green);

        // create a secondary scene
        Scene secondaryScene = new Scene(box,200,150);

        // create a secondary stage
        Stage secondaryStage = new Stage();

        // add the secondary scene to the secondaryary stage
        secondaryStage.setScene(secondaryScene);

        // set the modality of the secondary stage
        secondaryStage.initModality(Modality.APPLICATION_MODAL);

        // code the button so that the secondary stage is made visible
        button.setOnAction(e ->
        {               secondaryStage.setX(primaryStage.getX() + 250);
                                secondaryStage.setY(primaryStage.getY() + 100);
                                secondaryStage.show();
                                        });

        // code the menu items
        red.setOnAction(e -> {
                                 root.setBackground(new Background(new BackgroundFill(Color.RED,
                                                                        CornerRadii.EMPTY, Insets.EMPTY)));
                                 secondaryStage.hide();
                             }
                        );

        yellow.setOnAction(e -> {
                                 root.setBackground(new Background(new BackgroundFill(Color.YELLOW,
                                                                         CornerRadii.EMPTY, Insets.EMPTY)));
                                 secondaryStage.hide();
                                }
                           );

        green.setOnAction(e -> {
                                 root.setBackground(new Background(new BackgroundFill(Color.GREEN,
                                                                         CornerRadii.EMPTY, Insets.EMPTY)));
                                 secondaryStage.hide();
                                }
                            );

        // create the primary scene and stage
        Scene primaryScene = new Scene(root, WIDTH, HEIGHT);
        primaryStage.setScene(primaryScene);
        primaryStage.setTitle("Modal Context Menu");
        primaryStage.show();

        }

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

Much of this you will have seen before. The important thing here is how we create a secondary scene (which will hold a `VBox` containing the buttons) and a secondary stage:

```
Scene secondaryScene = new Scene(box,200,150);
Stage secondaryStage = new Stage();
secondaryStage.setScene(secondaryScene);
```

The `Stage` class has a method called `initModality`, which allows us to make the stage modal with respect to the rest of the application, so that once the stage (our menu in this case) appears the application freezes until it disappears. We do this with the following line of code:

```
secondaryStage.initModality(Modality.APPLICATION_MODAL);
```

17.4 Combo Boxes

You will be familiar with a combo box as it is a very common way of offering choices. Our example is shown in Fig. 17.5.

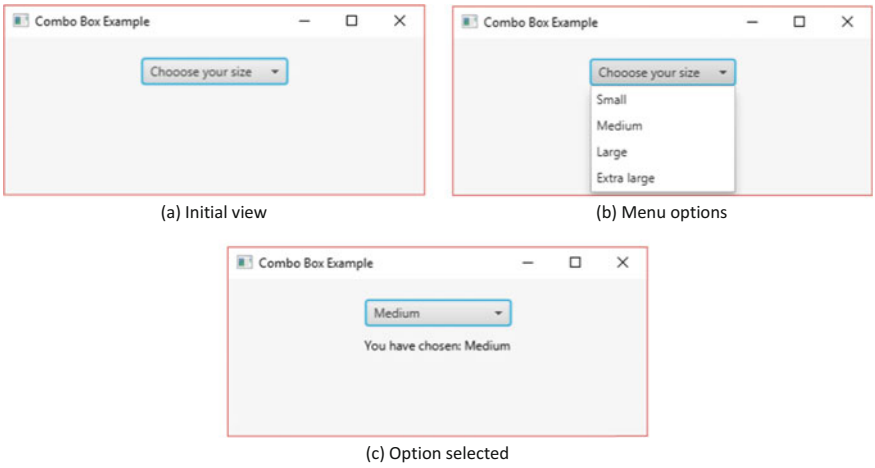


Fig. 17.5 Combo box example

Here is the code:

ComboBoxExample

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ComboBoxExample extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 150;

        // declare a String type combo box
        ComboBox<String> box = new ComboBox<>();

        // add the choices
        box.getItems().addAll("Small", "Medium", "Large", "Extra large");

        // set the intitial text
        box.setValue("Chooose your size");

        Label message = new Label();

        // display the user's choice
        box.setOnAction(e -> message.setText("You have chosen: " + box.getValue()));

        VBox root = new VBox(10);
        root.setPadding(new Insets(20, 20, 20, 20));
        root.setAlignment(Pos.TOP_CENTER);

        root.getChildren().addAll(box, message);

        Scene scene = new Scene(root, WIDTH, HEIGHT);
        stage.setScene(scene);
        stage.setTitle("Combo Box Example");
        stage.show();
    }

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

The only new thing here is the combo box itself. The `ComboBox` class is a generic class, so that the type of items held can vary—most commonly the box will hold strings, but we could just as easily have images for example. In our case we are using strings, and the declaration is therefore as follows:

```
ComboBox<String> box = new ComboBox<>();
```

The menu items are added by using the `getItems` method:

```
box.getItems().addAll("Small", "Medium", "Large", "Extra large");
```

We want the box to start off displaying the instruction, so we use the `setValue` method for this purpose:

```
box.setValue("Chooose your size");
```

The other point to note is the use of the `getValue` method to retrieve the current item displayed—we use this to display the choice made by the user:

```
box.setOnAction(e -> message.setText("You have chosen: " + box.getValue()));
```

17.5 Check Boxes and Radio Buttons

Check boxes are a very familiar way of offering choices. Our simple example is shown in Fig. 17.6.

You can see the code for this below. By now this should be self-explanatory—but notice that in this case we used the method `isSelected` to determine whether the box is selected or not.

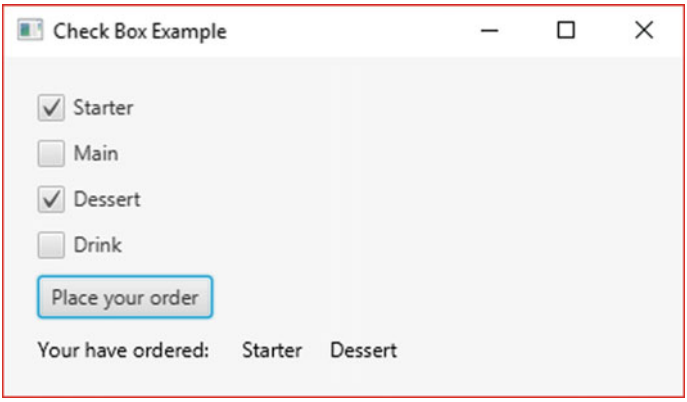


Fig. 17.6 Check box example

CheckBoxExample

```
import javafx.scene.control.Button;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CheckBoxExample extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 400;
        final double HEIGHT = 200;

        // create four check boxes
        CheckBox starter = new CheckBox("Starter");
        CheckBox mainCourse = new CheckBox("Main");
        CheckBox dessert = new CheckBox("Dessert");
        CheckBox drink = new CheckBox("Drink");

        Button submitButton = new Button("Place your order");
        Label message = new Label();

        // clicking the button
        submitButton.setOnAction(e -> {
                                        String yourOrder = "Your have ordered: ";

                                        if(!starter.isSelected()&& !mainCourse.isSelected()
                                                  && !dessert.isSelected() && !drink.isSelected())
                                        {
                                                yourOrder = "You did not select anything";
                                        }
                                        else
                                        {
                                                if(starter.isSelected())
                                                {
                                                        yourOrder = yourOrder + "     Starter";
                                                }
                                                if(mainCourse.isSelected())
                                                {
                                                        yourOrder = yourOrder + "     Main";
                                                }
                                                if(dessert.isSelected())
                                                {
                                                        yourOrder = yourOrder + "     Dessert";
                                                }
                                                if(drink.isSelected())
                                                {
                                                        yourOrder = yourOrder + "     Drink";
                                                }
                                        }
                                             message.setText(yourOrder);
                                      }
                                );


        VBox root = new VBox(10);
        root.setPadding(new Insets(20, 20, 20, 20));
        root.setAlignment(Pos.CENTER_LEFT);

        root.getChildren().addAll(starter, mainCourse, dessert, drink, submitButton, message);

        Scene scene = new Scene(root, WIDTH, HEIGHT);

        stage.setScene(scene);
        stage.setTitle("Check Box Example");
        stage.show();
    }

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

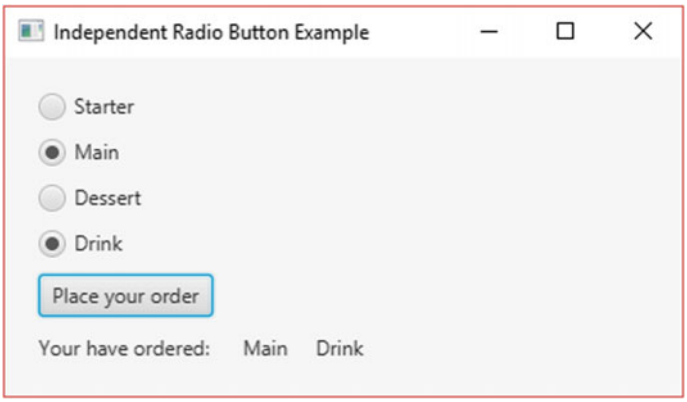


Fig. 17.7 Independent radio buttons

Radio buttons are very similar to check boxes, although they are round instead of square. They can operate in exactly the same way as check boxes, but can also be made to operate as a group, so that only one item can be selected at a time; if a box is selected and then the user chooses another box, the first one is cleared.

Figure 17.7 shows an example of radio buttons working independently. You can see that it is the same as our check box example, with the check boxes replaced by radio buttons.

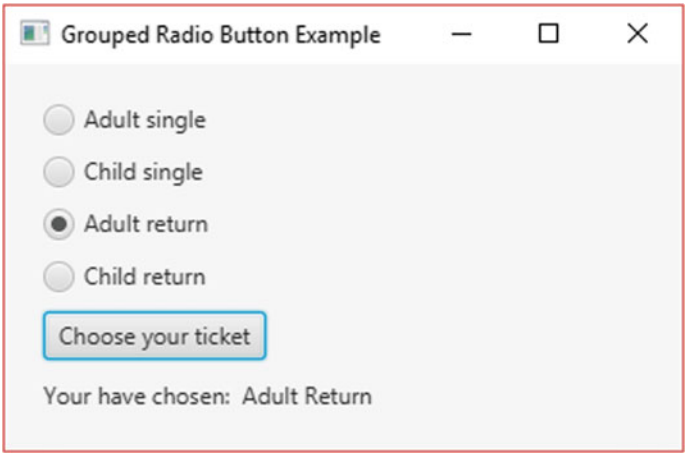


Fig. 17.8 Grouped radio buttons

All we needed to do was to replace the declarations of the four check boxes with the following code:

```
RadioButton starter = new RadioButton("Starter");
RadioButton mainCourse = new RadioButton("Main");
RadioButton dessert = new RadioButton("Dessert");
RadioButton drink = new RadioButton("Drink");
```

Figure 17.8 shows an example of radio buttons acting together in a group—only one item can be selected.

Here is the code:

GroupedRadioButtonExample

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.RadioButton;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.geometry.Pos;
import javafx.scene.control.ToggleGroup;

public class GroupedRadioButtonExample extends Application
{
    @Override
    public void start(Stage stage)
    {
        final double WIDTH = 350;
        final double HEIGHT = 200;

        // declare the radio buttons
        RadioButton adultSingle = new RadioButton("Adult single");
        RadioButton childSingle = new RadioButton("Child single");
        RadioButton adultReturn = new RadioButton("Adult return");
        RadioButton childReturn = new RadioButton("Child return");

        // add the radio buttons to a toggle group
        ToggleGroup group = new ToggleGroup();
        group.getToggles().addAll(adultSingle, childSingle, adultReturn, childReturn);

        Button submitButton = new Button("Choose your ticket");
        Label message = new Label();

        // clicking the button
        submitButton.setOnAction(e-> {
                                        String yourOrder = "Your have chosen: ";
                                        if(!adultSingle.isSelected()&& !childSingle.isSelected()
                                          && !adultReturn.isSelected() && !childReturn.isSelected())
                                        {
                                                yourOrder = "You did not chose a ticket";
                                        }
                                        else
                                        {
                                                if(adultSingle.isSelected())
                                                {
                                                       yourOrder = yourOrder + " Adult Single";
                                                }
                                                else if(childSingle.isSelected())
                                                {
                                                       yourOrder = yourOrder + " Child Single";
                                                }
                                                else if(adultReturn.isSelected())
                                                {
                                                       yourOrder = yourOrder + " Adult Return";
                                                }
```

```
                                                else if(childReturn.isSelected())
                                                {
                                                        yourOrder = yourOrder + " Child Return";
                                                }
                                        }
                                        message.setText(yourOrder);
                                }
                        );


        VBox root = new VBox(10);
        root.setPadding(new Insets(20, 20, 20, 20));
        root.setAlignment(Pos.CENTER_LEFT);
        root.getChildren().addAll(adultSingle, childSingle, adultReturn, childReturn,
                                                                        submitButton, message);

        Scene scene = new Scene(root, WIDTH, HEIGHT);
        stage.setScene(scene);
        stage.setTitle("Grouped Radio Button Example");
        stage.show();
    }

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

As you can see we have declared a `ToggleGroup` and added our radio buttons to it:

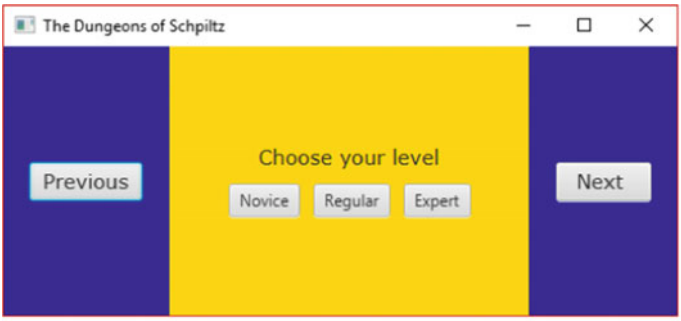
```
ToggleGroup group = new ToggleGroup();
group.getToggles().addAll(adultSingle, childSingle, adultReturn, childReturn);
```

The buttons now act as one unit—if a button is already selected when another button is chosen, then the first button is cleared.

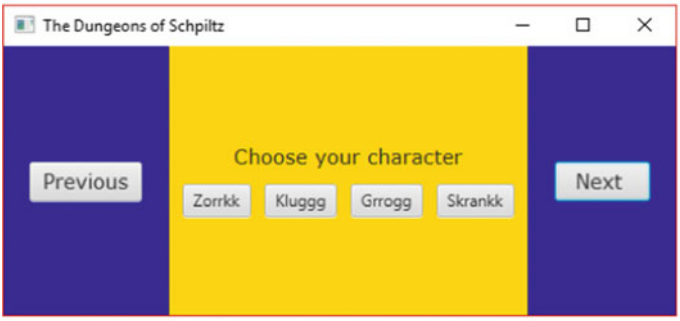
17.6 A Card Menu

A very familiar way of entering information into forms is via a serious of screens. As we mentioned in Chap. 10, one way to achieve this is via a `StackPane`, which enables us to present a series of containers as if they represented a pack of cards. The top "cards" can be removed in order, revealing the card underneath—or we could move in the other direction, replacing the cards in the order they were removed.

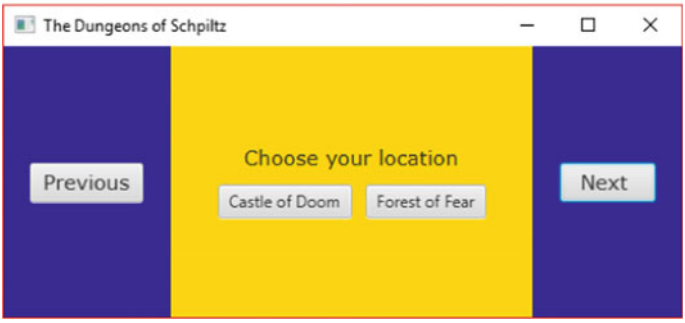
The example shown in Fig. 17.9 represents the initial screens in a made-up game ("The Dungeons of Schpiltz"). A `StackPane` holds the cards, each of which is a `VBox` containing buttons. We have not coded the buttons, as it is just for illustration. On either side of the `StackPane` are a "Previous" button and a "Next" button.



(a) The first screen allows the user to choose a level



(b) The second screen allows the user to choose a character



(c) The third screen allows the user to choose a location

Fig. 17.9 A card menu

Here is the complete code:

CardMenuExample

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Button;
import javafx.scene.paint.Color;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.CornerRadii;
import javafx.scene.text.Font;

public class CardMenuExample extends Application
{
    private int currentCard = 0; // to keep track of the cards
    private StackPane stack = new StackPane();  // to hold the cards

    @Override
    public void start(Stage stage)
    {
        // create a label for the first card (choosing the level)
        Label levelLabel = new Label("Choose your level");
        levelLabel.setFont(Font.font ("Verdana", 15));

        // create an HBox containing three dummy buttons for the first card (choosing the level)
        HBox levelButtons = new HBox(10);
        levelButtons.getChildren().addAll(new Button("Novice"),
                                          new Button("Regular"), new Button ("Expert"));
        levelButtons.setAlignment(Pos.CENTER);

        // create a VBox to act as the first card
        VBox levelPanel = new VBox(10);

        // add the label and buttons to the first VBox
        levelPanel.getChildren().addAll(levelLabel, levelButtons);
        levelPanel.setAlignment(Pos.CENTER);

        // create a label for the second card (choosing the character)
        Label characterLabel = new Label("    Choose your character    ");
        characterLabel.setFont(Font.font ("Verdana", 15));

        // create an HBox containing four dummy buttons for the second card (choosing the character)
        HBox characterButtons = new HBox(10);
        characterButtons.getChildren().addAll(new Button("Zorrkk"), new Button("Kluggg"),
                                              new Button ("Grrogg"), new Button("Skrankk"));
        characterButtons.setAlignment(Pos.CENTER);

        // create a VBox to act as the second card
        VBox characterPanel = new VBox(10);
        characterPanel.getChildren().addAll(characterLabel, characterButtons);
        characterPanel.setAlignment(Pos.CENTER);

        // create a label for the second card (choosing the location)
        Label locationLabel = new Label("Choose your location");
        locationLabel.setFont(Font.font ("Verdana", 15));

        // create an HBox containing two dummy buttons for the third card (choosing the location)

        HBox locationButtons = new HBox(10);
        locationButtons.getChildren().addAll(new Button("Castle of Doom"),
                                             new Button("Forest of Fear"));
        locationButtons.setAlignment(Pos.CENTER);

        // create a VBox to act as the third card
        VBox locationPanel = new VBox(10);
        locationPanel.getChildren().addAll(locationLabel, locationButtons);
        locationPanel.setAlignment(Pos.CENTER);


        // create and configure buttons for moving back and forth through the cards
        Button nextButton = new Button(" Next  ");
        Button previousButton = new Button("Previous");

        nextButton.setFont(Font.font ("Verdana", 15));
        previousButton.setFont(Font.font ("Verdana", 15));
```

```
        // configure the stack pane
        stack.setPadding(new Insets(10, 10, 10, 10));
        stack.setBackground(new Background(new BackgroundFill(Color.GOLD,
                                                              CornerRadii.EMPTY, Insets.EMPTY)));
        stack.setAlignment(Pos.CENTER);

        // add the cards to the stack pane
        stack.getChildren().addAll(levelPanel, characterPanel, locationPanel);

        // show the first card and hide the other two
        stack.getChildren().get(0).setVisible(true);
        stack.getChildren().get(1).setVisible(false);
        stack.getChildren().get(2).setVisible(false);

        // create and configure an HBox
        HBox root = new HBox(20);
        root.setBackground(Background.EMPTY);
        root.setAlignment(Pos.CENTER);

        // add the "previous" button, the stack of cards and the "next" button
        root.getChildren().addAll(previousButton, stack, nextButton);

        // add event handlers to call the relevant helper methods
        nextButton.setOnAction (e -> next());
        previousButton.setOnAction (e -> previous());


        // create and configure the scene
        Scene scene = new Scene(root, 500, 200, Color.DARKBLUE);

        // configure the stage
        stage.setScene(scene);
        stage.setTitle("The Dungeons of Schpiltz");
        stage.show();
    }

    // define the helper method for the "next" button
    private void next()
    {
            if(currentCard !=  stack.getChildren().size()- 1) // if we are not at the last card
            {
                currentCard++; // make the next card the current card

                // move through the cards: show the current card, hide the others
                for(int i = 0; i <=  stack.getChildren().size()- 1; i++)
                {
                    if(i == currentCard)
                    {
                        stack.getChildren().get(i).setVisible(true);
                    }
                    else
                    {
                        stack.getChildren().get(i).setVisible(false);
                    }
                }
            }
    }

    // define the helper method for the "previous" button
     private void previous()
     {
             if(currentCard != 0) // if we are not at the first card
             {
                 currentCard--; // make the previous card the current card

                 // move through the cards: show the current card, hide the others
                 for(int i = 0; i <=  stack.getChildren().size()- 1; i++)
                 {
                     if(i == currentCard)
                     {
                         stack.getChildren().get(i).setVisible(true);
                     }
                     else
                     {
                         stack.getChildren().get(i).setVisible(false);
                     }
                 }
             }
     }

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

}
```

We are using helper methods for the "Previous" button and the "Next" buttons, so we have defined a couple of attributes that can be accessed by these methods. The first is a counter that keeps track of the current card, the second is the stack pane that will hold the cards:

```
private int currentCard = 0;
private StackPane stack = new StackPane();
```

The `start` method begins by creating the first card. First we create a label which gives the instruction to the user:

```
Label levelLabel = new Label("Choose your level");
levelLabel.setFont(Font.font ("Verdana", 15));
```

Next an `HBox` to hold three dummy buttons:

```
HBox levelButtons = new HBox(10);
levelButtons.getChildren().addAll(new Button("Novice"),
                                  new Button("Regular"), new Button ("Expert"));
levelButtons.setAlignment(Pos.CENTER);
```

Finally we create a `VBox` to act as the first card, then add the label and the button container to it:

```
VBox levelPanel = new VBox(10);
levelPanel.getChildren().addAll(levelLabel, levelButtons);
levelPanel.setAlignment(Pos.CENTER);
```

We do the same thing for the other two cards, then go on to configure the stack pane and add the three cards to it.

The items held by the stack pane are indexed from zero (in the order they were added), and they are retrieved with the `get` method of the list retrieved by the `getChildren` method. Using the indices we set the initial state, with the first card visible and the other two invisible:

```
stack.getChildren().get(0).setVisible(true);
stack.getChildren().get(1).setVisible(false);
stack.getChildren().get(2).setVisible(false);
```

We go on to create and configure an `HBox`, to which we add the "Previous" button, the stack and the "Next" button.

Before we finally create the scene and the stage, we add our event handlers to the two buttons; these call the helper methods, `next` and `previous`.

```
nextButton.setOnAction (e -> next());
previousButton.setOnAction (e -> previous());
```

The code for the `next` method is as follows:

```
    private void next()
    {
            if(currentCard != stack.getChildren().size()- 1)
            {
                currentCard++;
                for(int i = 0; i <= stack.getChildren().size()- 1; i++)
                {
                    if(i == currentCard)
                    {
                        stack.getChildren().get(i).setVisible(true);
                    }
                    else
                    {
                        stack.getChildren().get(i).setVisible(false);
                    }
                }
            }
    }
```

First we have checked that we are not at the last card—for this purpose we have used the `getSize` method of the list of items (although we know that the index of the last item is 2, doing it this way would allow us to add more cards without changing the code).

If we are not at the last card we increment the counter and then cycle through the cards; we set the current card to be visible, the others to be invisible.

The `previous` method behaves in a similar way.

17.7 The *Dialog* Class

JavaFX provides a very useful control class called `Dialog`, which has subclasses called `Alert`, `ChoiceDialog`, `TextInputDialog`. These provide popup windows which allow the user to view or enter information. One of the very useful aspects of this is that we can begin an application by showing a popup window that gets some information from the user before showing the main scene graphic.

The program below demonstrates how these classes work.

DialogDemo

```
import java.util.Optional;
import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Button;
import javafx.scene.control.ChoiceDialog;
import javafx.scene.control.Label;
import javafx.scene.control.TextInputDialog;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class DialogDemo extends Application
{
    private String name;
    private String colour;

    @Override
    public void start(Stage stage)
    {
        name = getUserName(); // get the user name by calling a text input dialog

        Label label1 = new Label();
        Label label2 = new Label();
        Button button1 = new Button ("Alert");
        Button button2 = new Button ("Choice");

        button1.setOnAction(e -> showAlert()); // show an alert

        // call a choice dialog
        button2.setOnAction(e ->
                                {
                                   colour = showChoice();
                                   label2.setText("You chose " + colour);
                                }
                            );

        VBox root = new VBox(10);
        root.setAlignment(Pos.CENTER);
        root.getChildren().addAll(label1, button1, button2, label2);
        label1.setFont(Font.font("Ariel", 20));
        label2.setFont(Font.font("Ariel", 20));
        label1.setText("Hello " + name);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Demo");
        stage.setWidth(250);
        stage.setHeight(250);
        stage.show();
    }
    private String getUserName()
    {
        TextInputDialog dialog = new TextInputDialog();
        dialog.setHeaderText("Enter your name");
        dialog.setTitle("Text Input Dialog");

        Optional<String> response = dialog.showAndWait();
        return response.get();
    }

    private void showAlert()
    {
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setHeaderText("Information Alert");
        alert.setContentText(name + " is a cool name");
        alert.showAndWait();
    }

    private String showChoice()
    {
        ChoiceDialog<String> choice = new ChoiceDialog<>("Red", "Yellow", "Blue");
        choice.setContentText("Choose colour");
        choice.setHeaderText("Choice dialog");

        Optional<String> response = choice.showAndWait();
        return response.get();
    }

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

}
```

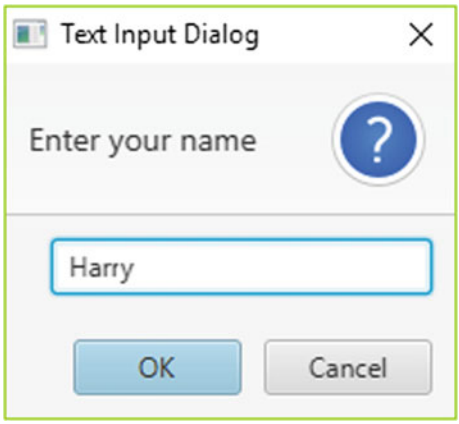


Fig. 17.10 A text input dialog

As you can see, the first thing that happens, even before the scene is configured and shown, is that a helper method `getUserName` is called. This causes the following popup to appear as shown in Fig. 17.10.

The code for `getUserName` is as follows:

```
private String getUserName()
{
    TextInputDialog dialog = new TextInputDialog();
    dialog.setHeaderText("Enter your name");
    dialog.setTitle("Text Input Dialog");

    Optional<String> response = dialog.showAndWait();
    return response.get();
}
```

As you can see from the code, we create and configure a `TextInputDialog`, then call its `showAndWait` method, which does exactly what it says—shows the dialogue and waits for a value to be entered. The value entered is returned as an `Optional` object (as explained in Chap. 14), in this case `Optional <String>`. The `String` value is retrieved with the `get` method of `Optional`.

Once the dialogue is closed, the main graphic appears (Fig. 17.11).

The two buttons are provided to demonstrate the `Alert` and the `ChoiceDialog` classes. Pressing the "Alert" button calls another helper method, `showAlert`, which brings up the dialogue window shown below in Fig. 17.12:

The code for `showAlert` is as follows:

```
private void showAlert()
{
    Alert alert = new Alert(AlertType.INFORMATION);
    alert.setHeaderText("Information Alert");
    alert.setContentText("name + " is a cool name");
    alert.showAndWait();
}
```

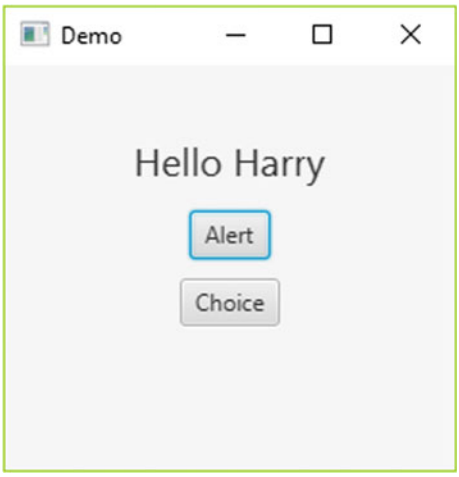


Fig. 17.11 Dialog demo application

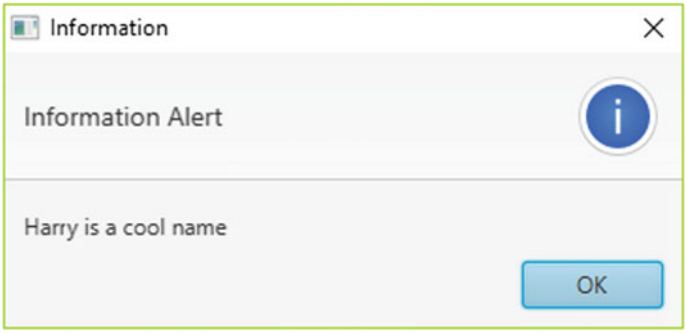
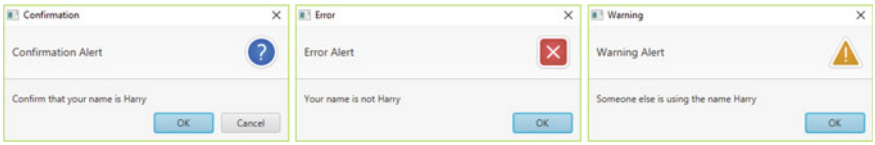


Fig. 17.12 An information alert

There are a number of different types of `Alert`, and the particular type is provided as a parameter to the constructor. The type that you see in Fig. 17.12 is `AlertType.INFORMATION`. Other types are shown in Fig. 17.13:

There is an additional constructor of `Alert` that enables you to choose which buttons you would like. It takes the following form:

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



`AlertType.CONFIRMATION` `AlertType.ERROR` `AlertType.WARNING`

Fig. 17.13 Other alert types

The last of these parameters, `buttons`, allows you to decide upon the type or button—or buttons—you require. So for example, the following statement:

```
Alert alert
   = new Alert(AlertType.INFORMATION, "Alert with three buttons", ButtonType.APPLY, ButtonType.OK, ButtonType.CLOSE);
```

would give rise to the alert shown in Fig. 17.14.

The button types available are shown in Table 17.1.

In order to find out which of the three buttons had been pressed you would need to examine the return type of the `showAndWait` method:

```
Optional<ButtonType> response = alert.showAndWait();
```

The `ButtonType` could then be extracted with the `get` method of `Optional`, and a string representing the button type could then be extracted with the `getText` method of `ButtonType`:

```
String buttonPressed = response.get().getText();
```

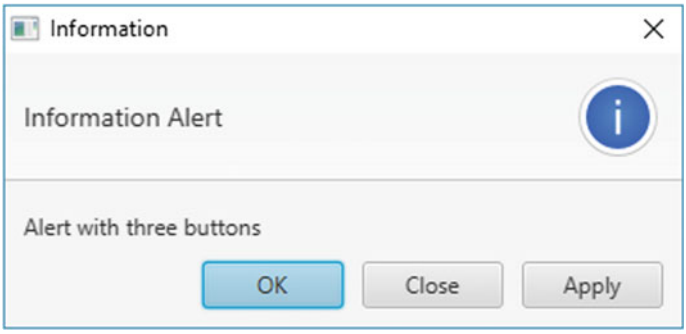


Fig. 17.14 A choice dialog with three buttons

Table 17.1 Available button types

`ButtonType.APPLY`
`ButtonType.CLOSE`
`ButtonType.CANCEL`
`ButtonType.FINISH`
`ButtonType.NEXT`
`ButtonType.PREVIOUS`
`ButtonType.YES`
`ButtonType.NO`
`ButtonType.OK`

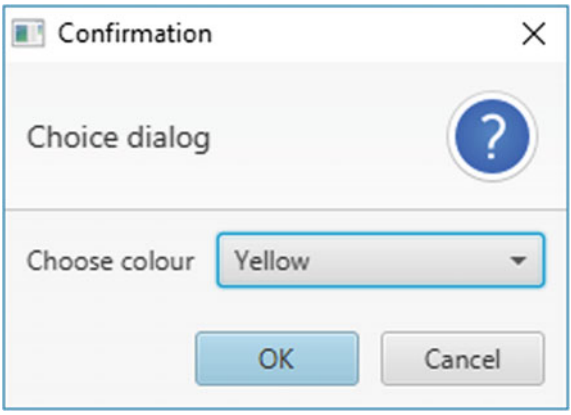


Fig. 17.15 A choice dialog

Finally, the "choice" button in Fig. 17.11 will invoke the `showChoice` method which brings up a choice dialog (Fig. 17.15).

Here is the code for `showChoice`, which by now should be self-explanatory (notice, however, that `ChoiceDialog` is a generic class and requires the type of the items to be held):

```
ChoiceDialog<String> choice = new ChoiceDialog<>("Red", "Yellow", "Blue");
choice.setContentText("Choose colour");
choice.setHeaderText("Choice dialog");

Optional<String> response = choice.showAndWait();
return response.get();
```

Once this dialogue window is closed, the choice made is shown (Fig. 17.16).

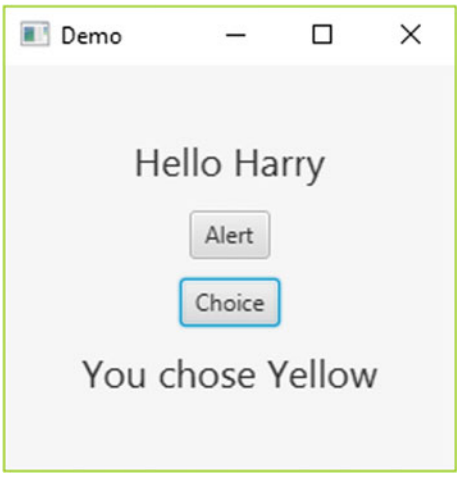
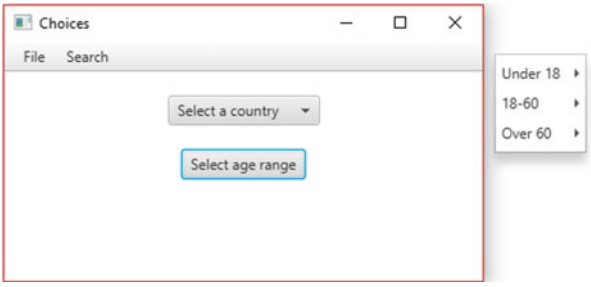


Fig. 17.16 Dialog demo after the choice of colour has been made

17.8 Self-test Questions

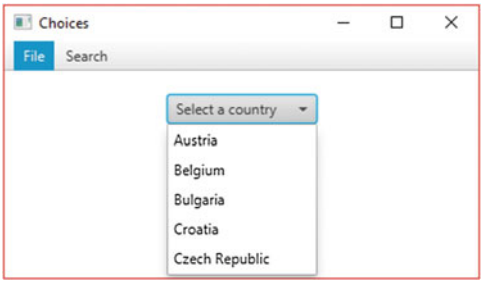
1. (a) In the application shown below, identify the various ways that users are given for making choices.



(b) What alternatives could have been used for selecting the age range?

2. What is the difference between a *modal* and a *non-modal* dialogue?
3. In what circumstances might a *context* menu be preferable to a *drop-down* menu?

4. Explain how a number of *radio buttons* can be made to work together.
5. The diagram below shows the choices available under the "Select a country" option of the application shown in question 1.



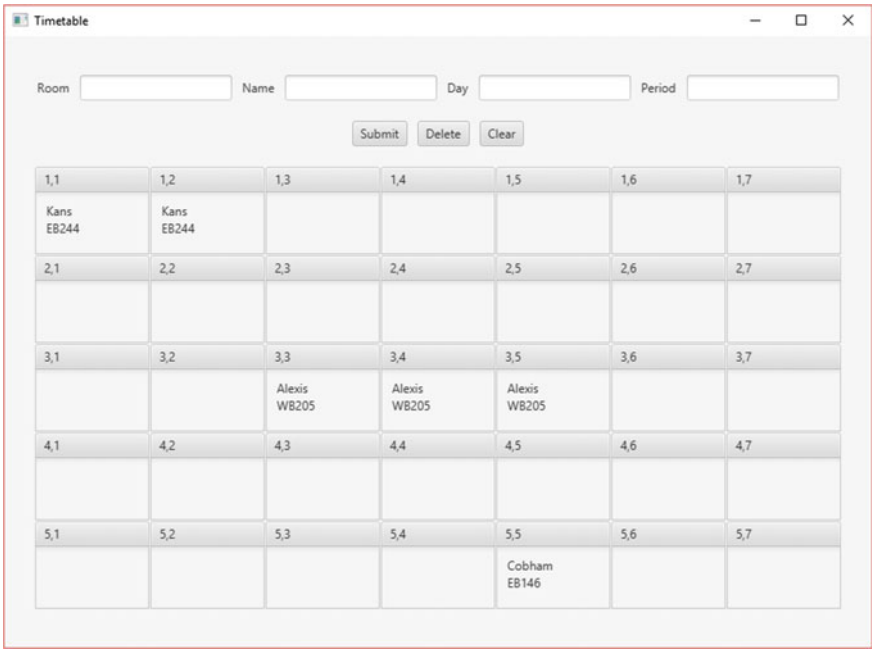
Referring to the above diagram, explain how you would begin a conditional statement that would execute some code if Bulgaria had been chosen.

17.9 Programming Exercises

1. Implement a few of the programs that we have developed in this chapter, and experiment with different settings in order to change some the features.
2. Adapt the `Hostel` case study of Chaps. 11 and 12 as follows:

   (a) Make use of a `ComboBox` to enter the month.
   (b) Enable the number of rooms to be entered via a `Dialog`, as described in Sect. 17.7.
   (c) Make use of `Alerts`, as described in Sect. 17.7.

3. Create a graphical user interface for the `Library` class that we developed in Chap. 15. Make use of the many JavaFX features discussed in this chapter.
4. At the end of Chap. 8 you were asked to develop a time table application. You later enhanced this application by making use of exceptions at the end of Chap. 14. Now develop a JavaFX GUI for this application, with the timetable displayed as a grid. The following is an example—you can choose your own design:


Timetable
Room
Name
Day
Period
Submit
Delete
Clear
1,1
1,2
1,3
1,4
1,5
1,6
1,7
Kans
EB244
Kans
EB244
2,1
2,2
2,3
2,4
2,5
2,6
2,7
3,1
3,2
3,3
3,4
3,5
3,6
3,7
Alexis
WB205
Alexis
WB205
Alexis
WB205
4,1
4,2
4,3
4,4
4,5
4,6
4,7
5,1
5,2
5,3
5,4
5,5
5,6
5,7
Cobham
EB146