ORACLE UNIVERSITY

Integrated Cloud Applications & Platform Services

Java SE 7: Develop Rich Client Applications

Student Guide - Volume I

D67230GC10

Edition 1.0 | December 2016 | D77353

Learn more from Oracle University at **education.oracle.com**

ORACLE

Authors

Michael J. Williams
Paromita Dutta
Anjana Shenoy
Cindy Church

Technical Contributors and Reviewers

Debra Masada
Ajay Bharadwaj
Aurelio Garcia-Ribeyro
Steve Watt
Nancy Hildebrandt
Alla Redko
Tom McGinn
Matt Heimer
Sharon Zahkour
Nicolas Lorain
Brian Goetz
Joe Darcy
Alessandro Leite
Jai Suri
Richard Bair
Jasper Potts

Editors

Daniel Milne
Anwesha Ray
Richard Wallis

Graphic Designer

Seema Bopaiah

Publishers

Pavithran Adka
Durga Ramesh

Contents

1

Introduction to the Course

Course Objectives

After completing this course, you should be able to:

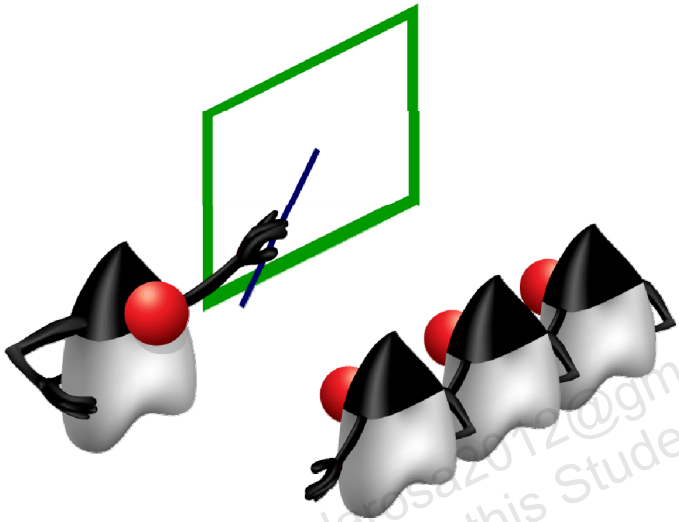
- Apply the Model View Controller (MVC) design pattern to create reusable classes
- Implement a complete program that can be used in an intranet application
- Leverage the Java Persistence API (JPA) in a Java SE environment
- Organize and set up GUI generation and event handling using JavaFX
- Implement the Logging API to generate log messages in the GUI
- Create two-tier and three-tier Java technology applications

Course Objectives

- Connect your application to a REST web service
- Package and deploy a Java SE application
- Secure a Java SE application

Audience

This course is intended for Java developers and JavaFX developers who have at least one year of development experience.

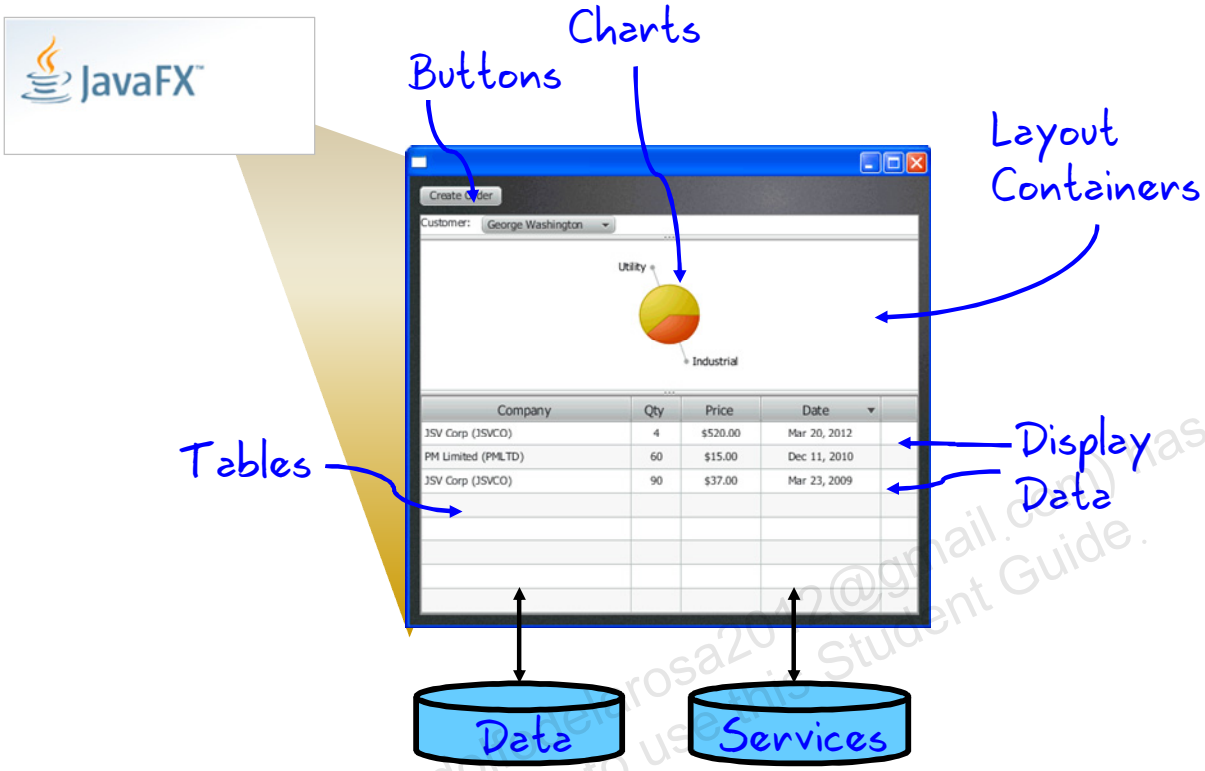


Prerequisite Skills

You should already know how to:

- Use object-oriented programming techniques
- Develop applications by using the Java programming language
- Understand how to implement interfaces and handle Java programming exceptions

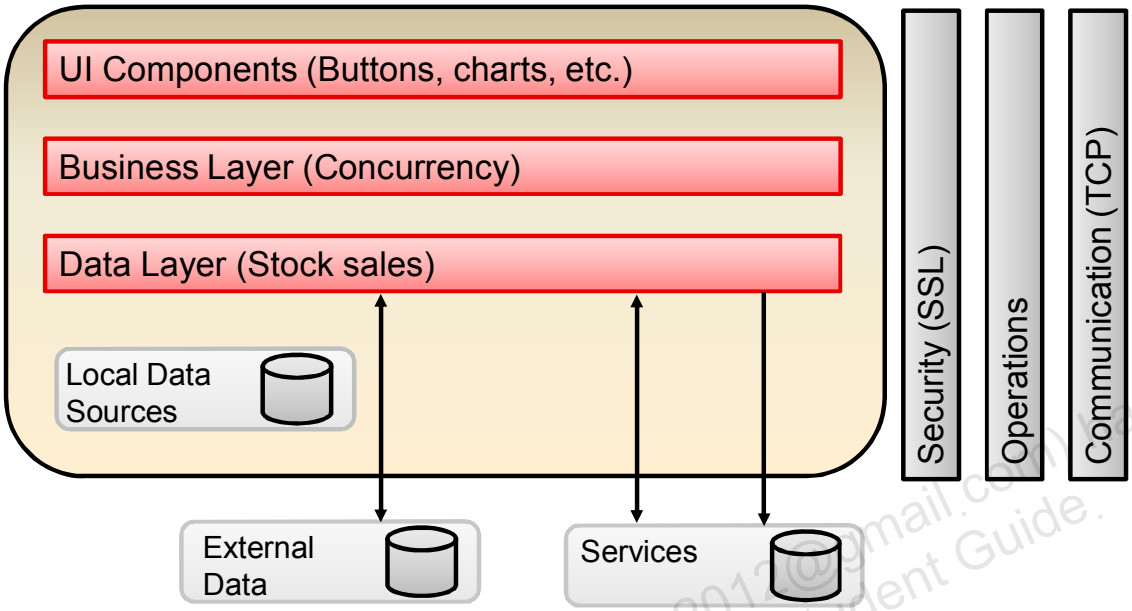
Use JavaFX to Create Rich-Client Applications



A rich-client application is an application that has an interface that relates to the back end without cluttering the user interface. JavaFX has a complete set of buttons, charts, tables, and layout containers that you can use to create a rich user interface. In addition, you can style the client by using CSS. All of these components connect to and display the data back end in an easy-to-read manner.

JavaFX is used to create the rich-client application that we develop throughout this course. Subsequent lessons provide more detail about JavaFX. All of the GUI development lessons use JavaFX.

Rich-Client Application



A rich-client application typically has the following attributes:

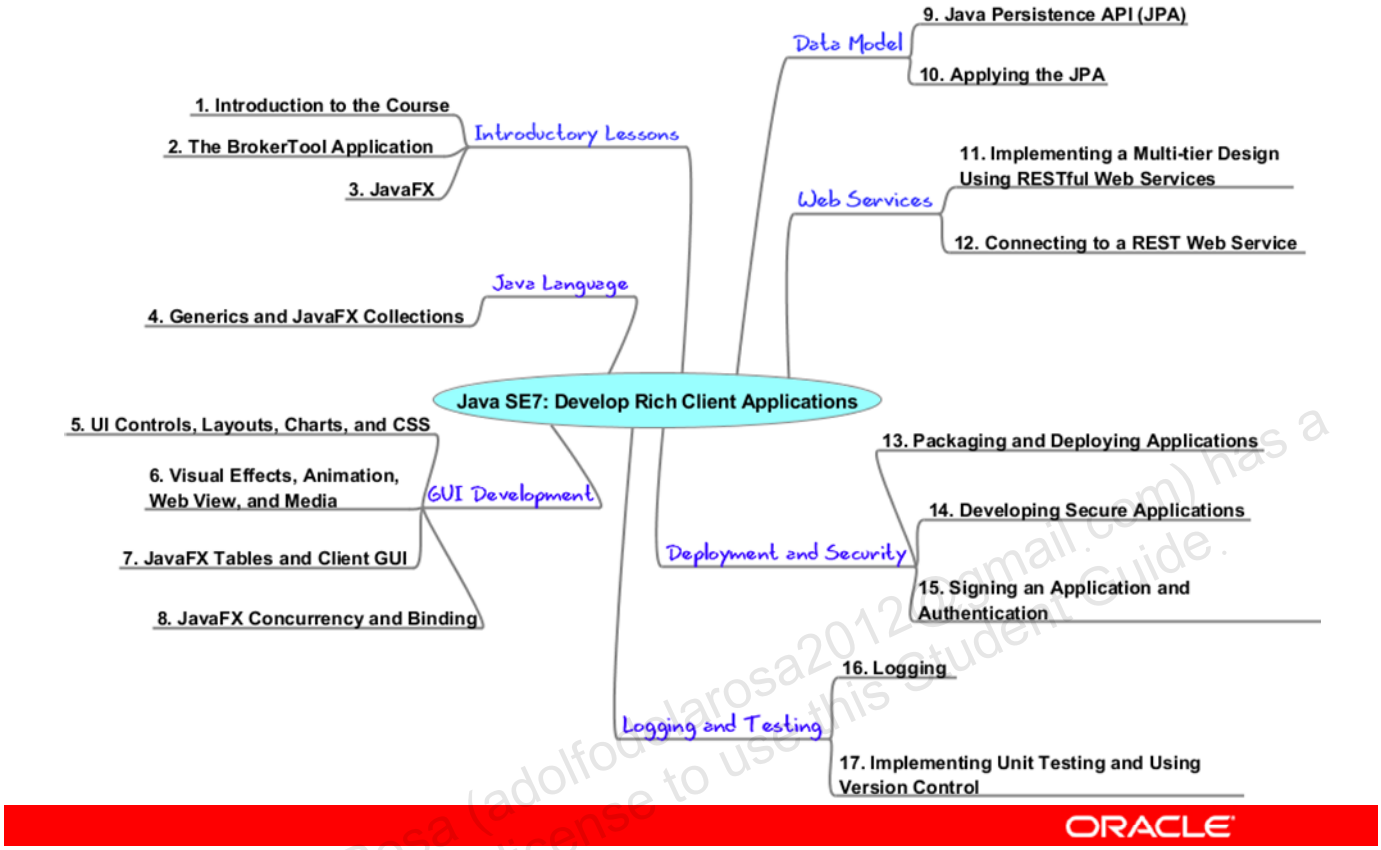
- A stand-alone executable application
- Contains a user interface that has controls and/or forms
- Is deployable from the desktop or web
- Connects to a database and server back end
- Is typically independent of operating system

Things to consider as you develop a rich client application are:

- Application and architecture requirements
- User interface design
- Data accessibility
- Available development tools
- Security
- Localization and accessibility
- Deployment options

Throughout this course, you develop a rich-client interface that connects to the database and server back end that we have created for you.

Course Roadmap



This course is divided into the following units:

- **Introductory Lessons:** These lessons introduce the course, applications, and JavaFX.
- **Java Language:** This lesson is an overview of generics and JavaFX collections.
- **GUI Development:** These lessons cover the rich client development.
- **Data Model:** These lessons introduce and apply the Java Persistence API (JPA).
- **Web Services:** These lessons provide an overview of RESTful web services.
- **Deployment and Security:** These lessons explain how to deploy and secure your application.
- **Logging and Testing:** These lessons cover logging and unit testing.

Schedule

- Day One
  - Lesson 1: Introduction to the Course
  - Lesson 2: The BrokerTool Application
  - Lesson 3: JavaFX
  - Lesson 4: Generics and JavaFX Collections
- Day Two
  - Lesson 5: UI Controls, Layouts, Charts, and CSS
  - Lesson 6: Visual Effects, Animation, Web View, and Media

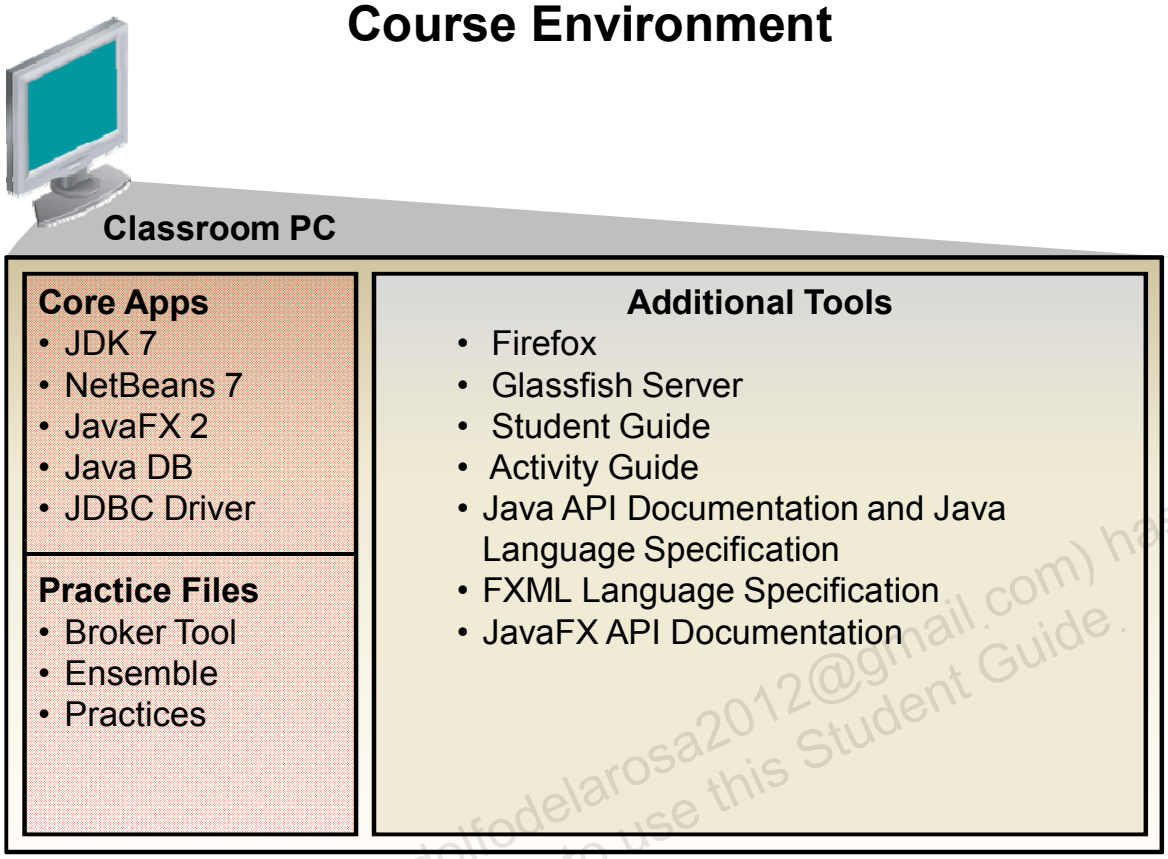
Schedule

- Day Three
  - Lesson 7: JavaFX Tables and Client GUI
  - Lesson 8: JavaFX Concurrency and Data Binding
  - Lesson 9: Java Persistence API (JPA)
- Day Four
  - Lesson 10: Applying the JPA
  - Lesson 11: Implementing a Multi-tier Design with RESTful Web Services
  - Lesson 12: Connecting to a RESTful Web Service
  - Lesson 13: Packaging and Deploying Applications

Schedule

- Day Five
  - Lesson 14: Developing Secure Applications
  - Lesson 15: Signing an Application and Authentication
  - Lesson 16: Logging
  - Lesson 17: Implementing Unit Testing and Using Version Control

Course Environment



In this course, the following products are preinstalled for the lesson practices:

- **JDK 7:** The Java SE Development Kit
- **Firefox:** A web browser is used to view the HTML documentation (Javadoc.
- **NetBeans IDE:** The NetBeans IDE is a free and open-source software development tool for professionals who create enterprise, web, desktop, and mobile applications.
- **JavaFX 2:** A tool used for creating GUIs
- **Java DB:** A database installed with NetBeans that can be used for creating databases. It is included in the JDK bundle.
- **JDBC Driver:** A database driver installed with NetBeans
- **Practice Files:** BrokerTool is the application you develop throughout the course. In addition, you work with JavaFX sample applications such as Ensemble. You use practice files to complete the practices for each lesson.
- **Glassfish Server:** This is an open-source server that is used to deploy applications.
- **Student Guide:** The guide has all of the materials that are discussed in class. In addition, the guide includes two appendixes that provide additional information about NetBeans IDE.

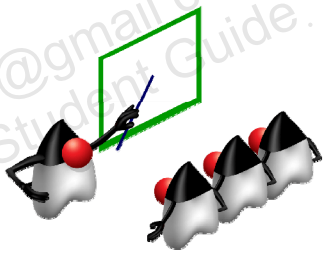
- **Activity Guide:** These are resources to use during the practice portions of the course.
- **Java API Documentation and the Java Language Specification:** The API documentation is the application programming interface specification, and the language specification describes specific language uses.
- **FXML Language Specification**
- **JavaFX API Documentation**

Quiz

a. What is your name?
b. What do you do for a living, and where do you work?
c. Why are you interested in Java?

Facilities in Your Location

- Enrollment, registration, sign-in
- Badges
- Parking
- Phones
- Internet
- Restrooms
- Labs
- Lunch
- Kitchen/snacks
- Hours
- Materials (paper, pens, and markers)

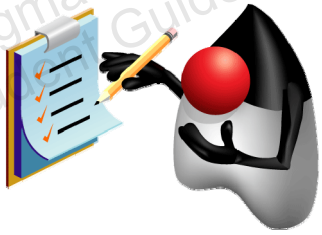


ORACLE

Summary

In this lesson, you reviewed the course objectives and the tentative class schedule. You met your fellow students, and you saw an overview of the course outline and computer environment that you will use during the course.

Enjoy the next five days of *Java SE 7: Develop Rich Client Applications*.



2

Introduce Rich Client Applications

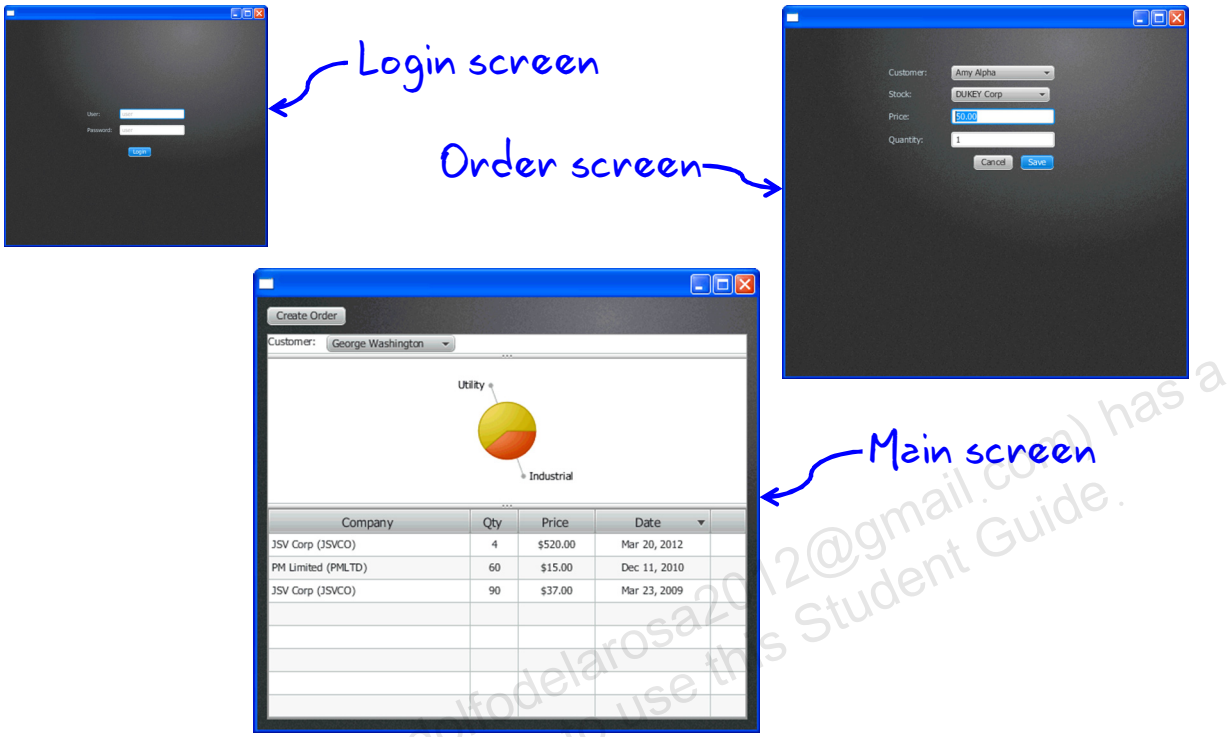
ORACLE

Objectives

After completing this lesson, you should be able to:

- Describe an overview of the BrokerTool application
- Explain the problem statement of the BrokerTool project

BrokerTool Application Overview



The BrokerTool application is a rich client, three-tier application that you will develop in this course. BrokerTool is Java technology-based, interactive, client-server system for creating, updating, and viewing customer and stock information that is contained in the database.

The application has been developed using Java, JavaFX, JDBC, and JavaDB.

The Login screen includes layout containers, text fields, labels, and a button.

The Order screen includes layout containers, combination boxes, labels, text fields, and buttons.

The Main screen includes buttons, labels, a combination box, an animated chart, a smart table , and layout containers.

BrokerTool Application Customer Problem Statement: Requirements

Functional requirements:

- Buy, sell, and update stocks for customers in the database by using the graphical user interface (GUI).
- Add customers to and remove customers from the database.
- Modify customer's name and address.
- View the current price of any stock in the database.

You are a working with a customer who wants you to develop an application based on the functional requirements. If you had enough time during this course, you would implement all of the features listed above. However, we have scaled back on the features list in the application that you develop during this course.

BrokerTool Application Customer Problem Statement: Resources & Constraints

Project resources:

- A server machine, on which the Java DB database is installed
- Multiple client graphic workstations on which the JDK is installed

Project constraints:

- Database queries should use the customer's name or a unique stock symbol.
- Some initial work was completed by the consultant.

ORACLE

The customer's requirement is that the application will post data in a JavaDB that is hosted on a server machine. In the case of this course, the server is hosted locally on your student machine.

BrokerTool Application: Customer Table

The `Customer` Table

Field Name	Type	Comment
customer_ID	int	Primary Key
account	char(15)	
fullname	char (255)	
address	char (255)	
broker_ID	int	

BrokerTool Application: Shares Table

The `Shares` Table

Field Name	Type	Comment
shares_id	int	Primary Key
symbol	char (8)	
quantity	int	
customer_id	int	
purchaseprice	double	
purchasedate	timestamp	

BrokerTool Application: Stock Table

The `Stock` Table

Field Name	Type	Comment
symbol	char (8)	Primary Key
stockname	char (255)	
sector	char (255)	

BrokerTool Application: Broker Table

The `Broker` Table

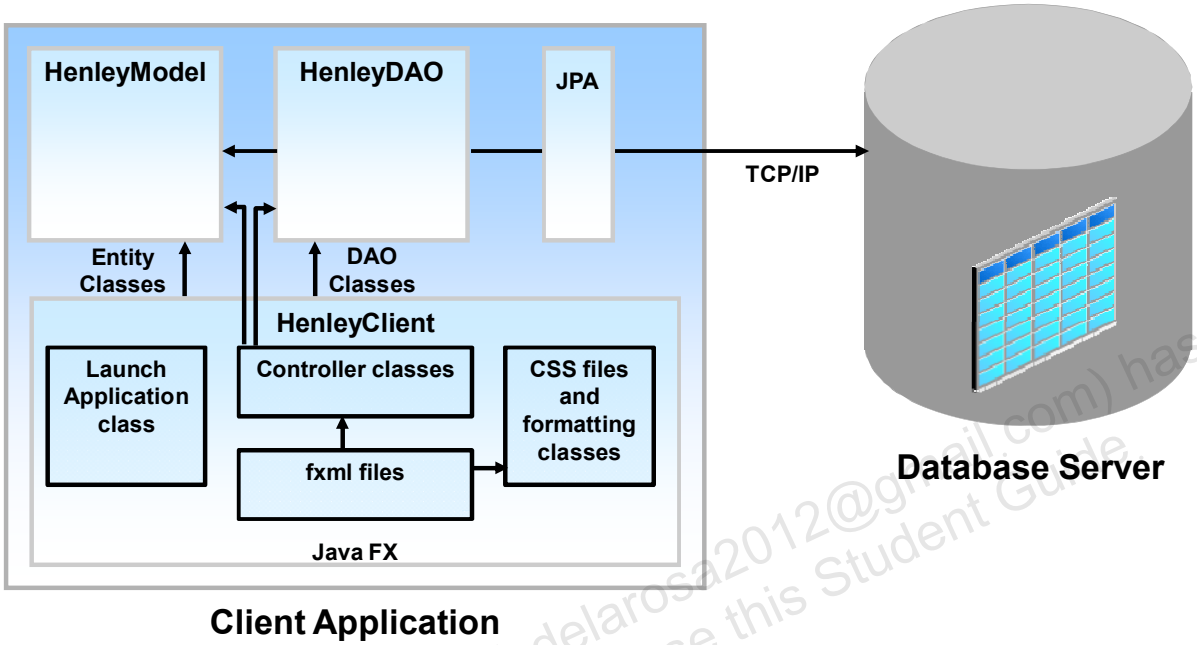
Field Name	Type	Comment
broker_id	int	Primary Key
brokername	char (255)	
address	char (255)	

Your Assignment

You need to enhance the BrokerTool application to a full-fledged Portfolio Manager application using the following technologies:

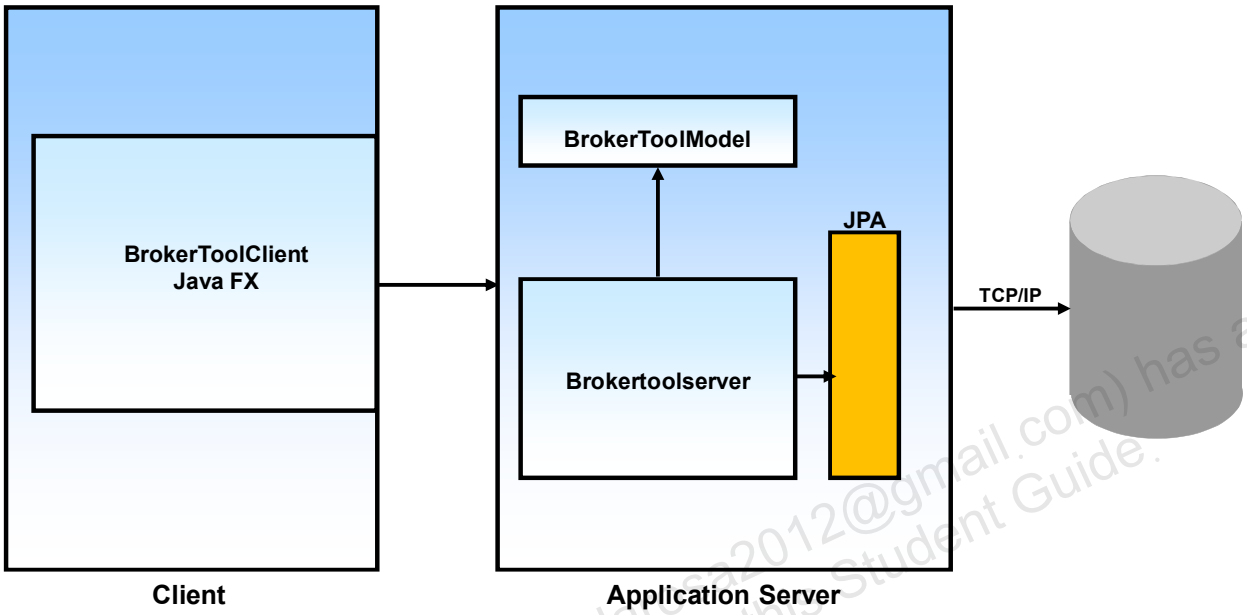
- NetBeans IDE with Java EE
  - JavaSE 7
  - JavaFX 2
  - JavaDB
  - Glassfish Server
- JPA
- JUnit for testing
- Logging API (Log4j or Java logging APIs)

Two-Tier Application Design



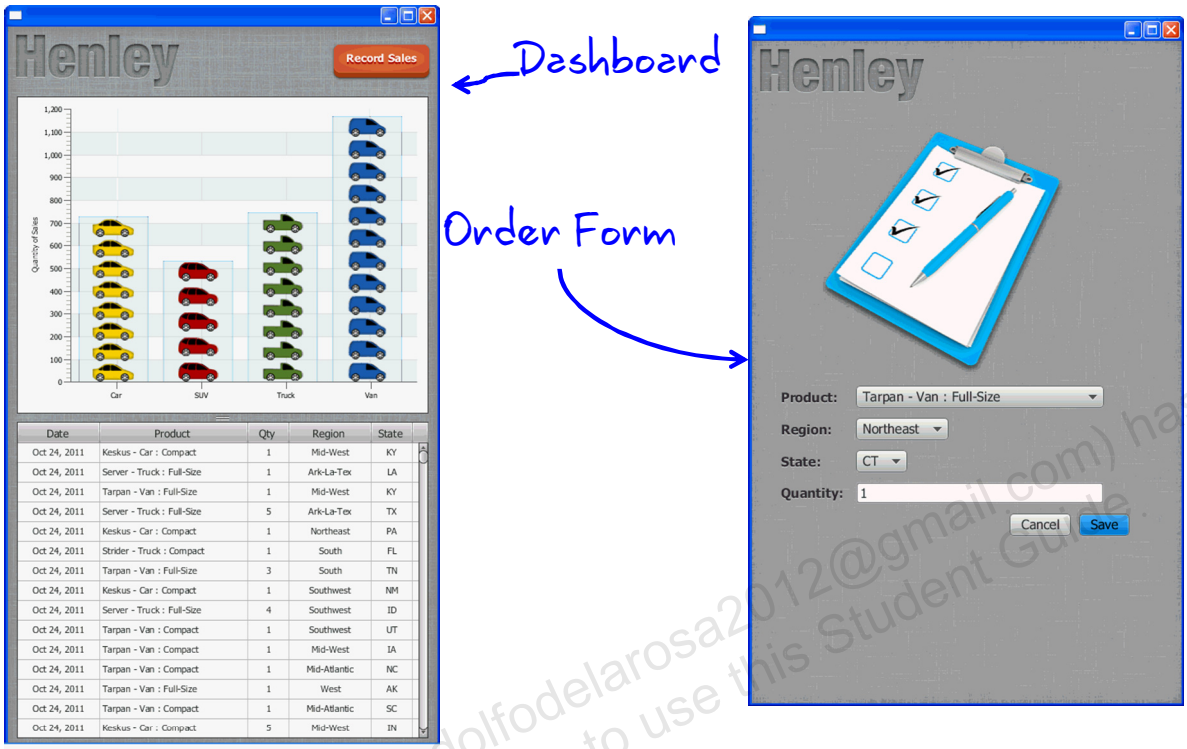
The figure in the slide shows the relationship between the model, server, and client projects. Later in the course, you will learn about multi-tier applications. You get a chance to create a two-tier application similar to the drawing in the slide.

BrokerTool Application Design: Three-Tier



Later in the course, you will learn about multi-tier applications. You get a chance to create a three-tier application similar to the drawing in the slide.

HenleyApp Car Sales Application



The HenleyApp car sales application is an example of an enterprise-level application that will be used throughout this course by the instructor for example purposes. The application includes a main screen that is used as a dashboard to display current car sales and an order form that is used to register car sales. The Henley Car Sales Application includes:

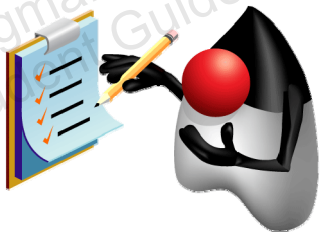
- FXML
- Tables
- Threading
- Layout
- Charts
- CSS
- UI Controls
- Animation
- Custom cells

Sales person can log sales for the product type, region, state, and quantity. The data is logged in the database, and the dashboard reflects the latest sale.

Summary

In this lesson, you should have learned how to:

- Describe an overview of the BrokerTool application
- Explain the problem statement of the BrokerTool project



Lesson 2 Practice Overview: Set up environment and use BrokerTool

This practice covers the following topics:

- 2-1: Set up the development environment
- 2-2: Run the BrokerTool application
- 2-3: Run the JavaFX Ensemble sample application

JavaFX

ORACLE

Objectives

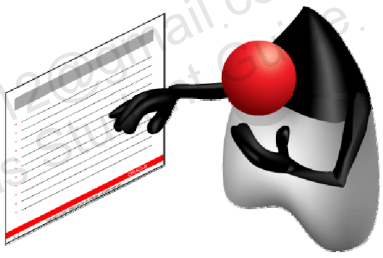
After completing this lesson, you should be able to:

- Describe the features of JavaFX
- Identify the features of the JavaFX scene graph
- Describe the JavaFX development tools
- Describe how JavaFX is integrated into a Java application
- Use FXML and determine when to use it in an application
- Download and install JavaFX and related samples

ORACLE

Topics

- Describe the features of JavaFX
- Describe a JavaFX application
- Download and install JavaFX and related samples



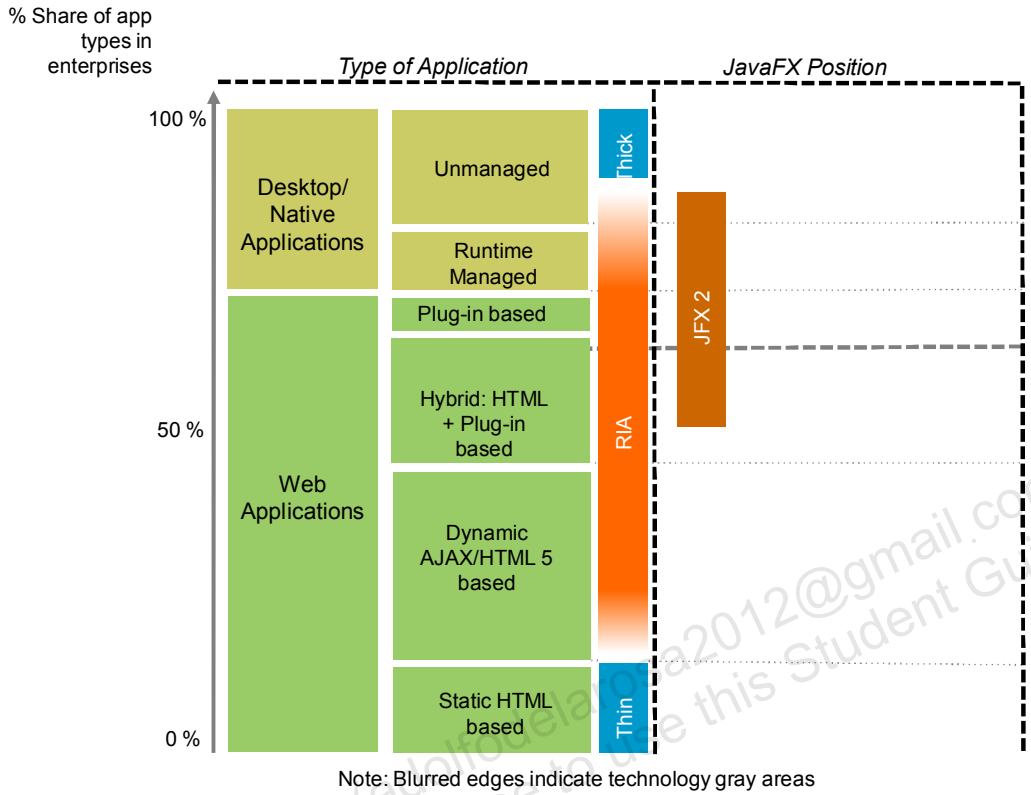
JavaFX Features

- Integrated with Java SE 7
- More than 50 charts, forms, and layout components
- Platform includes FXML, which is an alternate language to create a UI.
- Integrates JavaFX into Swing applications
- Improved graphics engine
- Integrates with scripting languages such as Groovy, JRuby, and Scala

ORACLE

- Java APIs for JavaFX: JavaFX 2 applications are completely developed in Java.
- JavaFX is integrated into existing Java libraries.
- Use your favorite Java development tools.
- Use popular JVM-based scripting languages such as Groovy, JRuby, and Scala.
- Develop and maintain complex user interfaces easily.
- Web-rendering engine
- Mix and match native Java capabilities and the dynamic capabilities of web technologies in your applications.
- High-performance, hardware-accelerated graphics pipeline
- High-performance media engine
- Play back video and audio content in popular formats within your application.

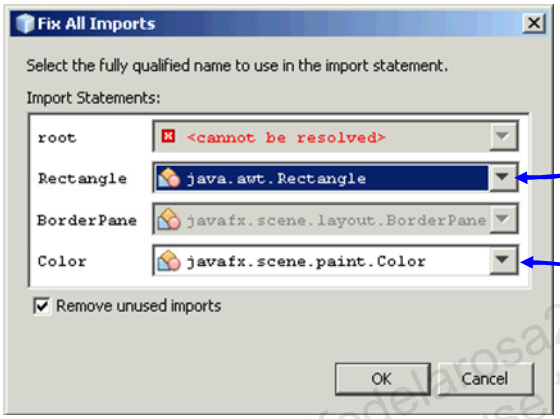
Overview of JavaFX in the Enterprise



In the realm of enterprise application, JavaFX is a rich client that sits between a web application and a desktop application. Other tools that could fit in this space include Flash, Flex, Oracle ADF, GWT, Silverlight, and others.

Swing

- Swing has been updated in Java SE 7
- You can mix Swing and JavaFX in an application (although we do not cover such mixing in this course).
- Do not confuse Swing, AWT, and JavaFX APIs.



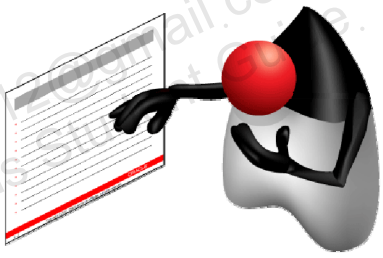
Swing is still available and supported in Java SE 7. As you develop JavaFX applications, you have to choose which API to use when you create objects. NetBeans enables you to "Fix Imports" instead of typing the package statements. The image in the slide shows that you are given choices that include JavaFX and AWT. We use JavaFX throughout the course, so be sure to choose the JavaFX packages.

Swing is fully supported in JDK 7 and has the following enhancements, including:

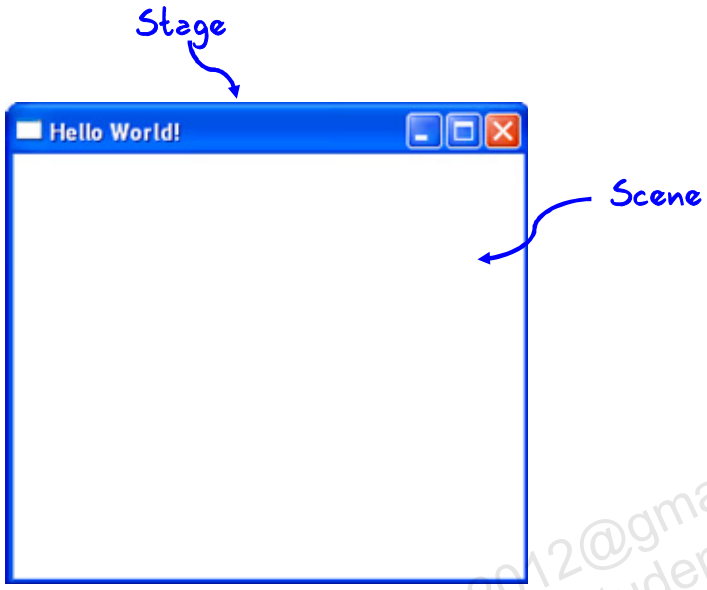
- JLayer class
- Nimbus Look and Feel
- Heavyweight and Lightweight components
- Shaped and translucent windows
- Hue-Saturation-Luminance (HSL) Color Selection in JColorChooser Class

Topics

- Describe the features of JavaFX
- Describe a JavaFX application
- Download and install JavaFX and related samples



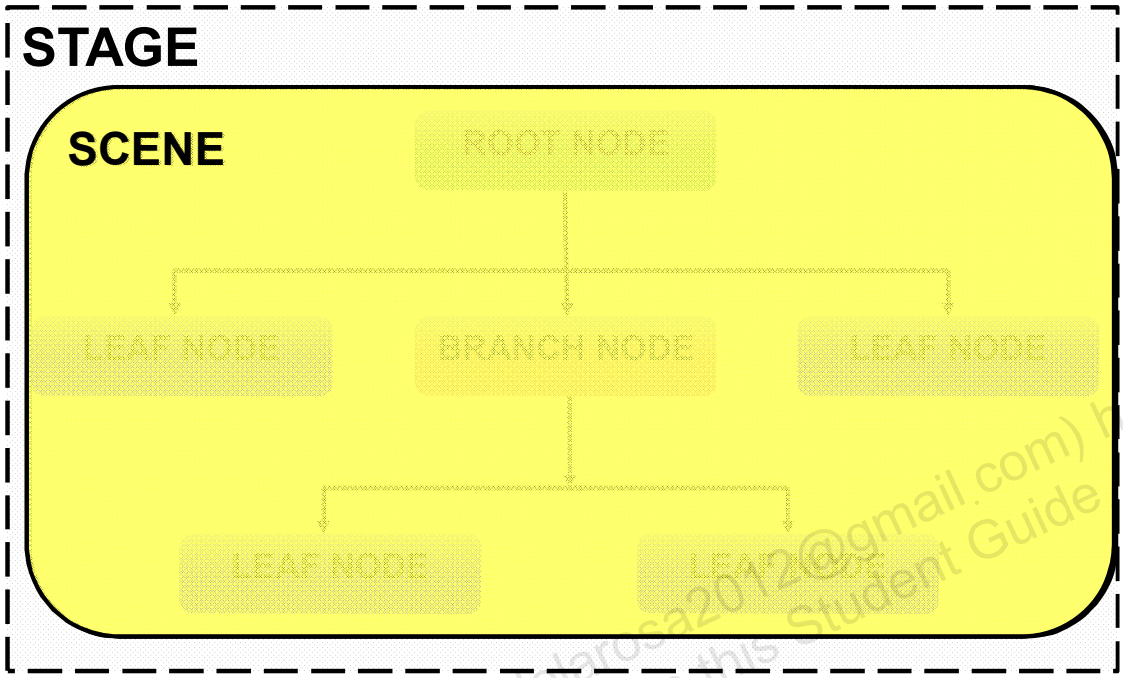
Stage and Scene



The image is an example of an application with a stage and a scene. Even applications embedded in a browser have a stage in the application code. A stage can be decorated with a window title or not, but there is always a stage.

Notice that the stage in the example is constructed by the platform, which in this case is a Windows XP platform.

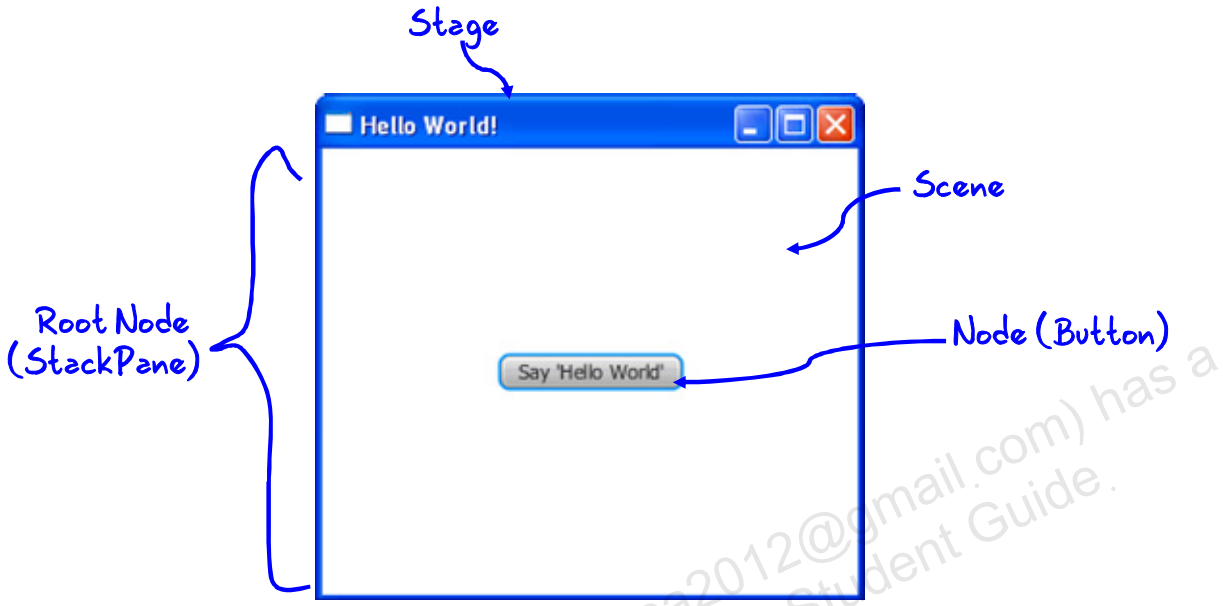
JavaFX Scene Graph: Stage and Scene



The stage (`javafx.stage.Stage`) is the top-level GUI container for all graphical objects, and the scene is the base container. The primary stage is constructed by the platform. Additional stage objects may be constructed by the application. Stage objects must be constructed and modified on the JavaFX Application Thread.

The JavaFX `Scene` class (`javafx.scene.Scene`) is the container for all content in a scene graph.

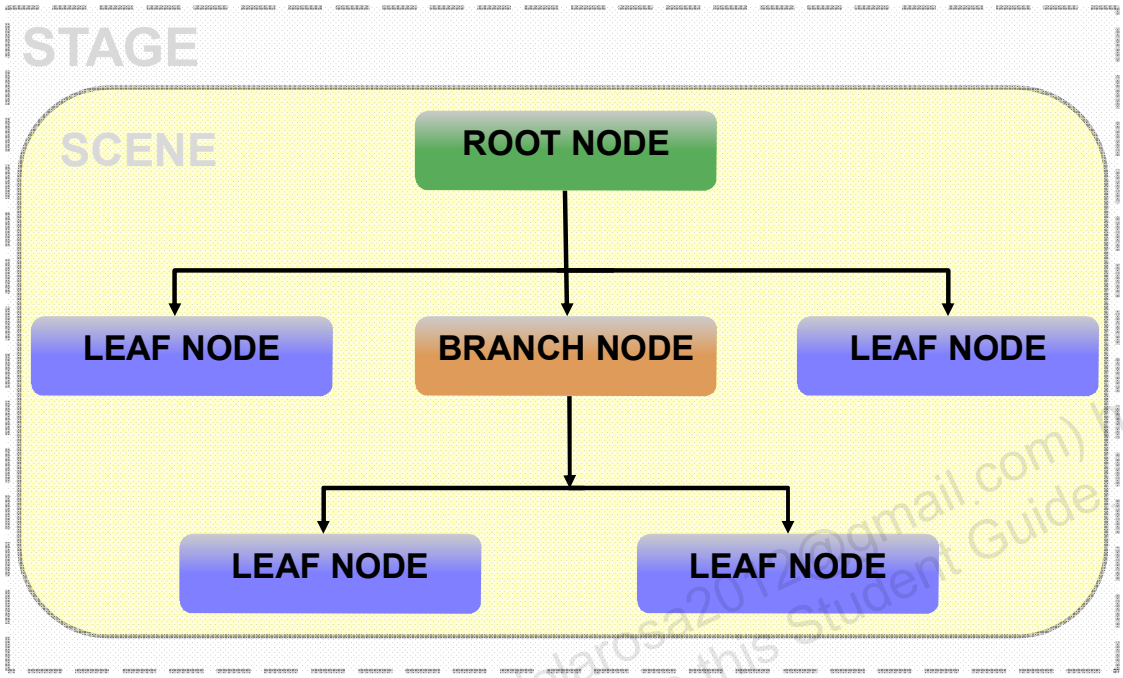
Stage, Scene, and Nodes



Here is an example of an application with a stage, a scene, and some nodes. The nodes in the example include a StackPane and a button. A StackPane is a layout container, and a button is a component.

These are covered in the lesson titled "*Visual Effects, Animation, Web View, and Media*."

JavaFX Scene Graph: Nodes Hierarchy

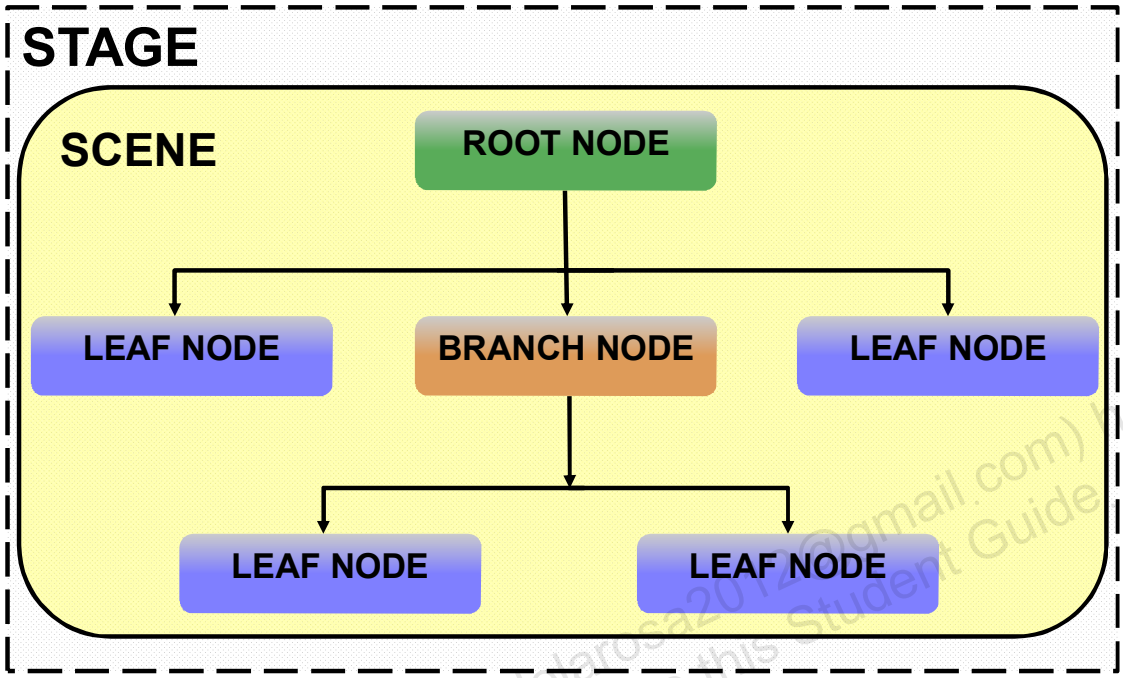


The individual items held within the JavaFX scene graph are known as *nodes*. Each node is classified as one of the following:

- A branch node (or parent, meaning that it can have children)
- A leaf node

The top node in the tree is called the *root node*, and it does not have a parent.

JavaFX Scene Graph



A scene graph is a tree data structure, most commonly found in graphical applications and libraries such as vector editing tools, 3D libraries, and video games.

The JavaFX scene graph API makes graphical user interfaces easier to create, especially when complex visual effects and transformations are involved.

The JavaFX scene graph is a retained-mode API, meaning that it maintains an internal model of all graphical objects in your application. At any given time, it knows what objects to display, what areas of the screen need repainting, and how to render it all in the most efficient manner. Instead of invoking primitive drawing methods directly, you use the scene graph API and let the system automatically handle the rendering details. This approach significantly reduces the amount of code that is needed in your application.

Rich Client Application

The image in the slide is the Data App sample, which is a rich client developed using JavaFX. The labels show the various parts of the scene graph of the application.

The scene is the main window of the application, and each tab is displayed in that scene when it is clicked.

Each tab can be considered to be a scene graph. On each tab, there are nodes such as charts, tables, labels, buttons, and more.

Minimum JavaFX Application

Basic Java application with enabled JavaFX features

```
1  package javafxapplication;

                                        Code creates a stage only.

2  import javafx.application.Application;
   import javafx.stage.Stage;

3  public class JavaFXApplication extends Application {


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

5     @Override public void start(Stage primaryStage) {

6         primaryStage.show();
      }

   }
```

ORACLE

A JavaFX application is a basic Java application with enabled JavaFX features. The minimum code needed to run a JavaFX application consists of the following:

- An extension of the `javafx.application.Application` class
- A `main()` method that calls the `launch()` method an override of the `start()` method
- A primary stage that is visible

The following refers to the code sample:

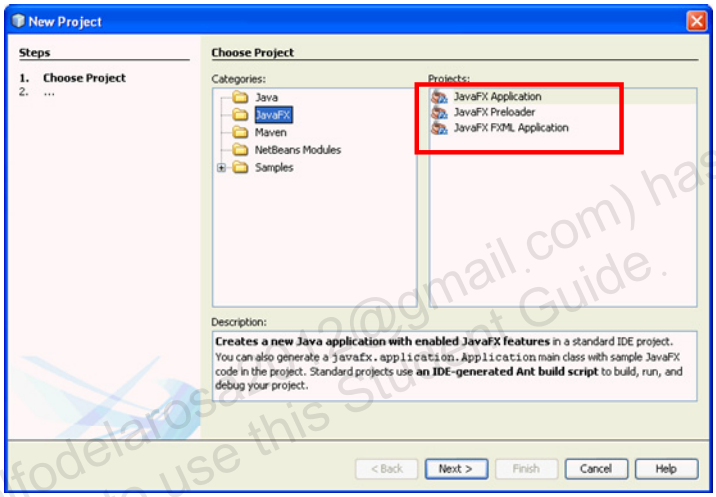
1. Package name
2. Package imports for `javafx.stage.Stage` and `javafx.application.Application`
3. The class you create inherits all the features and functionality of the `Application` class.
4. `main()` method, which calls the `launch()` method. As a best practice, the `launch()` method is the only method that is called by the `main()` method.
5. The `JavaFXApplication` class overrides the abstract `start()` method in the `Application` class. The `start()` method takes as an argument the primary stage for the application.
6. The final line of code makes the stage visible.

If you run this code, you see a stage without a scene.

JavaFX Application Types

Three types of JavaFX applications:

- JavaFX
- JavaFX FXML
- JavaFX preloader



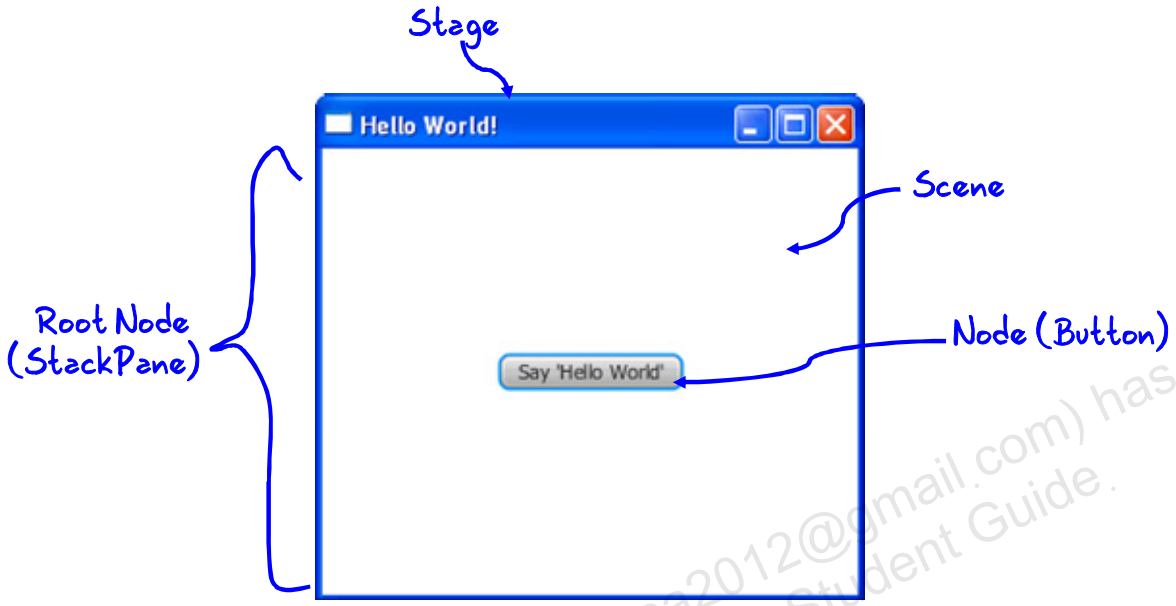
When you create a new JavaFX application by using NetBeans, you have a choice of three options. Each option creates a simple, prebuilt application.

JavaFX: Uses traditional Java syntax with JavaFX APIs

FXML: FXML is new for JavaFX 2. FXML is an XML-based declarative markup language for defining the user interface in a JavaFX application. FXML is well suited for defining static layout such as forms, controls, and tables. With FXML, you can also construct layouts dynamically by including a script.

JavaFX preloader: Used in deployment

Simple JavaFX Application



The JavaFX and FXML code examples in the next several slides represent the application shown in this slide. The application includes a stage, a scene, a button with text, and a layout container.

JavaFX Java Class

```
public class JavaFXApplication extends Application {

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Create Stage

Create Button (leaf node)

Create StackPane (root node)

Set the Scene

ORACLE

The code shown in the example is a default JavaFX application that you create in the NetBeans IDE. This example creates the application shown in slide 22.

The next three slides are the FXML equivalent of the same application.

JavaFX FXML

FXML is an XML-based declarative markup language

- Defines static layout such as forms, controls, and tables
- Constructs layouts dynamically by including a script

```
<AnchorPane>
    <children>
        <Button/>
        <Label/>
    </children>
</AnchorPane>
```

Group Node

Child Nodes

FXML offers the following advantages:

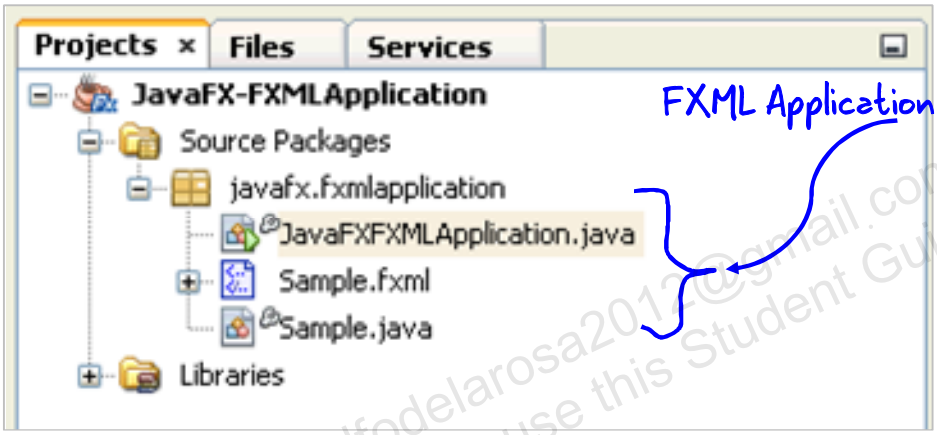
- FXML is based on XML and is familiar to most developers, especially web developers and developers using other RIA platforms.
- FXML is not a compiled language. You do not need to recompile the code to see the changes you make.
- FXML makes it easy to see the structure of your application's scene graph because of its hierarchical structure. This, in turn, makes it easier for members of a development team to collaborate on the application.

Note: The `<children>` and `</children>` tags are optional, but you can use them to improve the readability of your code.

JavaFX FXML Application Format

Three required files:

- *Applicationclass*.java
- Sample.java
- Sample.fxml



When you create a new JavaFX FXML application by using NetBeans, three files are created:

- ***Applicationclass*.java**: Contains code for creating the application scene and stage and for launching the application. The following line is especially important to understand:

```
Parent root =
FXMLLoader.load(getClass().getResource("Sample.fxml"));
```

  The `FXMLLoader.load()` method loads the object hierarchy from the resource file `Sample.fxml` and assigns it to the variable named root.
- **Sample.fxml**: This file is where you build the user interface. The NetBeans IDE automatically populates this file with markup for an Anchor Pane layout, which includes Button and Label components.
- **Sample.java**: This file is the controller file. NetBeans automatically populates this file so that it defines the action event for the button in the `Sample.fxml` file.

JavaFX FXML Application Format:
JavaFXFXMLApplication.java

```
public class JavaFXFXMLApplication extends Application {

    public static void main(String[] args) {
        Application.launch(JavaFXFXMLApplication.class,
    args);
    }
    @Override
    public void start(Stage stage) throws Exception {
        Parent root =
    FXMLLoader.load(getClass().getResource("Sample.fxml")
    );
        stage.setScene(new Scene(root));
        stage.show();
    }
}
```

Create Stage

Loads contents of Scene and makes Sample.fxml root

Set the Scene

In this example, the `JavaFXFXMLApplication` class is created. This is one of three classes required for a JavaFX FXML application.

By default, NetBeans generates code that creates the scene and stage and launches the application. This class functions as the application's main class.

The `FXMLLoader.load()` method loads the object hierarchy from the resource file `Sample.fxml` and assigns it to the variable named root.

This class is the Model in the MVC pattern.

JavaFX FXML Application Format: `Sample.fxml`

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
xmlns:fx=http://javafx.com/fxml
fx:controller="javafx.fxmlapplication.Sample">
    <children>
        <Button id="button" layoutX="126" layoutY="90"
         text="Click Me!" onAction="#handleButtonAction"
         fx:id="button" />
        <Label id="label" layoutX="126" layoutY="120"
         minHeight="16" minWidth="69" prefHeight="16"
         prefWidth="69" fx:id="label" />
    </children>
</AnchorPane>
```

AnchorPane is root node

Child Nodes

The `Sample.fxml` file holds the content that sits in the stage and scene and functions as the application interface. By default, NetBeans creates a button with an action and a label. Typically, you will delete this automatically generated code and replace it with your own code. In addition, you will rename the file using an intuitive name that describes its function within the application. In most applications, this file will become the user interface. The class is the View in the MVC pattern.

JavaFX FXML Application Format: `Sample.java`

```
public class Sample implements Initializable {

    @FXML
    private Label label;

    @FXML
    private void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
        label.setText("Hello World!");
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        // TODO
    }
}
```

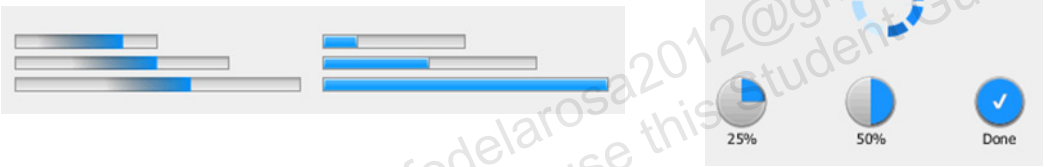
Declares this class as Controller class

Button event

By default, NetBeans creates a class that acts as the Controller class. Typically, you will delete this automatically generated code and replace it with your own code. In addition, you will rename the file using an intuitive name that describes its function within the application. This file will control `Sample.fxml`, which is the user interface. The name of this class should match the name of the fxml view class. This class acts as the Controller in the MVC pattern.

JavaFX Preloader

- A preloader is a small application that is started before the main application to customize the startup experience.
- JavaFX has two types of preloaders:
  - Default (exists in the runtime)
  - Custom (created by you)
- A preloader extends the `javafx.application.Preloader` package and can use a progress bar or progress indicator.



Preloaders improve the application startup experience. Use of a preloader can help to reduce perceived application startup time by showing some content to the user earlier, such as a progress indicator or login prompt.

A preloader application can also be used to present custom messaging to the user. For example, you can explain what is currently happening and what the user will be asked to do next, such as grant permissions to the application. Or you can create a preloader to present custom error messaging.

Not every application needs a preloader. For example, if the size of your application is small and does not have special requirements such as permissions, then it probably starts quickly. Even for larger applications, the default preloader included with the JavaFX Runtime can be a good choice, because it is loaded from the client machine rather than the network.

Preloaders are described in the lesson titled "*Packaging and Deploying Applications*."

Quiz

As a standard practice, which of the following is the only method called by the `main()` method?

a. `start()`
b. `initialize()`
c. `launch()`
d. `stop()`

Answer: c

a. The `main` class overrides the abstract `start()` method and calls the `start()` method.
b. The `initialize()` method is not called in the `main` class.
c. The `launch()` method is typically called by the `main()` method.
d. The `stop()` method is not called in the `main()` method.

Quiz

Which node in the JavaFX Scene Graph is always the top node?

a. Top
b. Simple
c. Graphic
d. Root

ORACLE

Answer: d

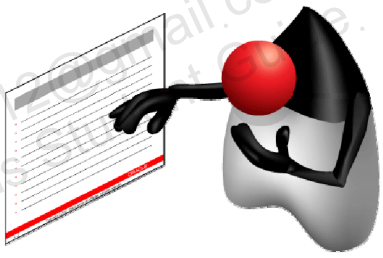
a. A Top node is not a type of node.
b. A Simple node is not a type of node.
c. Graphic is not a type of node.
d. Root nodes are the top node in the scene graph.

Practice 3: Overview

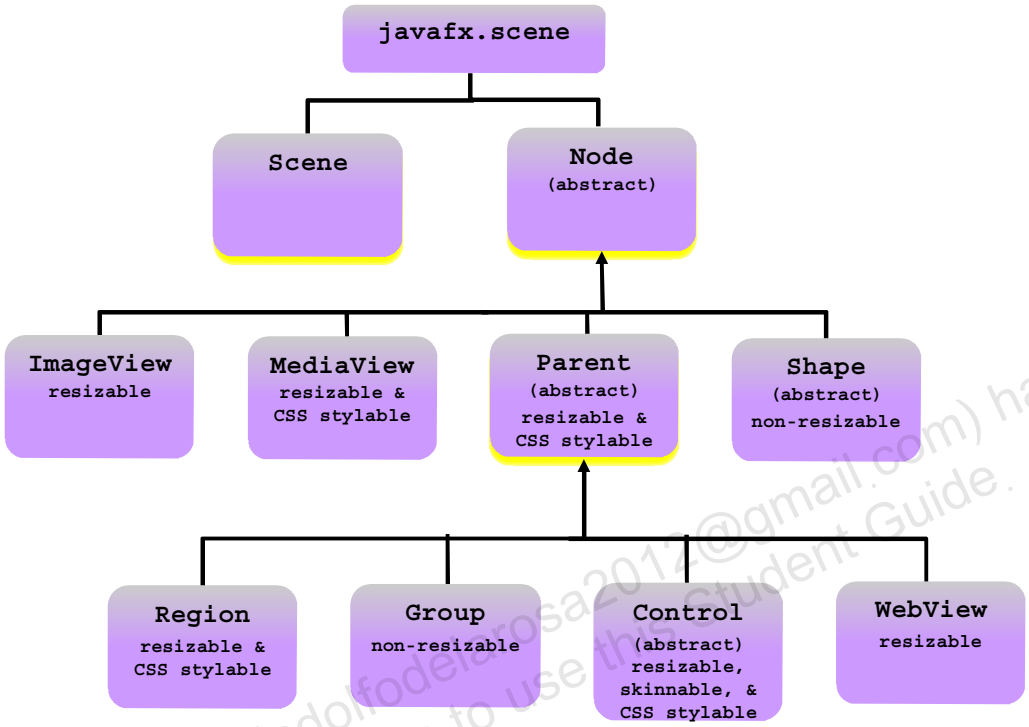
- 3-1: Creating a Login Window by Using JavaFX
- 3-2: Creating an FXML Login Window

Topics

- Describe the features of JavaFX
- Describe a JavaFX application
- Download and install JavaFX and related samples



Scene Graph API



The `javafx.scene` package defines more than a dozen classes, but three classes in particular are most important when it comes to learning how the API is structured.

- `Node`: The abstract base class for all scene graph nodes
- `Parent`: The abstract base class for all branch nodes (This class directly extends `Node`.)
- `Scene`: The base container class for all content in the scene graph

These base classes define important functionality that is subsequently inherited by subclasses, including paint order, visibility, composition of transformations, support for CSS styling, and so on.

`Parent` is the base class for all nodes that have children in the scene graph and is unique to JavaFX. This class handles all hierarchical scene graph operations, including adding and removing child nodes, marking branches as dirty for layout and rendering, picking, bounds calculations, and executing the layout pass on each pulse.

There are three direct concrete `Parent` subclasses:

- Group effects and transforms to be applied to a collection of child nodes
- `Region` class for nodes that can be styled with CSS and layout children
- Base `Control` class for high-level skinnable nodes designed for user interaction

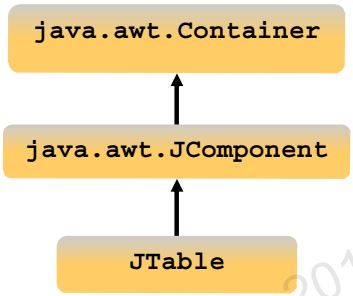
The JavaFX Scene class `javafx.scene.Scene` is the container for all content in a scene graph. The background of the scene is filled as specified by the fill property. The application must specify the root node for the scene graph by setting the root property. If `Group` is used as the root, the contents of the scene graph will be clipped by the scene's width and height and changes to the scene's size (if the user resizes the stage) will not alter the layout of the scene graph. If a resizable node (layout `Region` or `Control)` is set as the root, the root's size will track the scene's size, causing the contents to be relayed out as necessary.

The scene's size can be initialized by the application during construction. If no size is specified, the scene automatically computes its initial size based on the preferred size of its content.

Swing Containment Hierarchy

Levels of the Swing component hierarchy include:

- Frame
- Panel
- Component



The levels of the Swing containment hierarchy are similar to the JavaFX scene graph's hierarchy.

- The frame is a top-level container. It exists mainly to provide a place for other Swing components to paint themselves. The other commonly used top-level containers are dialogs (JDialog) and applets (JApplet).
- The panel is an intermediate container. Its only purpose is to simplify the positioning of the button and label. Other intermediate Swing containers, such as scroll panes (JScrollPane) and tabbed panes (JTabbedPane), typically play a more visible, interactive role in a program's GUI.
- The button and label are atomic components that exist not to hold random Swing components but as self-sufficient entities that present bits of information to the user. Atomic components also often get input from the user. The Swing API provides many atomic components, including combo boxes (JComboBox), text fields (JTextField), and tables (JTable).

Download and Install JavaFX

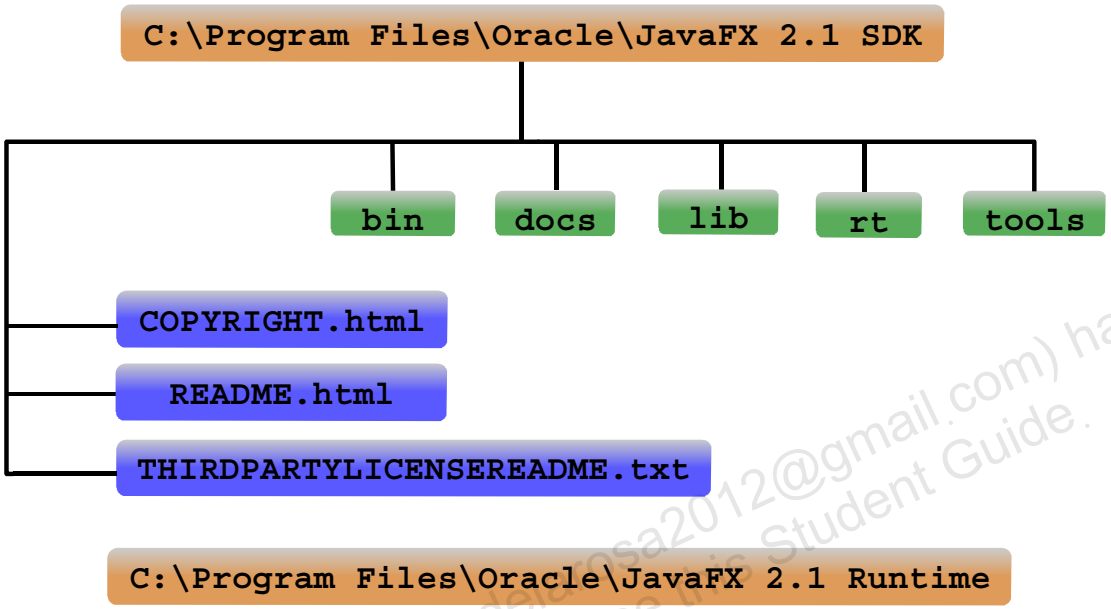
There are five options for downloading JavaFX bundles from oracle.com/javafx:

- JDK plus JavaFX SDK (includes Runtime)
- JavaFX SDK (includes Runtime)
- JavaFX Runtime only
- NetBeans with JavaFX
- Scene Builder: FXML editor

From the JavaFX downloads page, you can see there are several options for downloading JavaFX.

- The first option is to download JavaFX along with JDK 7. This option includes the JavaFX SDK and Runtime.
- The second option is the JavaFX SDK and JavaFX Runtime. The stand-alone JavaFX 2 SDK is recommended for developers using JDK 6u26 through 7u1. The JavaFX SDK includes the JavaFX libraries, the JavaFX Runtime, and project files that you need to get started. You can use the JavaFX SDK by itself or with your favorite Java IDE to leverage debugging, profiling, and other IDE features. You must have the Java SE Development Kit (JDK) 6 Update 26 or higher installed on your system if you want to use the JavaFX SDK.
- The third option is the JavaFX Runtime only. The JavaFX Runtime provides the runtime environment required for running JavaFX desktop applications and applets.
- The next option to download is NetBeans 7.1 with the JDK and JavaFX. This option enables you to start developing JavaFX applications with the NetBeans IDE for building, previewing, and debugging JavaFX applications.
- JavaFX Scene Builder (beta release) is a GUI editor that enables you to develop interfaces in JavaFX FXML.

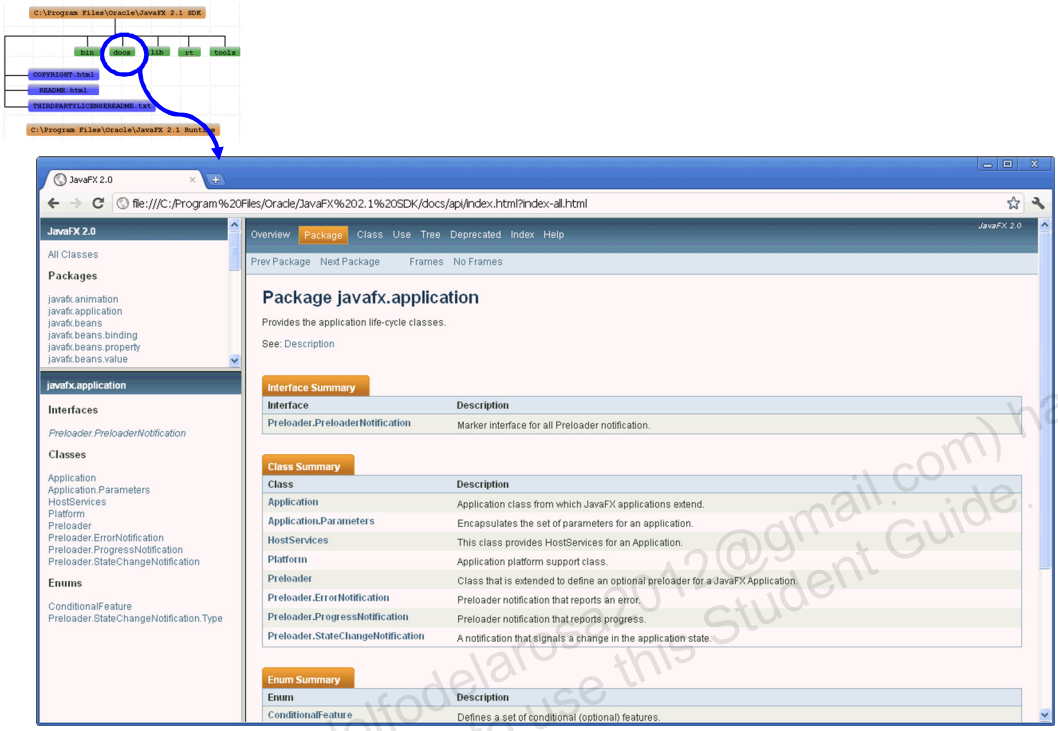
JavaFX Default Installation Directory



- `bin`/SDK build tools
- `docs`/API documentation
- **lib/`Ant`:** Tasks used by NetBeans for packaging and deployment. Developers can use the tools as well. `javafx-doclet.jar: http://docs.oracle.com/javafx/2.0/doclet/jfxpub-doclet.htm ant-javafx.jar: http://docs.oracle.com/javafx/2.0/deployment/deploy_quick_start.htm#JFXDP518`
- **`rt/`:** Contains the `jrxrt.jar` runtime file
- **`tools/Ant`:** Tasks used by NetBeans for packaging and deployment. Developers can use the tools as well. `javafx-doclet.jar: http://docs.oracle.com/javafx/2.0/doclet/jfxpub-doclet.htm ant-javafx.jar: http://docs.oracle.com/javafx/2.0/deployment/deploy_quick_start.htm#JFXDP518 (Note: this directory is maintained for backward compatibility with older versions of NetBeans.)`

- **`COPYRIGHT.html`:** Copyright information for the `readme.html` document
- **`README.html`:** Contains a link to the OTN readme page for Java SE, JavaFX Runtime, and JavaFX SDK
- **`THIRDPARTYLICENSEREADME.txt`:** License information for third-party software included in the JavaFX SDK and the documentation

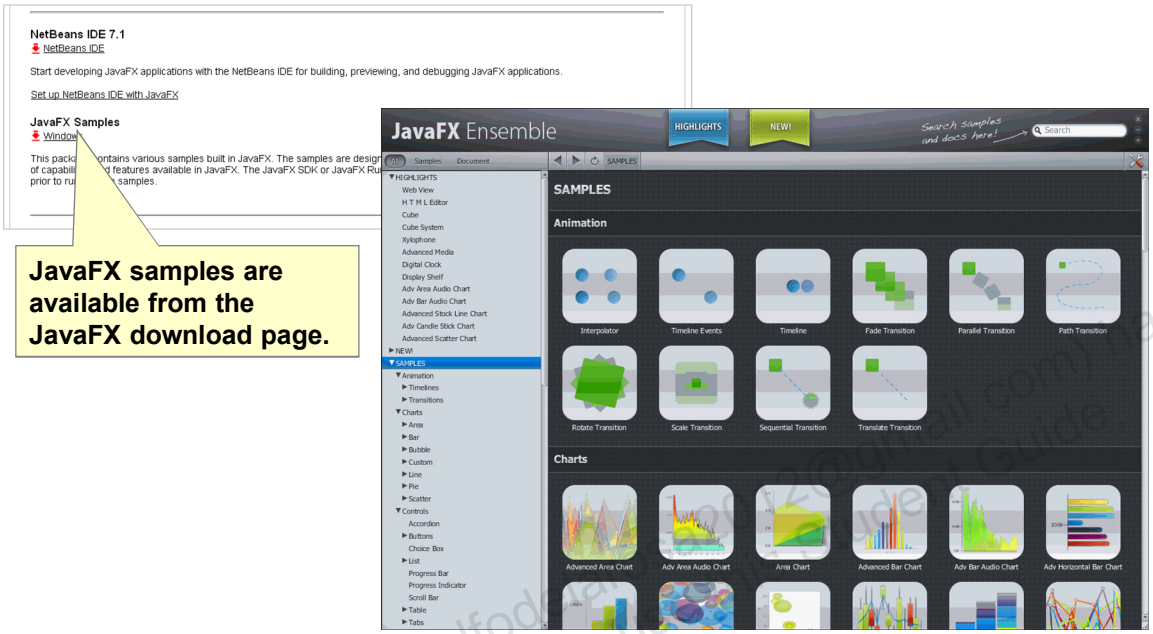
JavaFX API Documentation



The JavaFX API documentation is structured in the same manner as the Java API documentation.

Download JavaFX Samples

Download JavaFX samples from oracle.com/javafx.



JavaFX samples are a good way to get started using JavaFX. The free samples are available on the main JavaFX download page of oracle.com/javafx and are updated with each JavaFX release.

Resources

- oracle.com/javafx
  - JavaFX downloads
  - Samples and technical videos
  - Documentation
  - Blogs and forums
- Oracle Learning Library (oracle.com/oll)
  - Java and JavaFX tutorials, demos, interactive training, and OBEs (Oracle by Example)
- *Java Magazine* (by subscription)

To stay current with JavaFX and its latest releases, you can bookmark oracle.com/javafx, where you can find many JavaFX resources, including:

- Downloadable software
- Samples
- Documentation
- Blogs: http://blogs.oracle.com/javafx/
- Forums: https://forums.oracle.com/forums/forum.jspa?forumID=1385
- Technical videos: http://otn.oracle.com/javafx/

In addition, the Oracle Learning Library at oracle.com/oll has the following kinds of material:

- Tutorials
- Videos and demos
- OBEs (Oracle by Example): Java curriculum developers have developed several OBE (Oracle by Example) tutorials that describe how to use the new Swing enhancements. See http://www.oracle.com/goto/oll and search for “Java” to find tutorials, videos, and OBEs.
- Interactive learning modules

- Links to related Oracle University courses
- Subscription to *Java Magazine*:
  http://www.oracle.com/technetwork/java/javamagazine/index.html

Summary

In this lesson, you should have learned how to:

- Describe the features of JavaFX
- Identify the features of the JavaFX scene graph
- Describe the JavaFX development tools
- Describe how JavaFX is integrated into a Java application
- Use FXML and determine when to use it in an application
- Download and install JavaFX and related samples

4

Generics and JavaFX Collections

ORACLE

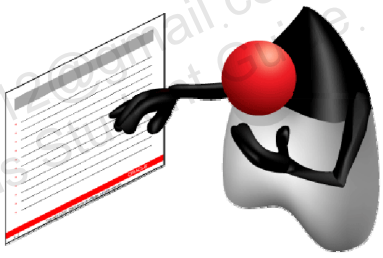
Objectives

After completing this lesson, you should be able to describe:

- Generics
- JavaFX Collections

Topics

- Generics
- JavaFX Collections



Generics

- Provide flexible type safety to your code
- Move many common errors from runtime to compile time
- Provide code that is cleaner and easier to write
- Reduce the need for casting with collections
- Are used heavily in the Java Collections API

Simple Cache Class Without Generics

```
public class CacheString {
    private String message = "";

    public void add(String message){
        this.message = message;
    }

    public String get(){
        return this.message;
    }
}
```

```
public class CacheShirt {
    private Shirt shirt;

    public void add(Shirt shirt){
        this.shirt = shirt;
    }

    public Shirt get(){
        return this.shirt;
    }
}
```


The two examples in the slide show very simple caching classes. Even though each class is very simple, a separate class is required for any object type.

Generic Cache Class

```
public class CacheAny <T>{

    private T t;

    public void add(T t){
        this.t = t;
    }

    public T get(){
        return this.t;
    }
}
```


To create a generic version of the `CacheAny` class, a variable named `T` is added to the class definition surrounded by angle brackets. In this case, `T` stands for "type" and could represent any type. As the example shows, the code has changed to use `t` instead of a specific type information. This change allows the `CacheAny` class to store any type of object.

`T` was chosen not by accident but by convention. A number of letters are commonly used with generics. Here are the conventions and what they stand for.

Note: You could use any identifier you want. These values are merely strongly suggested.

- `T`: Type
- `E`: Element
- `K`: Key
- `V`: Value
- `S`, `U`: Used if there are two, three, or more types

Generics in Action

Compare the type-restricted objects to their generic alternatives:

```
public static void main(String args[]){
    CacheString myMessage = new CacheString(); // Type
    CacheShirt myShirt = new CacheShirt();     // Type

    //Generics
    CacheAny<String> myGenericMessage = new CacheAny<String>();
    CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();

    myMessage.add("Save this for me"); // Type
    myGenericMessage.add("Save this for me"); // Generic

}
```


Note how the one generic version of the class can replace any number of type-specific caching classes. The `add()` and `get()` functions work exactly the same way. In fact, if the `myMessage` declaration were changed to generic, no changes would need to be made to the remaining code.

The example code shown in the slide can be found in the Generics project in the `TestCacheAny.java` file.

Generics with Diamond Operator

- Syntax
  - There is no need to repeat types on the right side of the statement.
  - Angle brackets indicate that type parameters are mirrored.
- Simplifies generic declarations
- Saves typing

```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
}
```


ORACLE

The diamond operator is a new feature in JDK 7. In the generic code, notice how the type definition on the right is always equivalent to the type definition on the left. In JDK 7, you can use the diamond operator to indicate that the right type definition is equivalent to the left type definition. This helps to avoid repeatedly typing redundant information.

Example: `TestCacheAnyDiamond.java`

This example works in an opposite way from a "normal" Java type assignment. In a "normal" assignment, we have the following:

`Employee emp = new Manager();` makes the `emp` object an instance of `Manager`.

But in the case of generics, we have the following:

```
ArrayList<Manager>  managementTeam = new ArrayList<>();
```

The type is determined by the left side of the expression, and not the right side.

`ArrayList` Without Generics

```
1 public class OldStyleArrayList {
2  public static void main(String args[]){
3    List partList = new ArrayList(3);
4
5    partList.add(new Integer(1111));
6    partList.add(new Integer(2222));
7    partList.add(new Integer(3333));
8    partList.add("Oops a string!");
9
10    Iterator elements = partList.iterator();
11    while (elements.hasNext()) {
12      Integer partNumberObject = (Integer) (elements.next()); // error?
13      int partNumber = (int) partNumberObject.intValue();
14
15      System.out.println("Part number: " + partNumber);
16    }
17  }
18 }
```


In the example in the slide, a part number list is created using an `ArrayList`. Using syntax prior to Java version 1.5, there is no type definition. So any type can be added to the list as shown on line 8. It is therefore up to the programmer to know what objects are in the list and what their order is. If an assumption is made that the list is only for `Integer` objects, a runtime error occurs on line 12.

On lines 10–16, with a nongeneric collection, an `Iterator` is used to iterate through the list of items. Notice that a lot of casting is required to get the objects back out of the list so you can print the data.

In the end, there is a lot of needless “syntactic sugar” (extra code) working with collections in this way.

If the line that adds the `String` to the `ArrayList` is commented out, the program produces the following output:

```
Part number: 1111
Part number: 2222
Part number: 3333
```

Generic `ArrayList`

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class GenericArrayList {
5 public static void main(String args[]){
6
7    List<Integer> partList = new ArrayList<>(3);
8
9    partList.add(new Integer(1111));
10    partList.add(new Integer(2222));
11    partList.add(new Integer(3333));
12    partList.add(new Integer(4444)); // ArrayList auto grows
13
14    System.out.println("First Part: " + partList.get(0)); // First item
15    partList.add(0, new Integer(5555)); // Insert an item by index
16
17    // partList.add("Bad Data");  // compile error now
```

ORACLE

With generics, things are much simpler. When the `ArrayList` is initialized on line 6, any attempt to add an invalid value (line 17) results in a compile-time error.

The example in the slide shows several `ArrayList` features:

- Line 12 demonstrates how an `ArrayList` grows automatically when an item is added beyond its original size.
- Line 14 shows how elements can be accessed by their index.
- Line 15 shows how elements can be inserted in the list based on their index.

Note: On line 7, the `ArrayList` is assigned to a `List` type. Using this style enables you to swap out the `List` implementation without changing other code.

Quiz

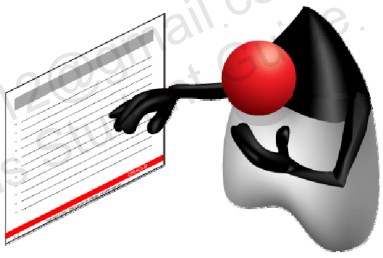
Which of the following is *not* a conventional abbreviation for use with generics?

a. T: Table
b. E: Element
c. K: Key
d. V: Value

Answer: a

Topics

- Generics
- JavaFX Collections



JavaFX Collections and Interfaces

- JavaFX Collections are an extension of the Java Collections Framework.
- JavaFX Collections are defined by the `javafx.collections` package.
- `ObservableList`: A list that enables listeners to track changes when they occur
- `ObservableMap`: A map that enables observers to track changes when they occur
- `ListChangeListener`: An interface that handles list-related events generated by an `ObservableList`

ObservableList

- Extends `javafx.beans.Observable` and `java.util.List`
- Provides a list that supports observability
- Has methods for adding or removing `ListChangeListener`
  - `addListener(ListChangeListener<?super E> listener)`: Adds a listener to this observable list
  - `removeListener(ListChangeListener<? super E> listener)`: Tries to remove a listener from this observable list. If the listener is not attached to this list, nothing happens.
- Has methods for set and retain that add or retain elements of a collection

An `ObservableList` is a list that enables listeners to track changes when they occur.

An observer may be present at most once on a particular list. If an observer is already present and `add` is called a second time, there is no effect. If a `remove` call is made on an observer that is not present, there is no effect.

ObservableMap

- Extends `javafx.beans.Observable` and `java.util.Map`
- Provides a list that supports observability
- Has methods for adding or removing `MapChangeListener`
  - `addListener(MapChangeListener<? super K,? super V> listener)`: Adds a listener to this observable map
  - `removeListener(MapChangeListener<? super K,? super V> listener)`: Tries to remove a listener from this observable map

An `ObservableMap` is a map that enables observers to track changes when they occur.

An observer may be present at most once on a particular list. If an observer is already present and `add` is called a second time, there is no effect. If a `remove` call is made on an observer that is not present, there is no effect.

`FXCollections` Classes

- `FXCollections`: A utility class that consists of static methods that are one-to-one copies of `java.util.Collections` methods
- Wrapper methods return `ObservableList` and are suitable for methods that require `ObservableList` on input.
- Utility methods are used for performance reasons. Methods are optimized to yield only limited numbers of notifications.

ObservableList from Strings

```
public class ListExample {

    public static void main(String[] args) {
        final List<String> list = new ArrayList<String>();
        final ObservableList<String> observableList =
     FXCollections.observableList(list);

        observableList.addListener(new ListChangeListener() {

            @Override
            public void onChanged(ListChangeListener.Change change) {
                System.out.println("List changed");
                System.out.println("Observable list length: " +
                     observableList.size() + "  List length: " + list.size());
            }
        });

        System.out.println("--Adding items--");
        observableList.add("First item");
        observableList.add("Second item");
        observableList.add("Third item");
        observableList.add("Fourth item");

        System.out.println("List contents: " + list.toString());
    }
}
```

The `javafx.collections.FXCollections` class is used to create and return the `ObservableList` objects. `ListChangeListener.Change` represents a change made to an `ObservableList`. The program produces the following output:

```
--Adding items--
List changed
Observable list length: 1  List length: 1
List changed
Observable list length: 2  List length: 2
List changed
Observable list length: 3  List length: 3
List changed
Observable list length: 4  List length: 4
List contents: [First item, Second item, Third item, Fourth item]
```

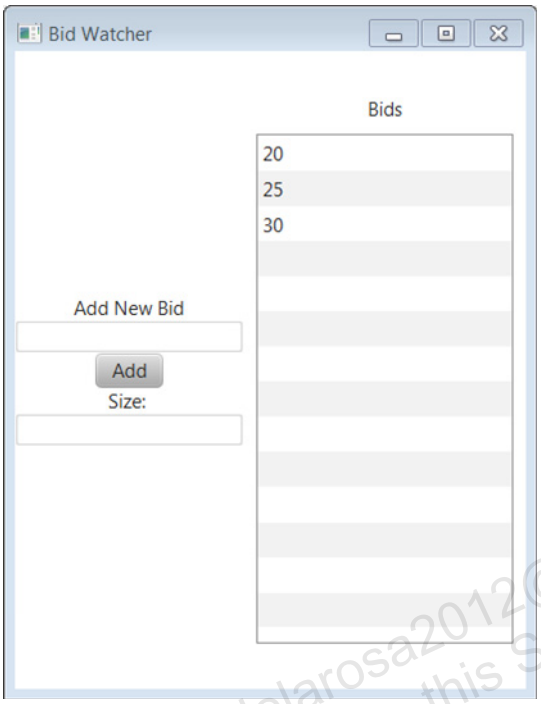
ObservableMap from Strings

```
public class MapExample {

    public static void main(String[] args) {
        final Map<String, String> map = new HashMap<>();
        final ObservableMap<String, String> observableMap =
     FXCollections.observableMap(map);

        observableMap.addListener(new MapChangeListener() {

            @Override
            public void onChanged(MapChangeListener.Change change) {
                System.out.println("Map changed!");
                System.out.println("Observable map length: " +
                    observableMap.size() + "  Map length: " + map.size());
            }
        });

        System.out.println("--Adding map items--");
        observableMap.put("1111", "Couch");
        observableMap.put("1112", "Chair");
        observableMap.put("1113", "Lamp");
        observableMap.put("1114", "Table");

        System.out.println("Map contents: " + map.toString());
    }
}
```


The `javafx.collections.FXCollections` class is used to create and return the `ObservableMap` objects. `MapChangeListener.Change` represents a change made to an `ObservableMap`. The program produces the following output:

```
--Adding map items--
Map changed!
Observable map length: 1  Map length: 1
Map changed!
Observable map length: 2  Map length: 2
Map changed!
Observable map length: 3  Map length: 3
Map changed!
Observable map length: 4  Map length: 4
Map contents: {1111=Couch, 1112=Chair, 1113=Lamp, 1114=Table}
```

Bid Example: Demo



The `BidExample` program in the `ObserveExample` project demonstrates how a `ListView` and `ListChangeListener` can be used to observe the same `ObservableList`.

Quiz

Which of the following is an interface that receives notifications of changes to an `ObservableMap`?

a. `ObservableMap`
b. `javafx.collections`
c. `MapChangeListener`
d. `ObservableList`

Answer: c

Summary

In this lesson, you should have learned how to describe:

- Generics
- JavaFX Collections

Practice 4: Overview

4-1: Adding a Second Listener to Your List

5

UI Controls, Layouts, Charts, and CSS

ORACLE

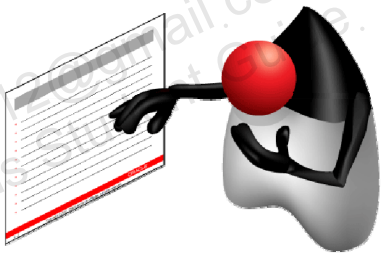
Objectives

After completing this lesson, you should be able to:

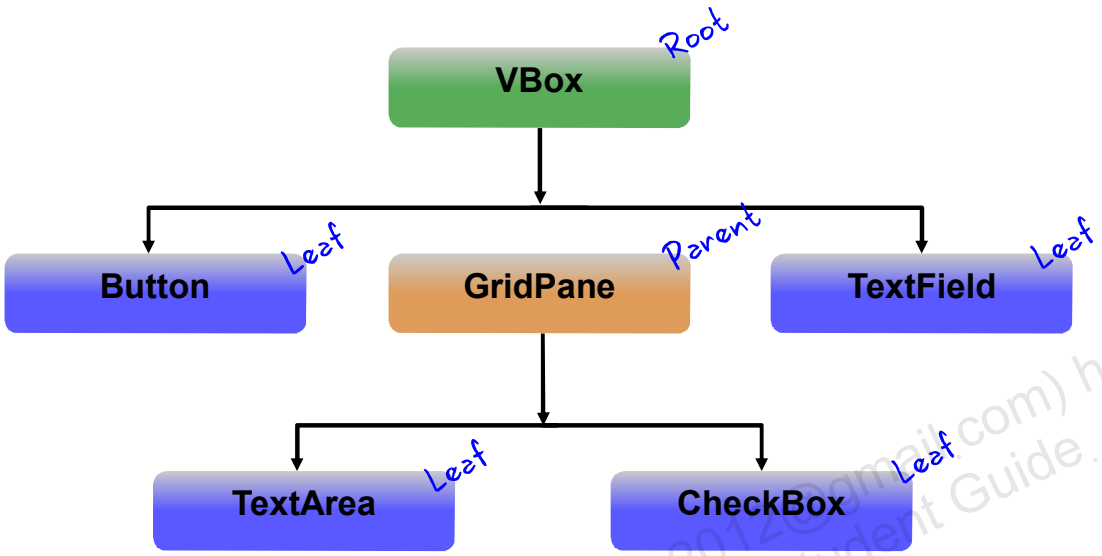
- Relate UI components to the scene graph
- Describe and implement JavaFX UI components such as controls, images, shapes, and layout containers
- Use CSS
- Add events to JavaFX controls
- Create charts

Topics

- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts



JavaFX Scene Graph and UI Elements



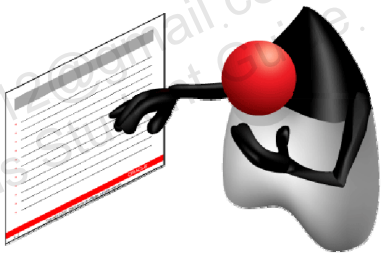
The JavaFX UI components that are available through the API are built by using nodes in the scene graph. Therefore, the components can use the visually rich features of the JavaFX platform. Because UI controls from the `javafx.scene.control` package are all extensions of the Node class, they can be integrated with the scene graph rendering, animation, transformations, and animated transitions. Scene graphs can become much larger than the example shown in the slide, but the basic organization (that is, the way in which parent nodes contain child nodes) is a pattern that repeats in all applications.

In the graphic in the slide, VBox is the root element and is from the `javafx.scene.layout` package. Button and TextField are leaf elements from the Control class, and GridPane is a parent node from the `javafx.scene.layout` package with the children TextArea and CheckBox, both from the `javafx.scene.control` package.

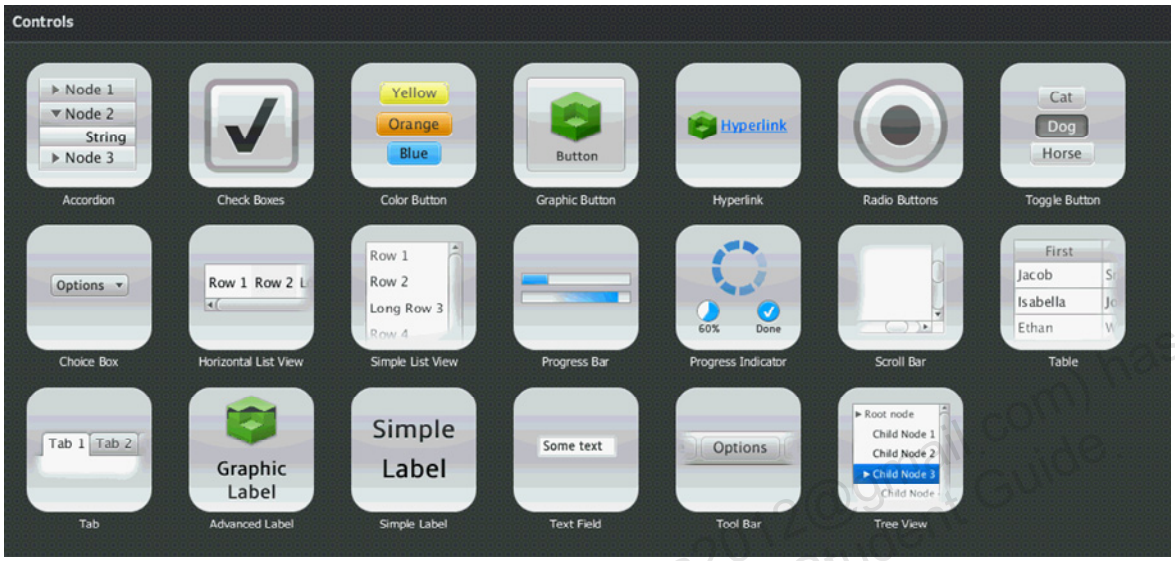
Throughout the rest of this lesson, you will see various applications of the scene graph. You will also learn how to manipulate nodes in the graph.

Topics

- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts



UI Controls

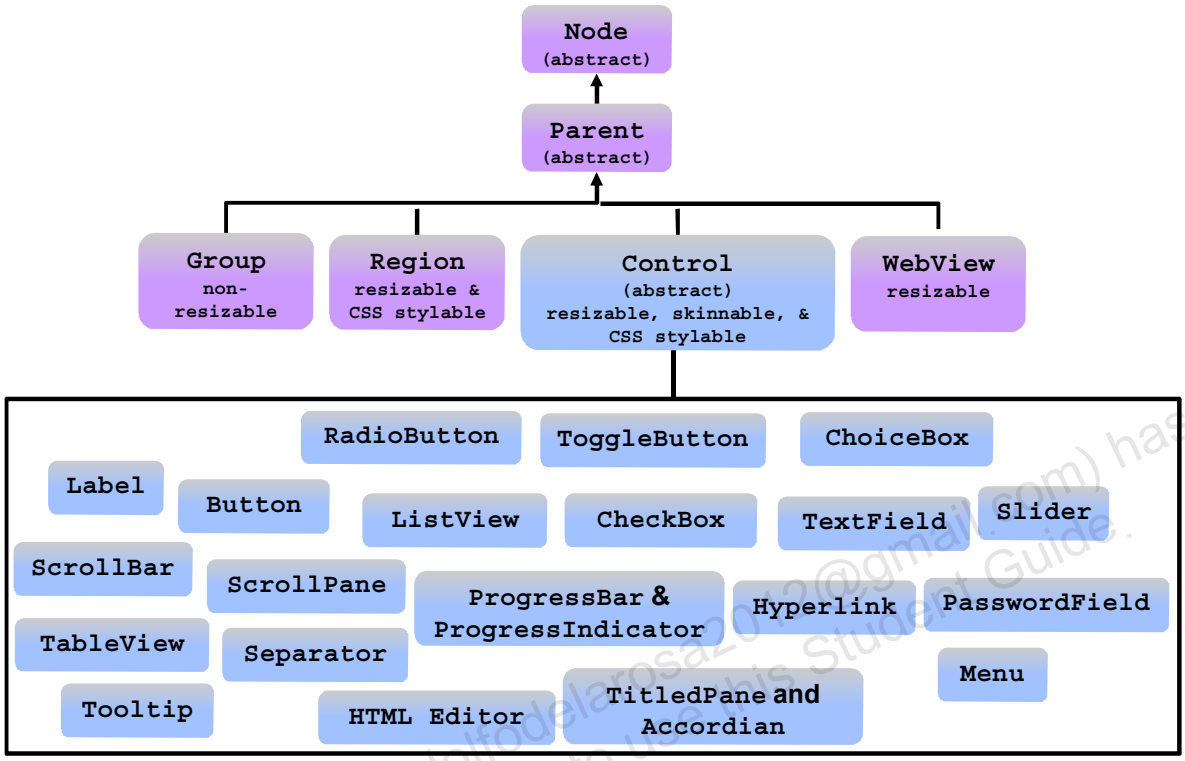


Some of the UI controls are:

- Label
- Button
- Radio Button
- Toggle Button
- Check Box
- Choice Box
- Text Field
- Password Field
- Scroll Bar
- Scroll Pane
- List View
- Table View
- Tree View
- Slider
- Progress Bar and Progress Indicator
- Hyperlink
- Tooltip
- HTML Editor
- Titled Pane and Accordion
- Menu
- Separator

See the Ensemble application for a complete list of UI controls.

Control Is a Node



The JavaFX user interface controls (*UI controls,* or just *controls*) are specialized nodes in the JavaFX scene graph especially suited for reuse in many different application contexts. They are designed to be highly customizable visually by designers and developers. They are designed to work well with layout mechanisms. Examples of prominent control nodes include Button, Label, ListView, and TextField. The graphic in the slide shows the available controls.

Because controls are nodes in the scene graph, they can be freely mixed with images, media, text, and basic geometric shapes, and they can be combined into groups.

The UI controls are resizable, skinnable, and CSS-stylable. Although writing new UI controls is not trivial, using and styling them are very easy—especially for web developers.

A control extends the Parent class and, as such, is not a leaf node. From the perspective of a developer or designer, the control can be thought of as if it were a leaf node in many cases. For example, the developer or designer can consider a button as if it were a rectangle or other simple leaf node.

Because a control is resizable, it is auto-sized to its preferred size on each scene graph pulse. Setting the width and height of the control does not affect its preferred size. When a control is used in a layout container, the layout constraints imposed on the control (or manually specified on the control) determine how it is positioned and sized.

The skin of a control can be changed at any time. Doing so marks the control as needing to be laid out because changing the skin is likely to change the preferred size of the control. If no skin is specified at the time that the control is created, a default CSS-based skin is provided for all of the built-in controls.

Each control may have an optional tooltip specified. The tooltip is a component that displays some (usually textual) information about a control when the cursor moves over the control for some period of time. As with other controls, a tooltip can be styled from CSS.

Henley Car Sales Button

UI Control: Example

The button is created in `HenleyClient.FXML`, and events are controlled in `HenleyClient.java`.

Button Created in `HenleyClient.fxml`

```
…
<children>
       <ImageView GridPane.columnIndex="0"
    GridPane.rowIndex="0"><image><Image
    url="@logo.png"/></image></ImageView>
       <Pane HBox.hgrow="ALWAYS"/>
       <Button fx:id="recordSalesButton" text="Record
    Sales" onAction="#recordSalesAction"/>
</children>
…
```

Button

In this example of an FXML application, the button is created in `HenleyClient.fxml`. This example represents the code for the button displayed in the slide titled “Henley Car Sales Button.”

Button Example in a Java Class

```
@Override
   public void start(Stage primaryStage) {
       primaryStage.setTitle("Hello World!");
       Button btn = new Button();
       btn.setText("Say 'Hello World'");
       btn.setOnAction(new EventHandler<ActionEvent>()
   {

           @Override
           public void handle(ActionEvent event) {
               System.out.println("Hello World!");
           }
       });
```


In this example of a JavaFX application, the button and event handles are created in the same class. Button is created using the `Button()` constructor. You can set the text for the button because the Button class inherits its properties and methods from the Labeled class.

This example represents the code for the button shown in the slide titled “Henley Car Sales Button.”

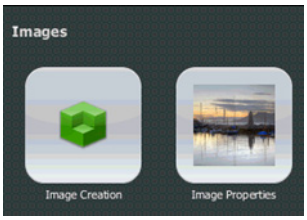
Tooltip

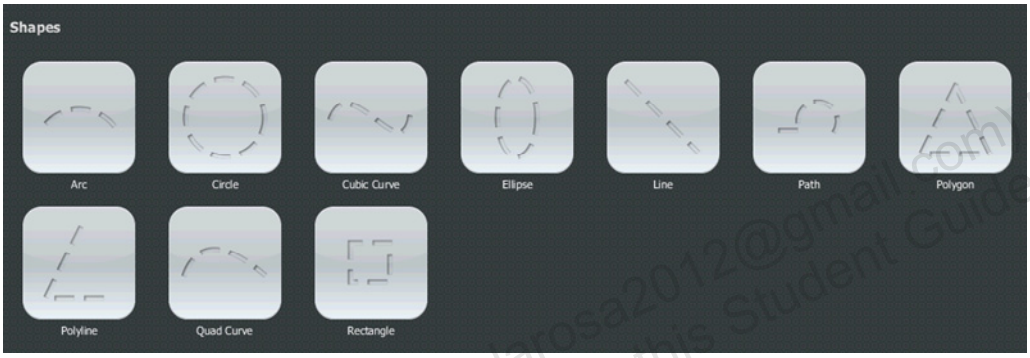
The Tooltip class represents a common UI component that is typically used to display additional information about the UI control. The tooltip is shown when the cursor is placed over the control.


```
final PasswordField pf = new
PasswordField(); final Tooltip tooltip
= new Tooltip(); tooltip.setText(
    "\nYour password must be\n" + "at
    least 8 characters in length\n" +
);
pf.setTooltip(tooltip);
```


The tooltip can be set on any control by calling the `setTooltip()` method.

Images and Shapes

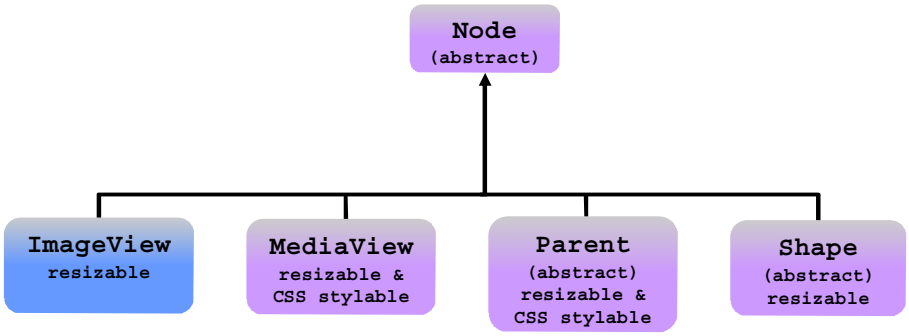




The `Image` class `javafx.scene.image` represents graphical images and is used for loading images from a specified URL. Images can be resized as they are loaded (for example, to reduce the amount of memory consumed by the image). The application can specify the quality of filtering used when scaling, and whether or not to preserve the original image's aspect ratio. Use `ImageView` to display images loaded with this class. The same `Image` instance can be displayed by multiple `ImageViews`.

The `Shape` class `javafx.scene.shape` is a set of 2D classes for defining and performing operations on objects related to two-dimensional geometry.

ImageView Is a Node

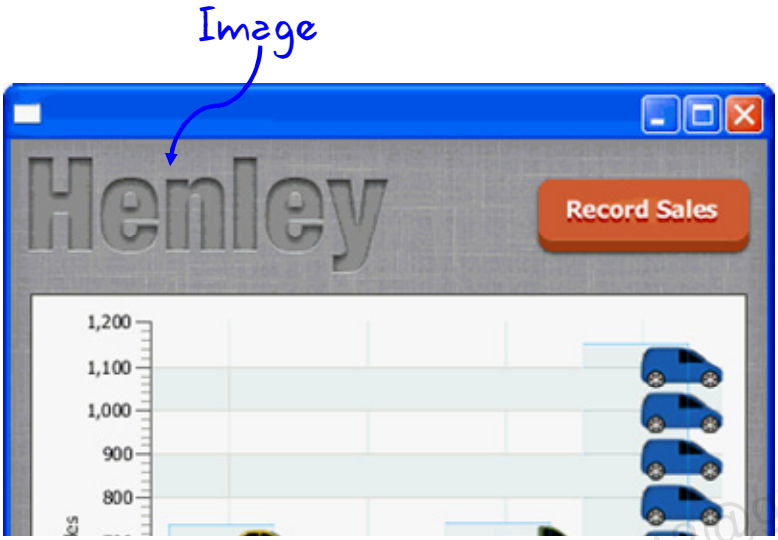



```
public class ImageView
extends Node
```

`ImageView` is a node used for painting images loaded with the `Image` class. This class allows resizing the displayed image (with or without preserving the original aspect ratio) and specifying a viewport into the source image for restricting the pixels displayed by this `ImageView`. Because the `ImageView` class is an extension of the `Node` class, you can apply visual effects, animations, and transformation to images in JavaFX.

The `Image` class is used to load images (synchronously or asynchronously). `Image` can be resized as it is loaded. The resizing can be performed with specified filtering quality and with the option of preserving the original aspect ratio of the image.

Henley Car Sales Image



The image is created in `HenleyClient.FXML`.

Image Created in `HenleyClient.fxml`

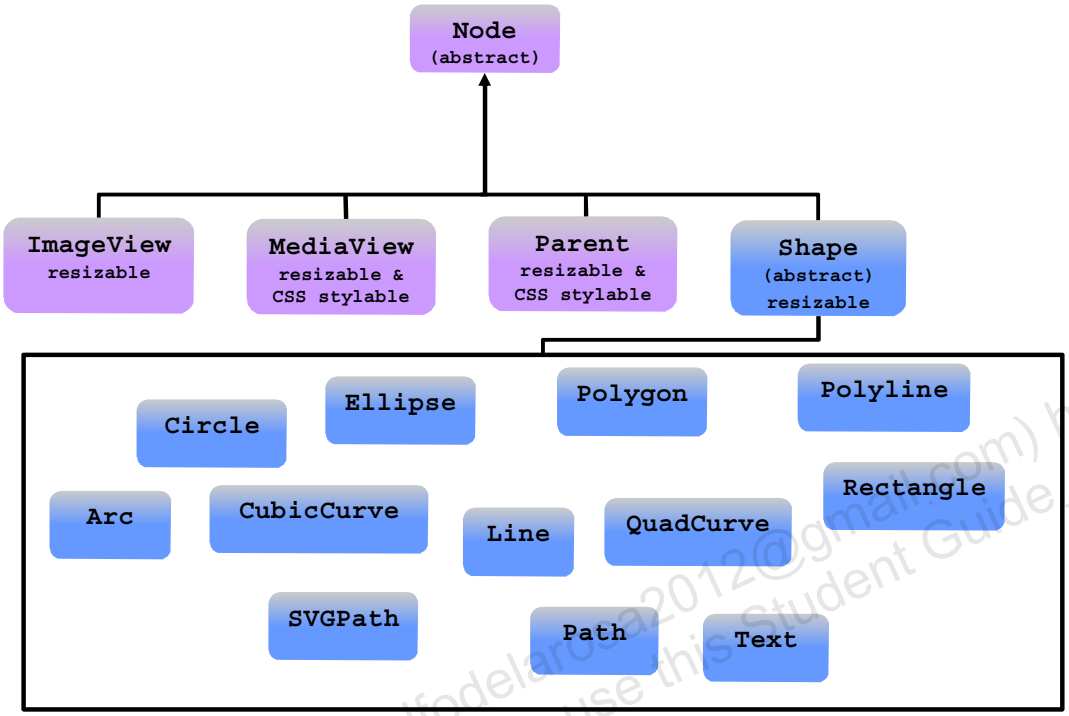
```
…
<children>
  <ImageView
    GridPane.columnIndex="0"GridPane.rowIndex="0"><image>
    <Image url="@logo.png"/></image></ImageView>
        <Pane HBox.hgrow="ALWAYS"/>
        <Button fx:id="recordSalesButton" text="Record
    Sales" onAction="#recordSalesAction"/>
</children>
…
```

Image

The code example `HenleyClient.fxml` shows the `ImageView` for the Henley logo, `logo.png`.

This example represents the code for the button shown in the slide titled "`ImageView` Is a `Node`."

Shape Is a Node




```
public abstract class Shape
extends Node
```

The `Shape` class provides definitions of common properties for objects that represent some form of geometric shape. These properties include:

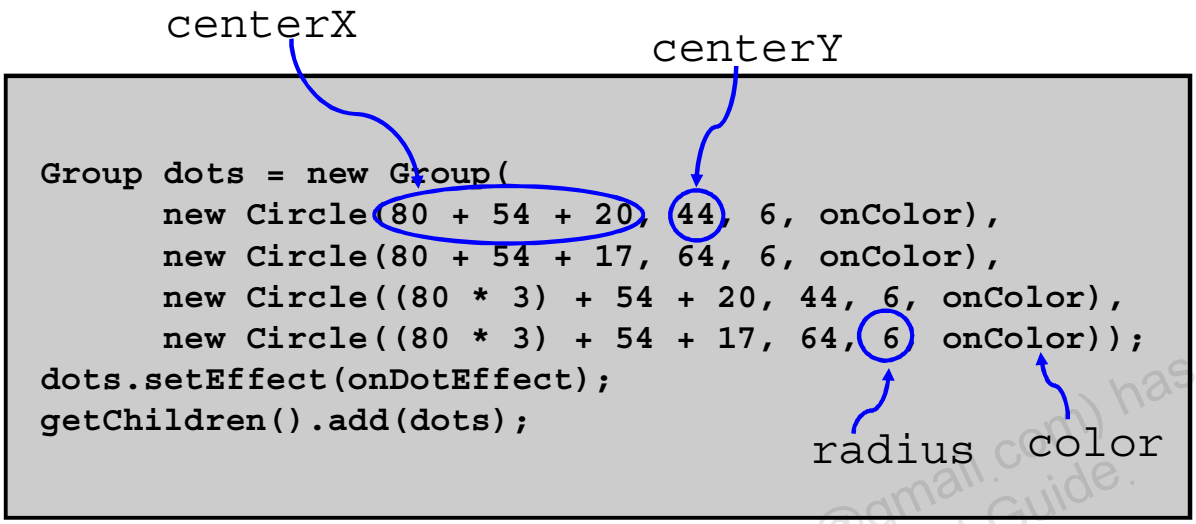
- The Paint to be applied to the interior of the shape (see `setFill`)
- The Paint to be applied to stroke the outline of the shape (see `setStroke`)
- The decorative properties of the stroke, including:
  - The width of the border stroke
  - Whether the border is drawn as an exterior padding to the edges of the shape (as an interior edging that follows the inside of the border) or as a wide path that follows along the border straddling it equally both inside and outside (see `StrokeType`)
  - Decoration styles for the joins between path segments and the unclosed ends of paths
  - Dashing attributes

Clock Example Uses Circles

In the DigitalClock sample application, the dots that separate the hours, minutes, and seconds are created using circles.

This Digital Clock example is in `D:\labs\06-CreateUI\examples\DigitalClock`.

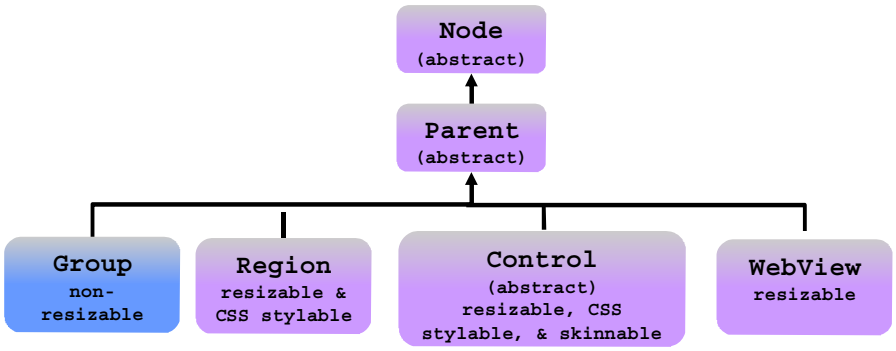
Clock Example: Circles



In the DigitalClock sample application shown in the previous slide, the dots that separate the hours, minutes, and seconds are created using circles. The four instances of Circle are created in a group. The size, position, and color of the circles are set.

- The first set of numbers is double `centerX`, which is the horizontal position of the center of the circle in pixels.
- The second set of numbers is double `centerY`, which is the vertical position of the center of the circle in pixels.
- The last number is the radius of the circle.
- The last entry sets the color.

Group Is a Node




```
public class Group
extends Parent
```

A Group node contains an `ObservableList` of children that are rendered in order whenever this node is rendered. A group will take on the collective bounds of its children and is not directly resizable.

Any transform, effect, or state applied to a group will be applied to all children of that group. Such transforms and effects will *not* be included in this group's layout bounds. However, if transforms and effects are set directly on children of this group, those will be included in this group's layout bounds.

By default, a group will "auto-size" its managed resizable children to their preferred sizes during the layout pass to ensure that regions and controls are sized properly as their state changes. If an application needs to disable this auto-sizing behavior, it should set `autoSizeChildren` to `false` and understand that if the preferred size of the children changes, they will not automatically resize.

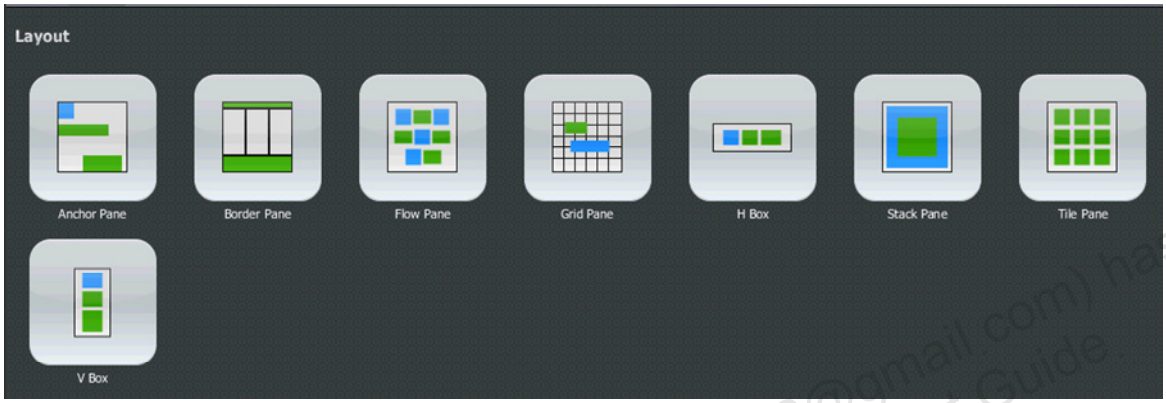
The previous slide of the DigitalClock code shows an example of objects grouped together.

Group of Circles

Group of four circles

```
Group dots = new Group(
      new Circle(80 + 54 + 20, 44, 6, onColor),
      new Circle(80 + 54 + 17, 64, 6, onColor),
      new Circle((80 * 3) + 54 + 20, 44, 6, onColor),
      new Circle((80 * 3) + 54 + 17, 64, 6, onColor));
dots.setEffect(onDotEffect);
getChildren().add(dots);
```

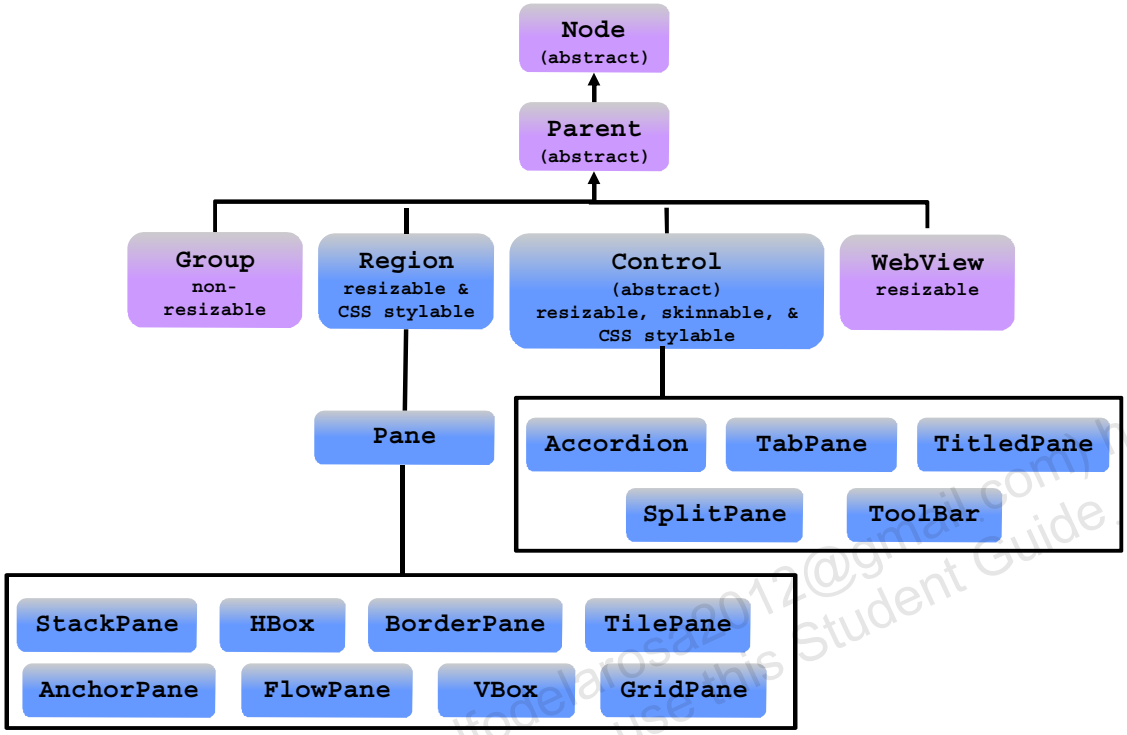
Layout Containers



The JavaFX SDK provides several layout container classes, called *panes*, for the easy setup and management of classic layouts such as rows, columns, stacks, tiles, and others. As a window is resized, the layout pane automatically repositions and resizes the nodes that it contains according to the properties for the nodes.

- BorderPane
- HBox
- VBox
- StackPane
- GridPane
- FlowPane
- TilePane
- AnchorPane

A Layout Container Is a `Node`



ORACLE

The packages `javafx.scene.layout` and `javafx.scene.control.Control` provide classes to support user interface layout. Each layout pane class supports a different layout strategy for its children, and applications may nest these layout panes to achieve the needed layout structure in the user interface. When a node is added to one of the layout panes, the pane will automatically manage the layout for the node, so the application should not position or resize the node directly.

The scene graph layout mechanism is driven automatically by the system after the application creates and displays a scene. The scene graph detects dynamic node changes that affect layout (such as a change in size or content), and it calls `requestLayout()`, which marks that branch as needing layout so that on the next pulse, a top-down layout pass is executed on that branch by invoking `layout()` on that branch's root. During that layout pass, the `layoutChildren()` callback method will be called on each parent to lay out its children. This mechanism is designed to maximize layout efficiency by ensuring that multiple layout requests are coalesced and processed in a single pass rather than executing re-layout on each minute change. Therefore, applications should not invoke layout directly on nodes.

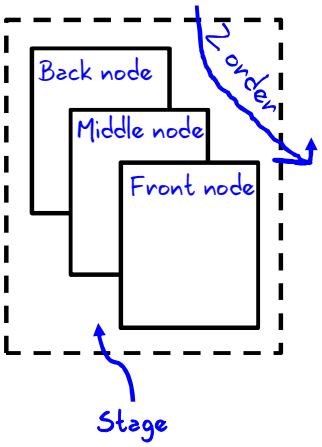
Types of Layout Containers

- StackPane
- BorderPane
- HBox
- VBox
- GridPane

Some of the layout containers used in the course application are listed in the slide.

StackPane

The StackPane stacks nodes in a Z order from back to front.



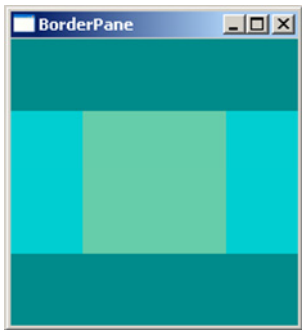

```
   StackPane stackPane = new StackPane();

        Rectangle rectangle = new Rectangle(280, 70,
Color.BISQUE);
        rectangle.setStroke(Color.BLACK);
        ImageView imageView = new ImageView(ICON_48);
        Label label = new Label("Your name could be
here.", imageView);
        label.setContentDisplay(ContentDisplay.RIGHT);

        stackPane.getChildren().addAll(rectangle,
label);

        root.getChildren().add(stackPane);
    }
```

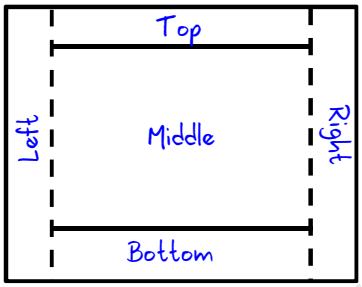

StackPane can be used to:

- Overlay text on an image or shape
- Overlap common shapes to make a complex shape

BorderPane

The BorderPane provides five regions in which to place nodes.





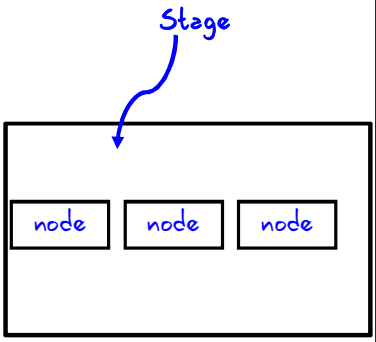

```
BorderPane root = new BorderPane();
root.setTop(new Rectangle(200, 50,
                          Color.DARKCYAN));
root.setBottom(new Rectangle(200, 50,
                             Color.DARKCYAN));
root.setCenter(new Rectangle(100, 100,
                       Color.MEDIUMAQUAMARINE));
root.setLeft(new Rectangle(50, 100,
                          Color.DARKTURQUOISE));
root.setRight(new Rectangle(50, 100,
                          Color.DARKTURQUOISE));

Scene scene = new Scene(root);
```


In the BorderPane, regions can be any size and do not have to be defined if they are not needed. A BorderPane can be used for the following:

- Menu across the top
- Status bar across the bottom
- Navigation bar along the left
- More information along the right
- Working area in the middle

HBox

The HBox arranges nodes in a horizontal row.




```
//Controls to be added to the HBox
    Label label = new Label("Test Label:");
    TextField tb = new TextField();
    Button button = new Button("Button...");

//HBox with spacing = 5
     HBox hbox = new HBox(5);
     hbox.getChildren().addAll(label, tb,button);
     hbox.setAlignment(Pos.CENTER);
     root.getChildren().add(hbox);
```

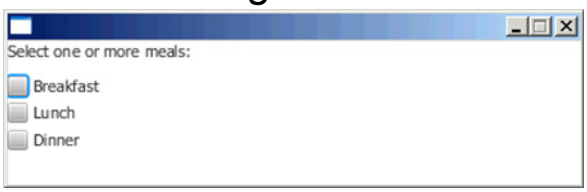
ORACLE

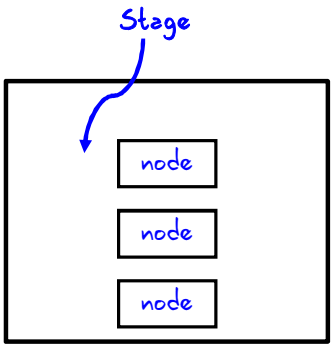
An HBox is easily modified:

- Padding attributes can be set to manage the distance between the nodes and the edges of the HBox pane.
- Spacing attributes can be set to manage the distance between the nodes.
- The style can be set to change the background color.

VBox

The VBox arranges nodes in a vertical column.






```
VBox vboxMeals = new VBox(5);
vboxMeals.getChildren().addAll(cb1, cb2, cb3);

Label label = new Label("Select one or more meals:");
VBox vboxOuter = new VBox(10);

vboxOuter.getChildren().addAll(label, vboxMeals);
vboxOuter.setAlignment(Pos.CENTER_LEFT);

root.getChildren().add(vboxOuter);
}
```

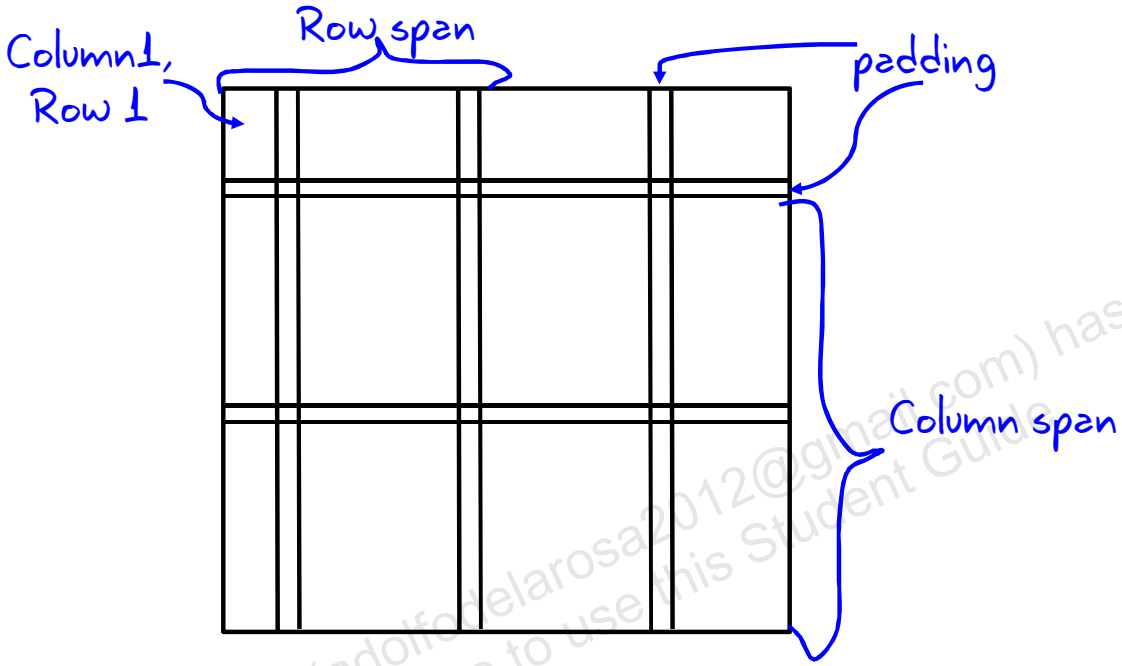
ORACLE

A VBox is easily modified:

- Padding attributes can be set to manage the distance between the nodes and the edges of the VBox pane.
- Spacing attributes can be set to manage the distance between the nodes.
- The style can be set to change the background color.

GridPane

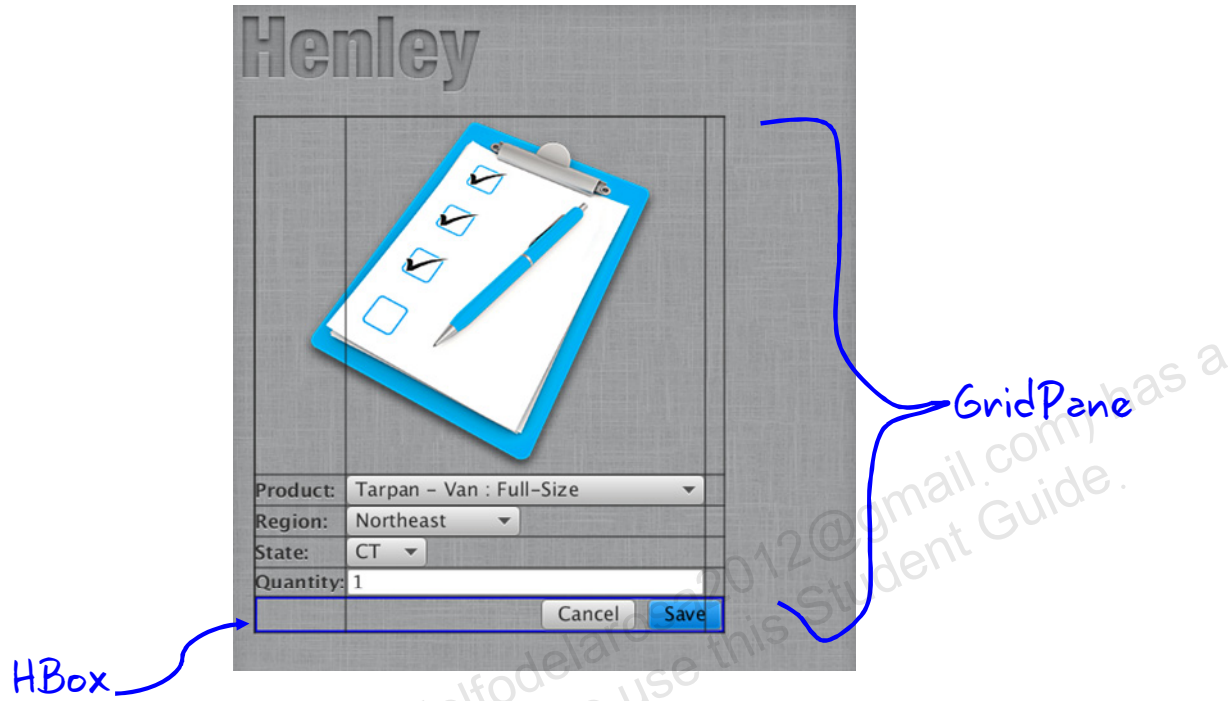
The GridPane is a flexible grid of rows and columns.



A GridPane enables you to create a flexible grid of rows and columns in which to lay out nodes. Nodes can be placed in any cell in the grid and can span cells as needed. A grid pane is useful for creating forms or any layout that is organized in rows and columns. Gap properties can be set to manage the spacing between the rows and columns. The padding property can be set to manage the distance between the nodes and the edges of the grid pane.

There is more than one approach to using a GridPane. First, the code can specify which rows and/or columns should contain the content. Second, the code can alter the constraints of the rows and/or columns themselves, either by specifying the preferred minimum or maximum heights or widths, or by specifying the percentage of the GridPane that belongs to certain rows or columns.

Henley Car Sales Form Layout



In the Henley Car Sales Form, a GridPane is the main layout class, and it contains a nested HBox layout container. All elements within the GridPane are child nodes of the GridPane. The image in the slide shows the GridPane with visible grid lines.

The GridPane has three columns and six rows, and the component position is specified by the row and column numbers. For example, the image is in Row 1, Column 1, and it spans two columns, as shown in the following code:

```
<ImageView GridPane.columnIndex="1" GridPane.rowIndex="1"
GridPane.columnSpan="2" GridPane.halignment="center"><image><Image
url="@clipboard.png"/></image></ImageView>
```

The `SalesForm.fxml` class is shown below:

```
<GridPane xmlns:fx="http://javafx.com/fxml"
fx:controller="henleyclient.SalesForm" fx:id="salesForm" hgap="15"
vgap="12">
    <padding><Insets top="10" right="10" bottom="10"
left="10"/></padding>
    <children>
        <ImageView GridPane.columnIndex="1" GridPane.rowIndex="1"
GridPane.columnSpan="2" GridPane.halignment="center"><image><Image
url="@clipboard.png"/></image></ImageView>
        <Label text="Product:" GridPane.columnIndex="1"
GridPane.rowIndex="2"/>
        <ChoiceBox fx:id="productsChoiceBox"
GridPane.columnIndex="2" GridPane.rowIndex="2" prefWidth="300"/>
        <ProgressIndicator fx:id="productsProgress" prefWidth="16"
prefHeight="16" GridPane.columnIndex="3" GridPane.rowIndex="2"/>
        <Label text="Region:" GridPane.columnIndex="1"
GridPane.rowIndex="3"/>
        <ChoiceBox fx:id="regionsChoiceBox" GridPane.columnIndex="2"
GridPane.rowIndex="3"/>
        <ProgressIndicator fx:id="regionsProgress" prefWidth="16"
prefHeight="16" GridPane.columnIndex="3" GridPane.rowIndex="3"/>
        <Label text="State:" GridPane.columnIndex="1"
GridPane.rowIndex="4"/>
        <ChoiceBox fx:id="statesChoiceBox" GridPane.columnIndex="2"
GridPane.rowIndex="4" GridPane.columnSpan="2"/>
        <Label text="Quantity:" GridPane.columnIndex="1"
GridPane.rowIndex="5" minWidth="-Infinity"/>
        <TextField fx:id="quantityTextField" text="1"
GridPane.columnIndex="2" GridPane.rowIndex="5"/>
        <HBox spacing="10" alignment="center_right"
GridPane.columnIndex="1" GridPane.rowIndex="6"
GridPane.columnSpan="3">
            <children>
                <Button text="Cancel" onAction="#cancelAction"
cancelButton="true"/>
                <Button text="Save" onAction="#saveAction"
defaultButton = "true"/>
            </children>
        </HBox>
    </children>
</GridPane>
```

Resizability

JavaFX has two mechanisms for achieving resizing:

- Layout container classes
- Resizing and layout at the node level

ORACLE

Nodes in the scene graph have the potential to be *resizable*. This means that a node expresses its minimum, preferred, and maximum size constraints and allows its parent to resize it during layout (hopefully within that range, if the parent is honorable).

Each layout pane implements its own policy for resizing a resizable child, given the child's size range. For example, a `StackPane` will attempt to resize all children to fill its width/height (honoring their max limits) while a `FlowPane` always resizes children to their preferred sizes, which is what we mean by the term *auto-size* ("resize-to-preferred").

Not all scene-graph node classes are resizable:

- **Resizable:** Region, Control, WebView, MediaView, ImageView
- ***Not* resizable:** Group, Text, Shape

If a Node class is not resizable, note the following results:

`isResizable()` returns `false`.

`minWidth()`, `minHeight()`, `prefWidth()`, `prefHeight()`, `maxWidth()`, and `maxHeight()` all return their current sizes.

`resize()` is a no-op.

See Amy Fowler's blog at http://amyfowlersblog.wordpress.com/2011/06/02/javafx2-0-layout-a-class-tour/ for information about resizability, including the following discussion:

- The size of a non-resizable node is affected by its various properties (width/height for Rectangle, radius for Circle, and so on) and it is the application's responsibility to set these properties (and remember that scaling is *not* the same as resizing, because a scale also transforms the stroke, which is not usually what you want for layout). When non-resizable nodes are placed into layout panes, they will be positioned but not resized.
- Why not make shapes resizable? Although it is fairly easy to do the math for basic shapes (Rectangle, Circle, and so on), it becomes a bit more difficult with Paths, Polygons, and 3D. Also, keeping the core graphics shapes non-resizable creates certain efficiencies, such as ensuring that a Group with non-resizable descendents pays no penalty for layout.

Note: For additional information about layout containers and resizing nodes, see the tutorial "Working with Layouts" at http://docs.oracle.com/javafx/2.0/layout/jfxpub-layout.htm.

Quiz

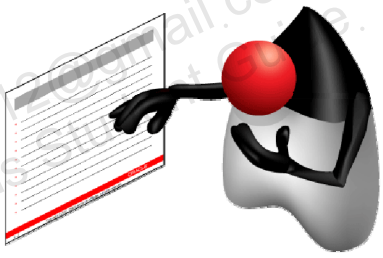
When a layout container is resized, its nodes are resized according to their properties.

a. True
b. False

Answer: a

Topics

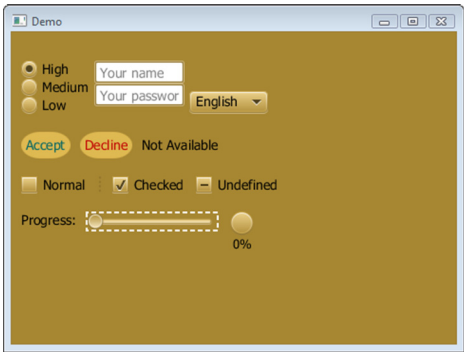
- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts

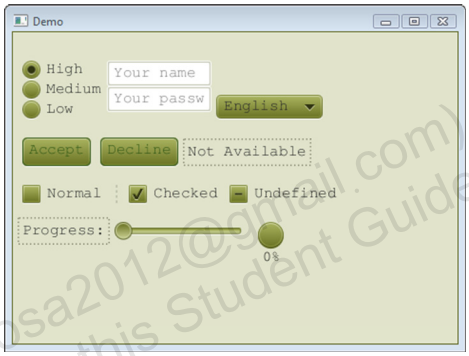


Skinning UI Controls

UI controls can be skinned using two methods:

- Applying CSS
- Overriding default styles on individual components





You can create a custom appearance (also called a *skin*) for your JavaFX application with cascading style sheets (CSS). CSS contain style definitions that control the appearance of user interface elements. Using CSS in JavaFX applications is similar to using CSS in HTML. JavaFX CSS are based on the W3C CSS version 2.1 specification (available at http://www.w3.org/TR/CSS21/) with some additions from current work on version 3 of the specification and some extensions that support specific JavaFX features.

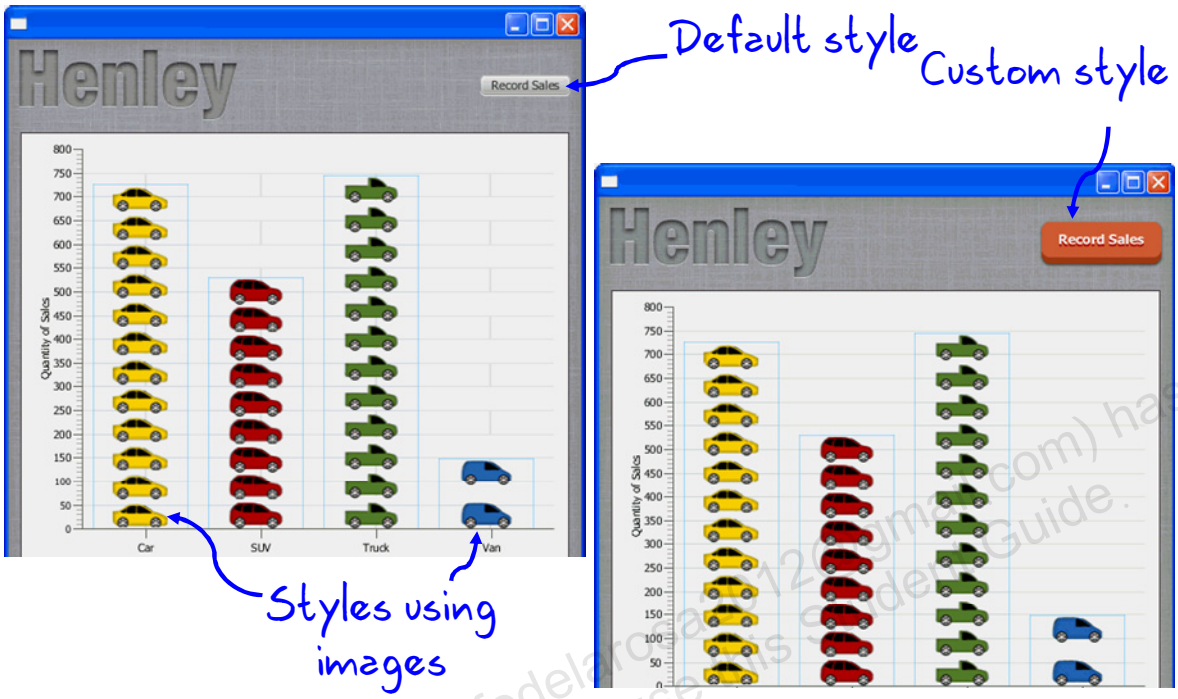
You can further customize your UI by defining styles for the different controls that you are using. You can override the definitions in the default style sheet or create new class or ID styles. You can also define the style for a node within your code.

The images in the slide show an application that uses two different style sheets. The default style sheet for JavaFX applications is `caspian.css`, which is found in the JavaFX Runtime JAR file, `jfxrt.jar`. This style sheet defines styles for the root node and the UI controls. To view this file, go to the `\rt\lib` directory under the directory in which the JavaFX SDK is installed. Use the following command to extract the style sheet from the JAR file:

```
jar -xf jfxrt.jar
com/sun/javafx/scene/control/skin/caspian/caspian.css
```

The JavaFX scene graph provides the capability of styling nodes by using CSS. The `Node` class contains the `styleClass` `id`, and style variables are used by CSS selectors to find nodes to which styles should be applied. The `Scene` class contains the `stylesheets` variable, which is a sequence of URLs that reference CSS style sheets that are to be applied to the nodes within that scene.

CSS in Henley Sales Application



The screenshots in the slide show two versions of the Record Sales button in the Henley Sales application. The first image shows the button using the default style. The second image shows the button with custom styles applied.

Note: For additional information about CSS, applying CSS styles to nodes, and the properties that are available for styling, see the "CSS Reference Guide" at `D:\labs\resources\JavaFX-CSS-Reference-Guide.pdf`.

CSS Syntax

The following example of JavaFX CSS syntax changes the color of the button text from the default black to white. It also includes a drop shadow.

```
#recordSalesButton .text {
    -fx-fill: white;
    -fx-effect: dropshadow( gaussian , #a30000 , 0,0,0,2
   );
}
```

An `id` for the element must be defined and then referenced in the associated Java file so that the style can be applied to the element.

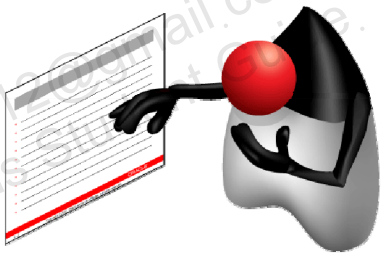
```
<ChoiceBox fx:id="customerChoiceBox"/>
```


Each node in the scene graph has an `id` variable (a string). This is analogous to the `id="..."` attribute that can appear in HTML elements. Supplying a string for a node's `id` variable causes style properties for this node to be looked up using that `id`. Styles for specific `id`s can be specified using the `#nodeid` selector syntax in a style sheet.

Note: For more information, see the "CSS Reference Guide" at `D:\labs\resources\JavaFX-CSS-Reference-Guide.pdf`.

Topics

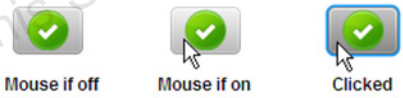
- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts



ORACLE

Events in JavaFX

- JavaFX uses event handlers and event filters for events such as:
  - Mouse events
  - Keyboard events
  - Drag-and-drop events
  - Window events
  - Action events
- Every event includes
  - Event type
  - Event source
  - Event target

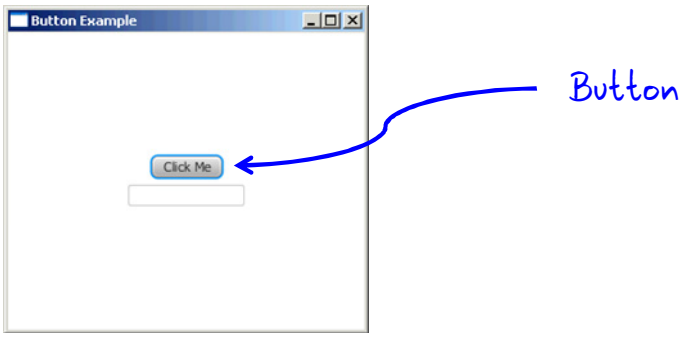


An event represents an occurrence of something of interest to the application, such as moving a cursor or pressing a key. In JavaFX, an event is an instance of the `javafx.event.Event` class or any subclass of Event. JavaFX provides several events, including `DragEvent`, `KeyEvent`, `MouseEvent`, `ScrollEvent`, and others. You can define your own event by extending the Event class.

The event filters and event handlers are implementations of the `EventHandler` interface. If an application wants to be notified when an event occurs, a filter or handler is registered for the event. The primary difference between a filter and a handler is when each one is executed. Event handlers are executed as the event traverses the event dispatch chain from the event target. Event filters are executed during the event capture phase as the event traverses the event dispatch chain from the stage to the event target.

- **Event type:** Type of event that occurred. An event type is an instance of the `EventType` class.
- **Source:** Origin of the event, with respect to the location of the event in the event dispatch chain. The source changes as the event is passed along the chain.
- **Target:** Node on which the action occurred and the end node in the event dispatch chain. The target does not change. The target of an event can be an instance of any class that implements the `EventTarget` interface.

Button Event

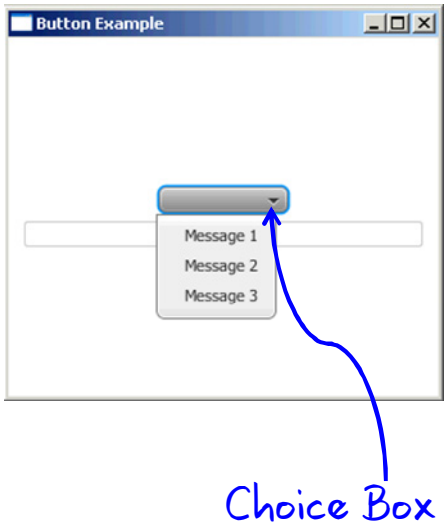



```
// Click event handler
   btn.setOnAction(new EventHandler<ActionEvent>() {
     @Override
       public void handle(ActionEvent event) {
       System.out.println("Button Clicked");
                clickCtr++;

       clickFld.setText(Integer.toString(clickCtr));
     }
   });
```


In this example, the button event prints `Button Clicked` each time the user clicks the button. The example also uses a click counter to count each button click.

Choice Box

A Choice Box enables users to choose one of several options.

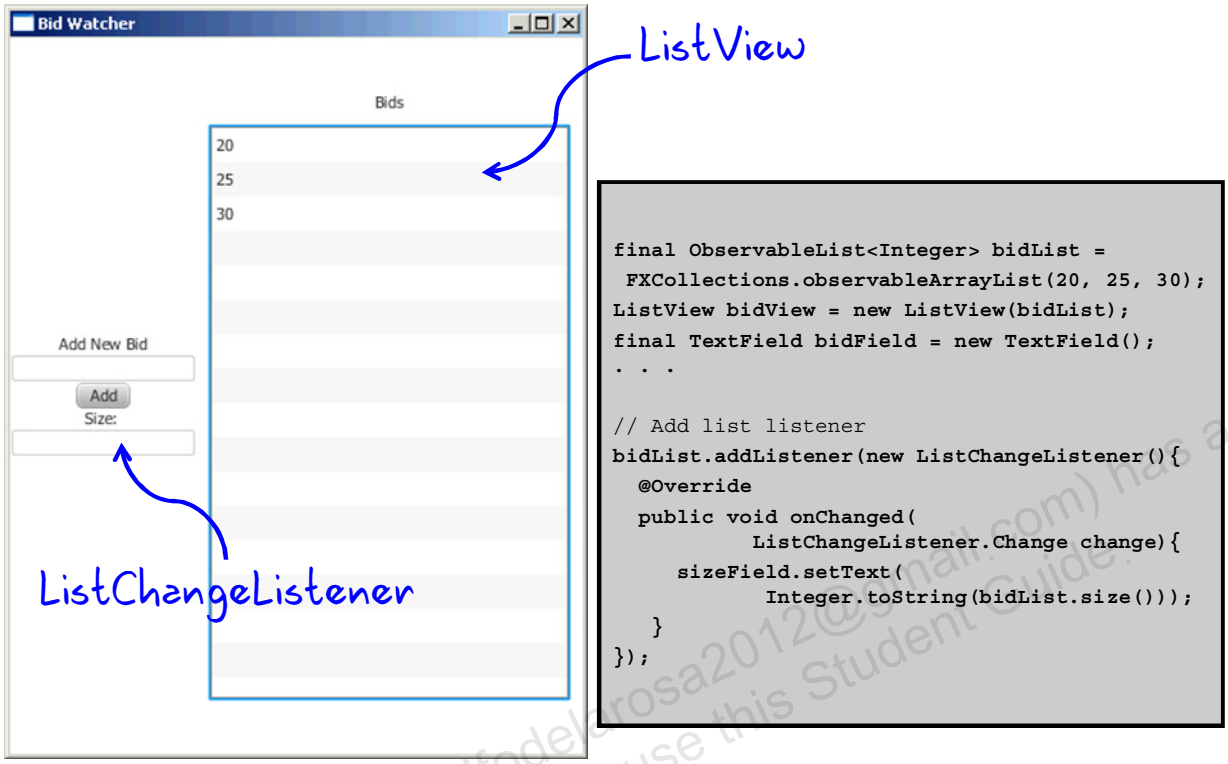



```
// ChoiceBox event handler
   choiceBox.getSelectionModel().selectedIndexProperty().
                                        addListener(
      new ChangeListener<Number>() {

        @Override
         public void changed(ObservableValue ov, Number
                                        value, Number newValue) {
            int index = newValue.intValue();
            messageFld.setText(messages.toArray()[
                                           index].toString());
            System.out.println("Message set to: " +
                     messages.toArray()[index].toString());
        }
    });
```


List items are created and populated within a constructor of the `ChoiceBox` class. The list items are specified by using an observable array. Alternatively, you can use an empty constructor of the class and set the list items by using the `setItems` method.

A Choice Box can contain not only text elements but also other objects.

Listener



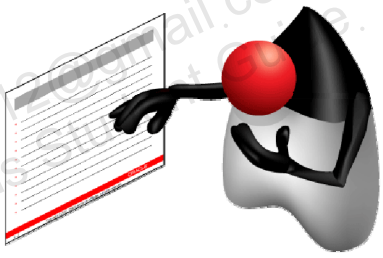

```
final ObservableList<Integer> bidList =
 FXCollections.observableArrayList(20, 25, 30);
ListView bidView = new ListView(bidList);
final TextField bidField = new TextField();
. . .

// Add list listener
bidList.addListener(new ListChangeListener(){
  @Override
  public void onChanged(
           ListChangeListener.Change change){
     sizeField.setText(
            Integer.toString(bidList.size()));
   }
});
```


The `ListView` is set up as one listener and automatically updates each time the list is updated.

The `ListChangeListener` event handler is a second listener. It displays the size of the `bidList` every time you add a new bid.

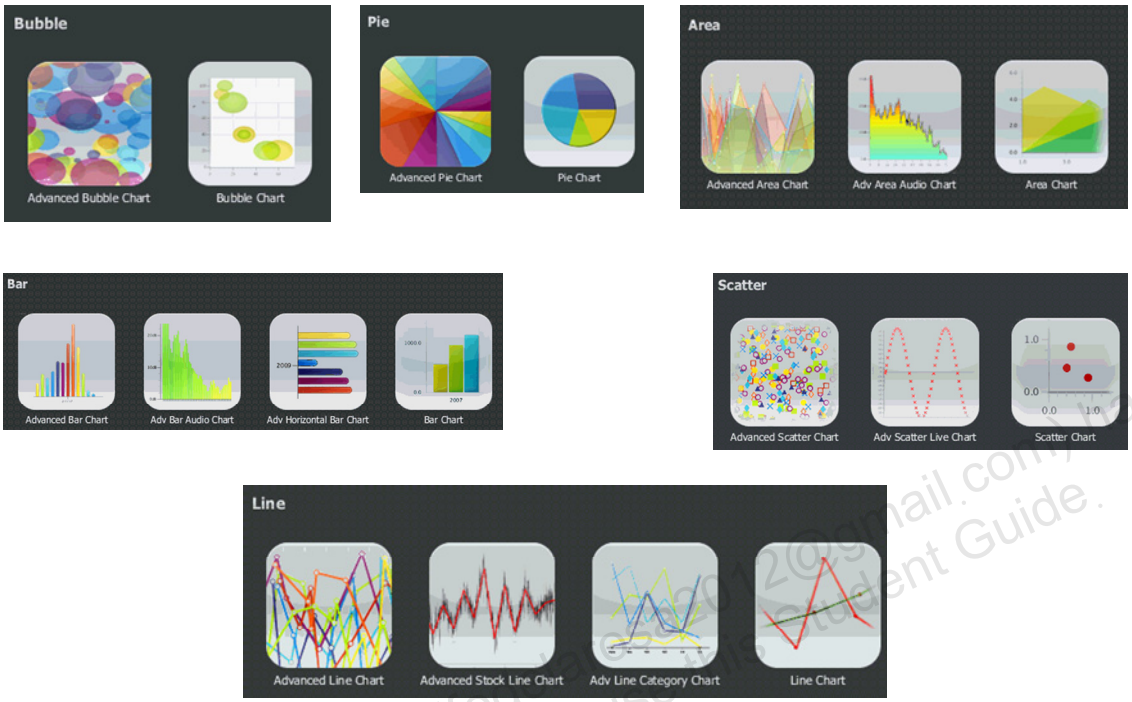
Topics

- Relating UI components to the scene graph
- Describing and implementing JavaFX UI components such as controls, images, shapes, and layout containers
- Using CSS
- Adding events to JavaFX components
- Creating charts



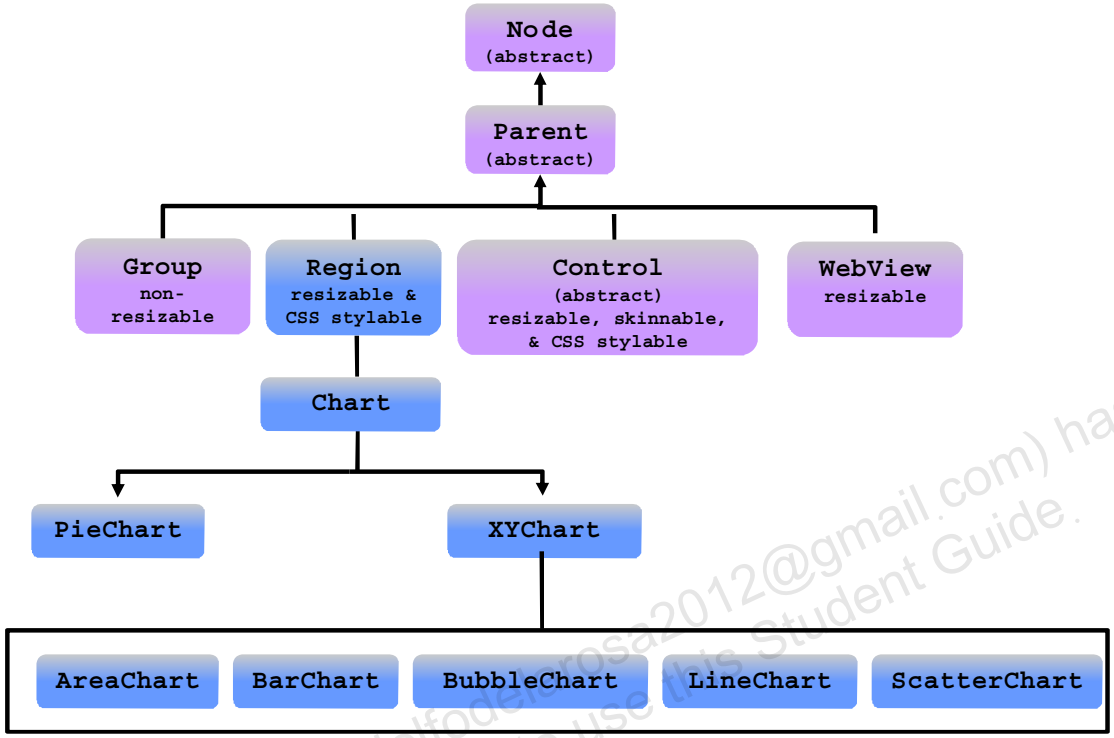
ORACLE

Charts



In addition to the typical elements of a user interface, the JavaFX SDK provides charts in the `javafx.scene.chart` package. The types of charts currently supported are bar, area, line, bubble, scatter, and pie.

Chart Is a Node



The JavaFX User Interface provides a set of chart components that are very convenient for displaying data. Application developers can make use of the graphical charts provided by the SDK to illustrate a wide variety of data.

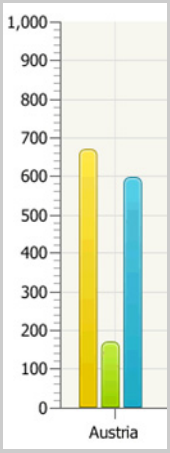
Common types of charts such as Bar, Line, Area, Pie, Scatter, and Bubble are provided. These charts are easy to create and are customizable. The JavaFX Charts API is a visual-centric (rather than model-centric) API.

JavaFX charts support animation of chart components as well as auto-ranging of the chart axes:

```
javafx.scene.chart.Chart

public abstract class Chart
extends Region
```

To create a chart, you need to instantiate the specific chart class, define the data, assign the data items to the specific chart class object, and add the chart to the application.

Chart Example in a Java Class

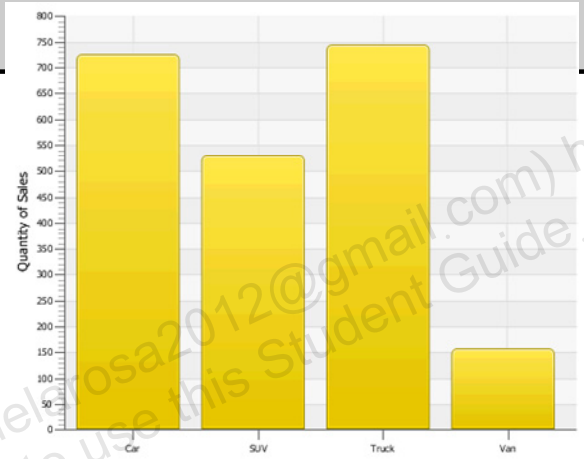



```
@Override public void start(Stage stage) {
        stage.setTitle("Bar Chart Sample");
final CategoryAxis xAxis = new CategoryAxis();
final NumberAxis yAxis = new NumberAxis();
final BarChart<String,Number> bc =
    new BarChart<String,Number>(xAxis,yAxis);
bc.setTitle("Country Summary");
xAxis.setLabel("Country");
yAxis.setLabel("Value");

XYChart.Series series1 = new XYChart.Series();
series1.setName("2003");
series1.getData().add(new XYChart.Data(austria,
   25601.34));
…
```


The code example shows a BarChart created in a Java class. To build a BarChart in your JavaFX application, you create two axes, instantiate the BarChart class, define the series of data, and assign the data to the chart.

Chart Example in FXML

```
<?import javafx.scene.chart.*?>
…
 <BarChart fx:id="salesChart" legendVisible="false">
  <xAxis><CategoryAxis/></xAxis>
  <yAxis><NumberAxis label="Quantity of Sales"/></yAxis>
 </BarChart>
…
```

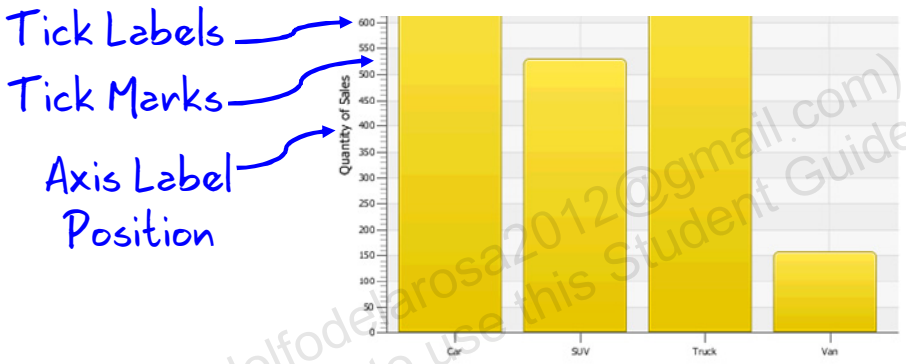


In the FXML example code, the BarChart is named `salesChart`, and the legend is hidden. The X axis is defined as `CategoryAxis` and the Y axis is "Quantity of Sales." The X axis labels will be defined using a style sheet.

Defining the X Axis and Y Axis

Change the default appearance of each chart by defining:

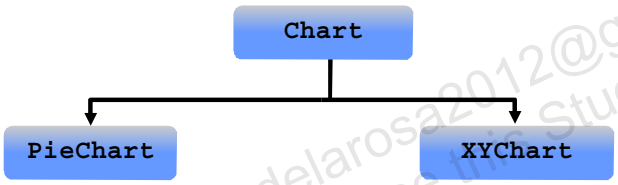
- The axis label
- The axis position relative to the chart plot
- The upper and lower boundaries of the axis range
- The minimum and maximum tick marks, tick units, the gap between two tick marks, and tick labels



You can also specify animation for any changes to an axis and its range, or you can enable the axis to automatically determine its range from the data.

Chart Data

- The `XYChart.Data` class specifies the data model:
  - `xValue` defines the value of the chart element plotted on the X axis.
  - `yValue` defines the value of the chart element plotted on the Y axis.
- The `PieChart.Data` class specifies values for each slice of the pie.
- The `XYChart.Series` class defines several series of data.



The `XYChart` class, a super class for all two-axis charts, provides basic capabilities for building area, line, bar, scatter, and bubble charts. Use the `XYChart.Data` class to specify the data model for these types of charts.

The `xValue` property defines the value of a chart element to be plotted on the X axis, and the `yValue` property defines the value for the Y axis.

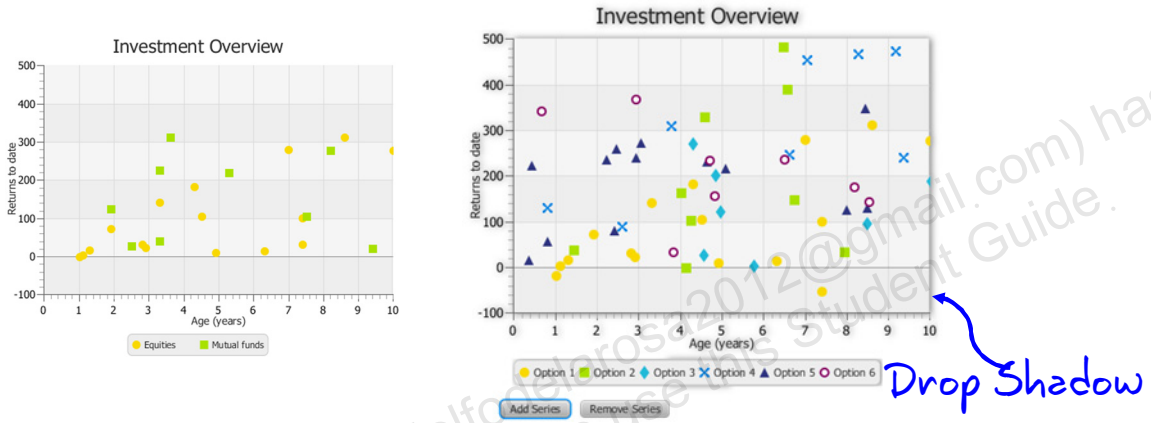
You can also set the extra value for each chart element. This value can be plotted in any way the chart needs, or it can be used to store additional information about the chart element. For example, it can be used to define a radius for bubble charts.

Use the `XYChart.Series` class to define as many sets of data as you need to represent on your graph. You can also assign a particular name to each series to display in the chart legend.

Unlike a two-axis chart, the pie chart does not require defining values for X and Y axes. You use the `PieChart.Data` class to specify values for each slice in the pie.

Apply Effects

```
final DropShadow shadow = new DropShadow();
shadow.setOffsetX(2);
shadow.setColor(Color.GREY);
sc.setEffect(shadow);
```



All the chart classes available in the `javafx.scene.chart` are extensions of the `Node` class. Therefore, you can apply visual effects or transformation to every type of chart.

The ScatterChart example shown in this slide is in `D:\labs\05-UIControls\examples\ScatterChartSample`.

Style Charts

```
.default-color0.chart-area-symbol { -fx-background-
   color: #e9967a, #ffa07a; }
.default-color1.chart-area-symbol { -fx-background-
   color: #f0e68c, #fffacd; }
.default-color2.chart-area-symbol { -fx-background-
   color: #dda0dd, #d8bfd855; }

.default-color0.chart-series-area-line { -fx-stroke:
   #e9967a; }
.default-color1.chart-series-area-line { -fx-stroke:
   #f0e68c; }
.default-color2.chart-series-area-line { -fx-stroke:
   #dda0dd; }
```


The colors of the chart elements are defined by the implementation of each particular chart class. However, you can alter these colors as well as chart symbols by applying CSS styles.

Animate Charts

The `Chart` class has the following:

- `animated` property
- `setAnimated` method

With the JavaFX SDK, you can make your chart change dynamically as the data changes. Use the `animated` property and the `setAnimated` method of the `Chart` class to toggle this functionality for the chart. You can use the `animated` property and the `setAnimated` method of the `Axis` class to animate changes to either axis and its range.

Quiz

Which class do you use to specify a series of data in a chart?

a. `XYChart.Data`
b. `PieChart.Data`
c. `PieChart.Series`
d. `XYChart.Series`

Answer: d

Summary

In this lesson, you should have learned how to:

- Relate UI components to the scene graph
- Describe and implement JavaFX UI components such as controls, images, shapes, and layout containers
- Use CSS
- Add events to JavaFX controls
- Create charts

Practice 5: Overview

- 5-1: Creating an Order Form by Using FXML
- 5-2: Adding Events to the Order Form
- 5-3: Creating an Interactive Pie Chart

6

Visual Effects, Animation, WebView, and Media

ORACLE

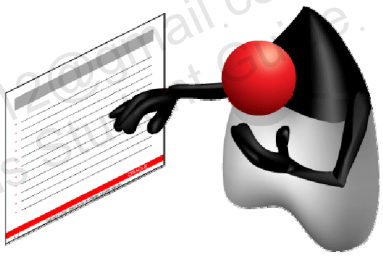
Objectives

After completing this lesson, you should be able to:

- Use animation and effects in an application
- Describe how to implement media in an application
- Describe the benefits of using WebView

Topics

- Using animation and effects in an application
- Describing how to implement media in an application
- Describing the benefits of using WebView



Animation: Timelines and Transitions

Animation in JavaFX can be divided into transitions and timelines.

`Transition` and `Timeline` are subclasses of the `javafx.animation.Animation` class.

Transitions: Transitions in JavaFX provide a way to incorporate animation in an internal timeline. Transitions can be composed to create multiple animations that are executed in parallel or sequentially.

Timelines: An animation is driven by its associated properties, such as size, location, and color. Timelines provide the capability to update the property values along the progression of time. JavaFX supports key frame animation. In key frame animation, the animated state transitions of the graphical scene are declared by start and end snapshots (*key frames*) of the scene's state at certain times. The system can automatically perform the animation. It can stop, pause, resume, reverse, or repeat movement when requested.

Using Animation in JavaFX: Introduction to Computer Animation

- In traditional hand-drawn animation, a lead animator draws key drawings in a scene.
- These key drawings are passed to assistant animators, who draw the in-between frames to achieve smooth movement.
- The action duration dictates how many in-between frames are needed. This process is called *tweening*.
- Borrowing from the traditional animation process, computer animation is a time period sliced with key frames.
- The computer does the *tweening* process by applying mathematical formulas to adjust the position, opacity, color, and other aspects required for the action. This is based on the timing constraints placed between two key frames.

ORACLE

The key drawings in a scene are the most important actions and represent the extremes of the action. The idea is to provide enough detail to present the main elements of movement. The lead animator decides how long each action should last on the screen.

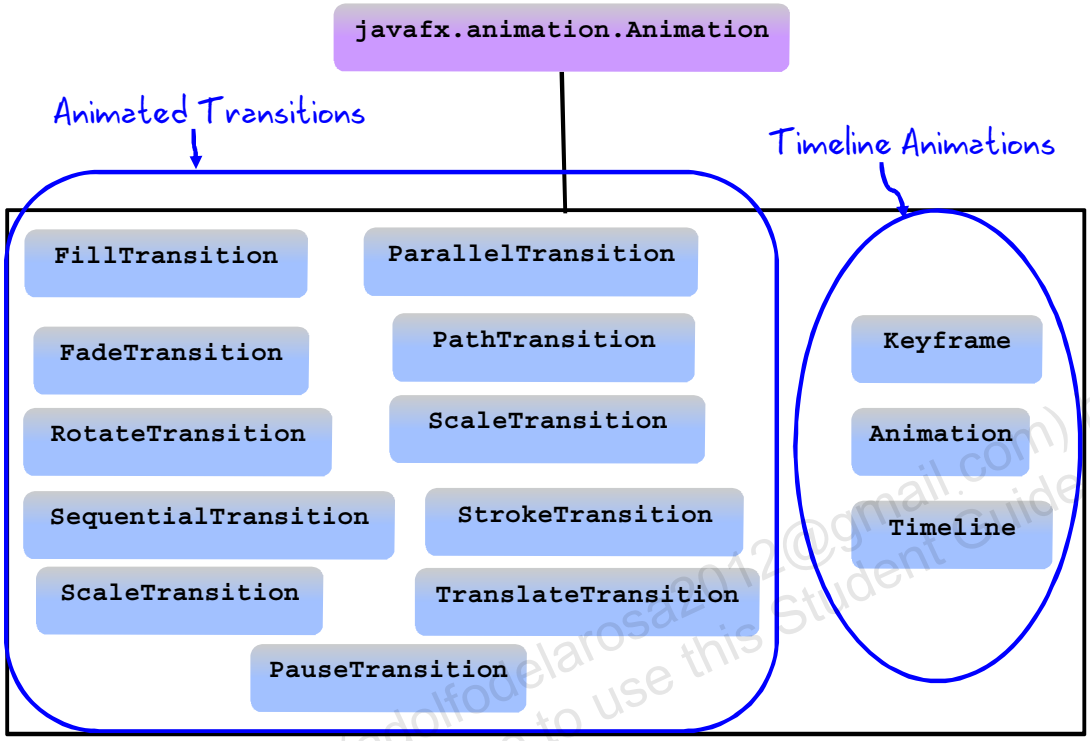
The word *tweening* is short for “in-betweening.”

In computer animation, the computer takes the role of the assistant animators. The computer decides how many in-between frames are required based on the timing constraints placed between two key frames.

The primary property of animation is time, so JavaFX supports the time period of an animation sequence with a timeline. The JavaFX class that represents animation actions spread across a time duration is `javafx.animation.Timeline`. Key frames, derived from `javafx.animation.KeyFrame`, are interspersed across the timeline at specified intervals, represented by time durations (`javafx.lang.Duration`).

These key frames might contain key values (`javafx.animation.KeyValue`) that represent the end state of the specified application values such as `position`, `opacity`, and `color`. These key values might also include actions that execute when the key time occurs. Key values containing a declaration of the mathematical formula, or interpolator (`javafx.animation.Interpolator`), should be used in the tweening process.

`Animation` Is Applied to a Node




```
public abstract class Animation
extends java.lang.Object
```

The Animation class provides the core functionality of all animations used in the JavaFX Runtime. An Animation can run in a loop by setting `cycleCount`. To make an animation run back and forth while looping, set the `autoReverse` flag.

Call `play()` or `playFromStart()` to play an animation. The animation progresses in the direction and speed specified by `rate`, and stops when its duration has elapsed. An animation with indefinite duration (a `cycleCount` of `INDEFINITE`) runs repeatedly until the `stop()` method is explicitly called, which will stop the running animation and reset its play head to the initial position.

An animation can be paused by calling `pause()`, and the next `play()` call will resume the animation from where it was paused.

An animation's play head can be randomly positioned whether it is running or not. If the Animation is running, the play head jumps to the specified position immediately and continues playing from the new position. If the animation is not running, the next `play()` will start the animation from the specified position.

Inverting the value of `rate` toggles the play direction.

Timeline Animation

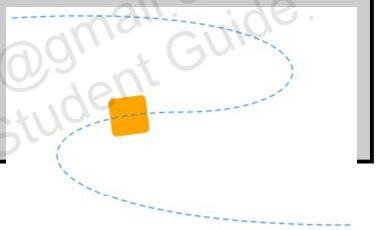
```
final Rectangle rectBasicTimeline = new Rectangle(100,
   50, 100, 50);
rectBasicTimeline.setFill(Color.BROWN);
...
final Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
final KeyValue kv = new
   KeyValue(rectBasicTimeline.xProperty(), 300);
final KeyFrame kf = new KeyFrame(Duration.millis(500),
   kv);
timeline.getKeyFrames().add(kf);
timeline.play();
```

Using the Transition Classes

Class	Description
TranslateTransition	Translates (moves) a node from one location to another
RotateTransition	Rotates a node
ScaleTransition	Scales (increases or decreases the size of) a node
FadeTransition	Fades (increases or decreases the opacity of) a node
PathTransition	Moves a node along a geometric path
SequentialTransition	Allows you to define a sequential series of transitions
PauseTransition	Used in a SequentialTransition to wait for a period of time
ParallelTransition	Allows you to define a parallel series of transitions
StrokeTransition	Changes the stroke of a color over a period of time
FillTransition	Allows a shape to fill over a period of time

Animated Transition: Path Transition

```
final Rectangle rectPath = new Rectangle (0, 0, 40, 40);
rectPath.setArcHeight(10);
rectPath.setArcWidth(10);
rectPath.setFill(Color.ORANGE);
...
Path path = new Path();
path.getElements().add(new MoveTo(20,20));
path.getElements().add(new CubicCurveTo(380, 0, 380, 120, 200, 120));
path.getElements().add(new CubicCurveTo(0, 120, 0, 240, 380, 240));
PathTransition pathTransition = new PathTransition();
pathTransition.setDuration(Duration.millis(4000));
pathTransition.setPath(path);
pathTransition.setNode(rectPath);
pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_
    TO_TANGENT);
pathTransition.setCycleCount(Timeline.INDEFINITE);
pathTransition.setAutoReverse(true);
pathTransition.play();
```



A path transition moves a node along a path from one end to the other over a given period of time.

In the example in the slide, a path transition is applied to a rectangle. The animation is reversed when the rectangle reaches the end of the path. In the code, a rectangle with rounded corners is created, and then a new path animation is created and applied to the rectangle. Setting the orientation `ORTHOGONAL_TO_TANGENT` keeps the node perpendicular to the path's tangent along the geometric path.

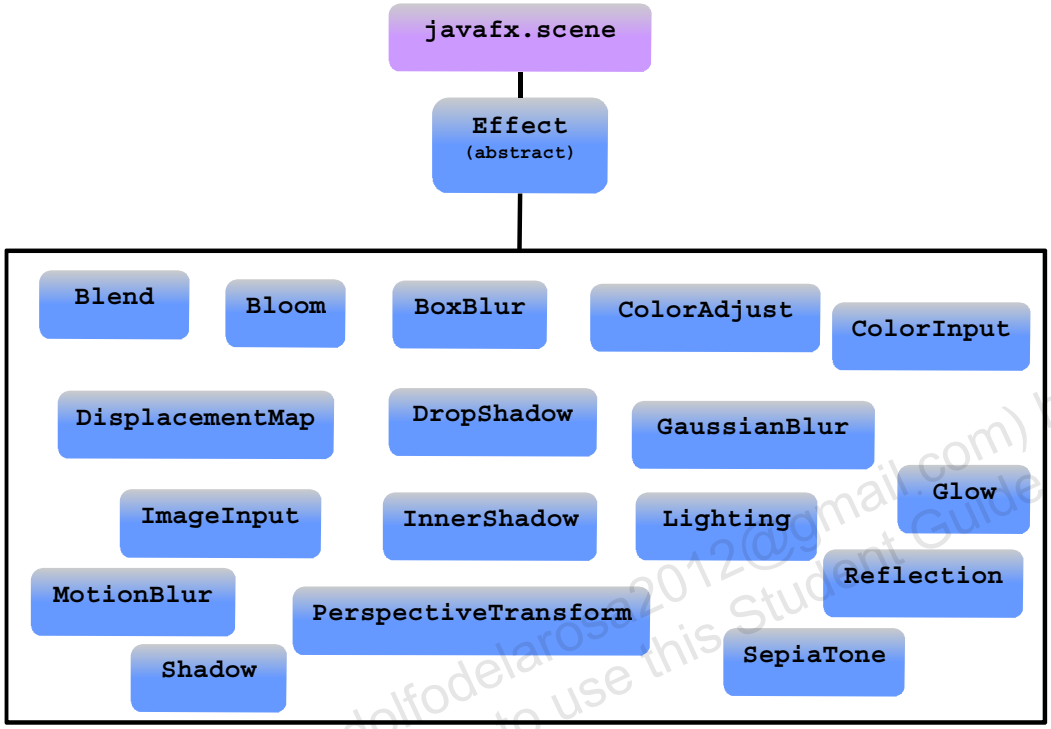
Effects

The Effect class `javafx.scene.effect` creates filter effects that operate on an image and on any node they are applied to and use algorithms that apply to pixels. Effects leverage SSE, OpenGL, D3D, or plain Java. You can apply effects to any node, and effects can be combined using the Blend effect.

Effects are graphical algorithms that produce an image, usually as a modification to a source image.

Effects can be associated with a node by setting the node's effect attribute.

`Effect` Is Applied to a Node



The `javafx.scene.effect` package provides the set of classes for attaching graphical filter effects to JavaFX scene graph nodes.

```
public abstract class Effect
extends java.lang.Object
```

`Effect` is the abstract base class for all effect implementations. An effect is a graphical algorithm that produces an image, typically as a modification of a source image. An effect can be associated with a scene graph node by setting the `Node.effect` attribute. Some effects change the color properties of the source pixels (such as `ColorAdjust`), others combine multiple images together (such as `Blend`), and still others warp or move the pixels of the source image around (such as `DisplacementMap` or `PerspectiveTransform`).

Each effect has at least one input defined. The input can be set to another effect to chain the effects together and combine their results, or it can be left unspecified—in which case, the effect will operate on a graphical rendering of the node it is attached to.

Note: This is a conditional feature. The `ConditionalFeature.EFFECT` indicates that filter effects are available on the platform. If an application uses an effect on a platform that does not support it, the effect will be ignored.

Effects can be combined using the `Blend` effect.

Effect: Example

```
static Node innerShadow() {
        InnerShadow is = new InnerShadow();
        is.setOffsetX(2.0f);
        is.setOffsetY(2.0f);

        Text t = new Text();
        t.setEffect(is);
        t.setX(20);
        t.setY(100);
        t.setText("Inner Shadow");
        t.setFill(Color.RED);
        t.setFont(Font.font("null", FontWeight.BOLD, 80));

        t.setTranslateX(300);
        t.setTranslateY(300);

        return t;
    }
```

An inner shadow is an effect that renders a shadow inside the edges of the given content with the specified color, radius, and offset.

Quiz

Which two classes are types of transitions?

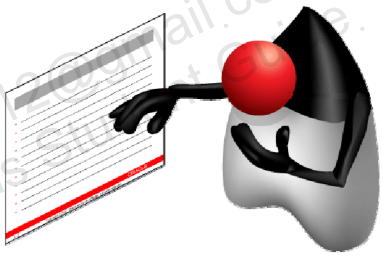
a. PerspectiveTransform, PathTransition
b. PathTransform, RotateTransform
c. SequentialTransition, PauseTransition
d. PathTransition, BlendTransition

Answer: c

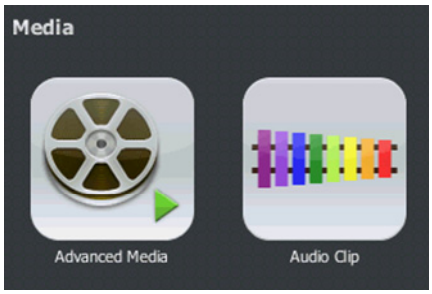
a. PerspectiveTransform is an effect.
b. PathTransform and RotateTransform are not transitions.
c. SequentialTransition and PauseTransition are transitions.
d. BlendTransition is not a transition.

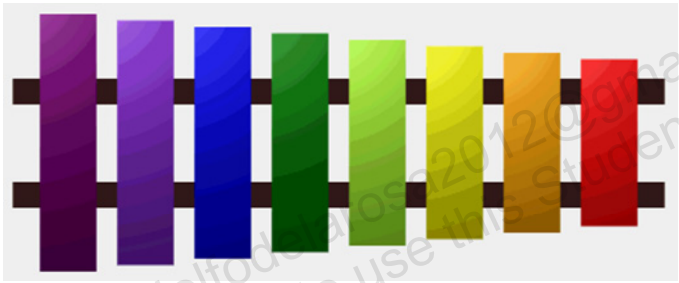
Topics

- Using animation and effects in an application
- Describing how to implement media in an application
- Describing the benefits of using WebView



Media





The `javafx.scene.media` package enables developers to create media applications that provide media playback in the desktop window or in a web page on supported platforms.

The media formats that are currently supported are the following:

- **Audio:** MP3, AIFF containing uncompressed PCM, WAV containing uncompressed PCM
- **Video:** FLV containing VP6 video and MP3 audio

Some of the features supported by the media stack include the following:

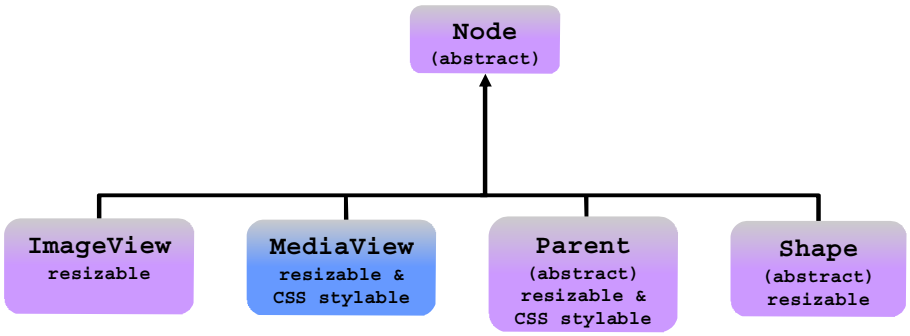
- VP6 + MP3 using a FLV container
- MP3 for audio
- HTTP, FILE protocol support
- Progressive download
- Seeking
- Buffer progress
- Playback functions (Play, Pause, Stop, Volume, Mute, Balance, Equalizer)

Building a Media Player

- `Media`: A media resource containing information about the media, such as its source, resolution, and metadata
- `MediaPlayer`: The key component providing the controls for playing media
- `MediaView`: A Node object that supports animation, translucency, and effects

Each element of the media functionality is available through the JavaFX API. The media classes are interdependent and are used in combination to create an embedded media player.

MediaView Is a Node




```
public class MediaView
extends Node
```

`MediaView` is a node that provides a view of media being played by a MediaPlayer.

MediaView

```
public MediaView(MediaPlayer mediaPlayer)
```

Creates a `MediaView` instance associated with the specified `MediaPlayer`. Equivalent to

```
MediaPlayer player; // initialization omitted
MediaView view = new MediaView();
view.setPlayer(player);
```

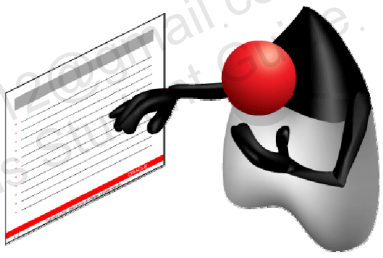
Parameters:

`mediaPlayer` - the `MediaPlayer` the playback of which is to be viewed via this class.

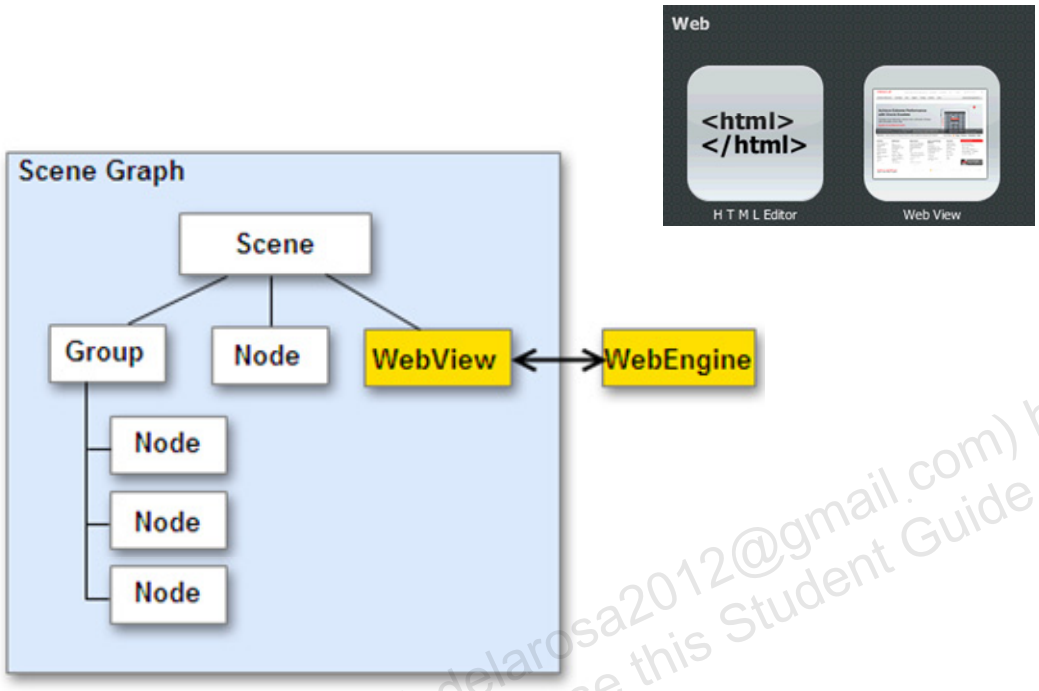
ORACLE

Topics

- Using animation and effects in an application
- Describing how to implement multimedia in an application
- Describing the benefits of using WebView



WebView

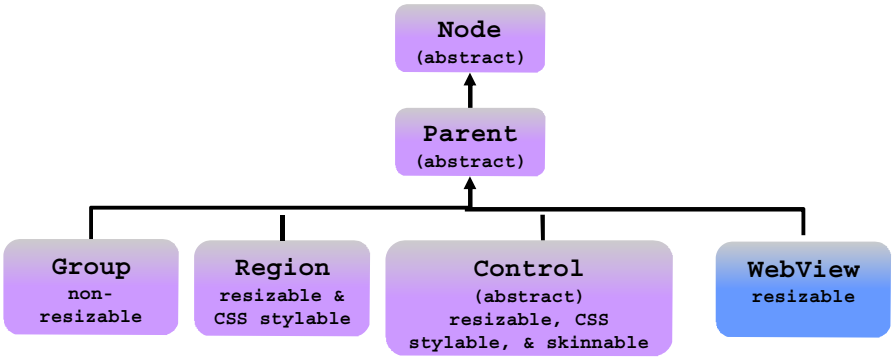


The JavaFX SDK introduces the embedded browser, which is a user interface component that provides a web viewer and full browsing functionality through its API. The embedded browser component is based on WebKit, an open source web browser engine. It supports Cascading Style Sheets (CSS), JavaScript, Document Object Model (DOM), and the following features of HTML5: rendering canvas and timed media playback.

The embedded browser enables you to perform the following tasks in your JavaFX applications:

- Render HTML content from local and remote URLs.
- Execute JavaScript commands.
- Access the document model from Java code.
- Handle events.
- Manage web pop-up windows.
- Apply effects to the embedded browser.

WebView Is a Node




```
public final class WebView
extends Node
```

`WebView` is a node that manages a WebEngine and displays its content. The associated WebEngine is created automatically at construction time and cannot be changed afterward.

`WebView` handles mouse and some keyboard events, and manages scrolling automatically, so there is no need to put it into a ScrollPane.

`WebView` objects must be created and accessed solely from the FX thread.

WebView: Example

```
WebView webView = new WebView();
WebEngine webEngine = webView.getEngine();
webEngine.load("http://javafx.com");
// add WebView to the scene
```

Use Cases for WebView

- Displaying oAuth login dialogs (Facebook, Flickr, and so on)
- Embedding existing web forms or UI
- Existing GWT pages for form submission
- Embedding maps
- Rendering FX content over nodes on a web page
- Displaying and editing rich text
- Complicated text layout

Quiz

An embedded browser can access only a remote URL.

a. True
b. False

Answer: b

a. Incorrect answer. An embedded browser can access a remote and local URL.
b. Correct answer. It can access a remote and local URL.

Summary

In this lesson, you should have learned how to:

- Use animation and effects in an application
- Describe how to implement media in an application
- Describe the benefits of using WebView

Practice 6: Overview

6-1: Adding a Fade Transition

7 JavaFX Tables and Client GUI

ORACLE

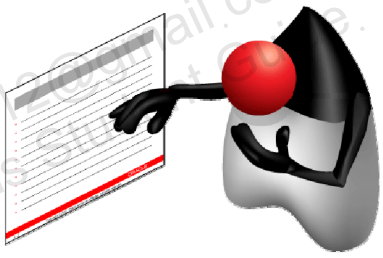
Objectives

After completing this lesson, you should be able to:

- Create a table and custom table cell
- Apply CSS to a table
- Recognize JavaFX development practices
- Describe the BrokerTool application interface
- Identify the JavaFX components and charts to use in the BrokerTool interface

Topics

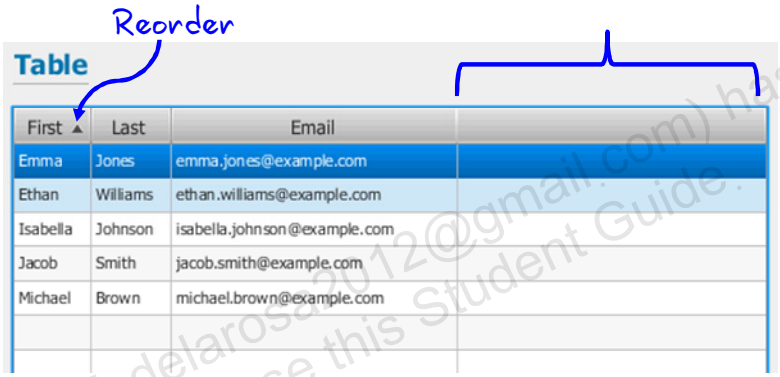
- Develop a table and custom table cell
- Add CSS to a table
- Describe the BrokerTool application interface



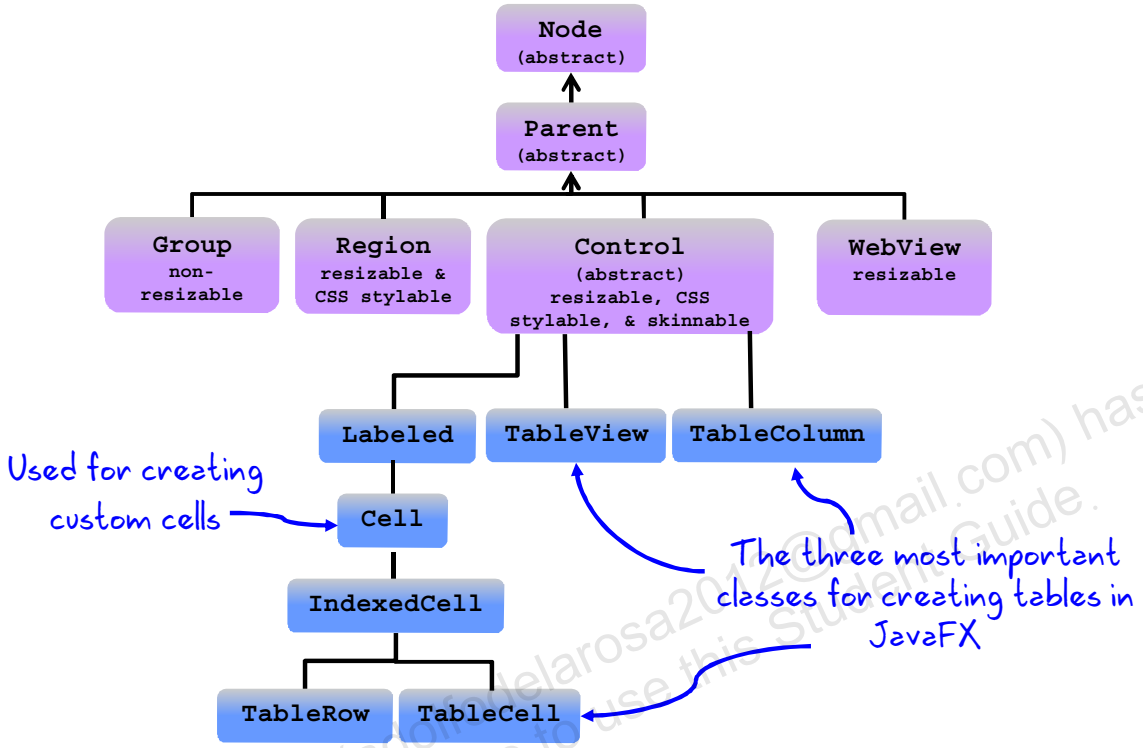
Tables in JavaFX

Tables in JavaFX are very powerful and support the following:

- Column reordering by user
- Multiple column sorting
- Width resizing
- Cell factories for customizing cell content



TableView is a Node



Several classes in the JavaFX SDK API are designed to represent data in a tabular form. The most important classes for creating tables in JavaFX applications are `TableView`, `TableColumn`, and `TableCell`. You can populate a table by implementing the data model and by applying a cell factory. The table classes provide built-in capabilities to sort data in columns and to resize columns when necessary. The `TableView` control has a number of features, including:

- `TableColumn` API:
  - Support for cell factories to easily customize cell contents in both rendering and editing states
  - Specification of `minWidth/ prefWidth/maxWidth` and also fixed-width columns
  - Width resizing by the user at runtime
  - Column reordering by the user at runtime
  - Built-in support for column nesting
- Different resizing policies to determine what happens when the user resizes columns
- Support for multiple column sorting by clicking the column header (press the Shift key while clicking a header to sort by multiple columns)

Creating a Table

`TableView` and `TableColumn` are the minimum classes required to create a table.

- `TableView<S> S`: The type of the objects contained in the `TableView` items list
- `TableColumn<S,T>`
  - `S`: The type of the `TableView` generic type (that is, `S` == `TableView<S>`)
  - `T`: The type of content in all cells in this `TableColumn`

The next slide shows the code example.

A `TableView` is made up of a number of `TableColumn` instances. Each `TableColumn` in a table is responsible for displaying (and editing) the contents of that column. In addition to being responsible for displaying and editing data for a single column, a `TableColumn` also contains the necessary properties to:

- Be resized (using `minWidth/prefWidth/maxWidth` and fixed-width properties)
- Have its `visibility` toggled
- Display `header text`
- Display any `nested columns` it may contain
- Have a `context menu` when the user right-clicks the column header area
- Have the contents of the table be sorted (using `comparator`, `sortable`, and `sortType`)

When you create a `TableColumn` instance, perhaps the two most important properties to set are the column `text` (what to show in the column header area) and the column `cell value factory` (which is used to populate individual cells in the column).

The entire code sample is displayed in the following slide.

Creating a Table: Code Example

```
public class Person {
 private StringProperty firstName;
 public void setFirstName(String value) { firstNameProperty().set(value); }
 public String getFirstName() { return firstNameProperty().get(); }
 public StringProperty firstNameProperty() {
   if (firstName == null) firstName = new SimpleStringProperty(this, "firstName");
   return firstName;
 }
 private StringProperty lastName;
 public void setLastName(String value) { lastNameProperty().set(value); }
 public String getLastName() { return lastNameProperty().get(); }
 public StringProperty lastNameProperty() {
    if (lastName == null) lastName = new SimpleStringProperty(this, "lastName");
    return lastName;
 }
}
...
TableView<Person> table = new TableView<Person>();
TableColumn<Person,String> firstNameCol
   = new TableColumn<Person,String>("First Name");
…
table.getColumns().setAll(firstNameCol, lastNameCol);
```

TableCell<S,T>

- Represents a single row/column intersection in a `TableView`
- Contains the following properties:
  - `tableColumn:` The `TableColumn` instance that backs this `TableCell`
  - `tableView:` The `TableView` associated with this `TableCell`
  - `tableRow:` The `TableRow` in which this `TableCell` is currently placed

To represent this intersection, a TableCell contains an index property, as well as a tableColumn property. In addition, a TableCell instance knows what TableRow it exists in.

Cell<T>

- Is used for an individual cell in a `TableView`
- Every cell is associated with a single data item represented by the `item` property.
- A cell is responsible for rendering any item that resides within it, which is usually `text`.
- A cell is a control and is essentially a "model" (in MVC terms).
- Enables customization by using a cell factory

The `Cell` API is used for virtualized controls such as `ListView`, `TreeView`, and `TableView`. A `Cell` is a `Labeled` Control, and is used to render a single “row” inside a `ListView`, `TreeView` or `TableView`. Cells are also used for each individual “cell” inside a `TableView` (that is, each row/column intersection).

Note: For details, see the JavaDoc for each individual control.

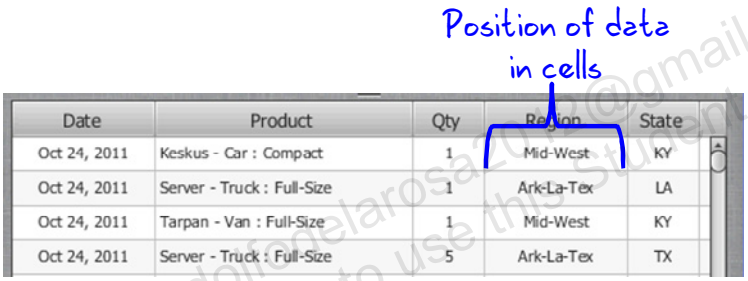
Every `Cell` is associated with a single data item (represented by the item property). The `Cell` is responsible for rendering that item and, where appropriate, for editing the item. An item within a `Cell` can be represented by text or some other control such as a `CheckBox`, `ChoiceBox`, or any other `Node` such as an `HBox`, `GridPane`, or even a `Rectangle`.

Because `TreeView`, `ListView`, `TableView`, and other such controls can potentially be used for displaying extremely large amounts of data, it is not practical to create an actual `Cell` for every item in the control. You can represent extremely large data sets by using very few Cells. Each `Cell` is “recycled,” or reused, which makes this control virtualized.

Because `Cell` is a `Control`, it is essentially a “model.” Its skin is responsible for defining the look and layout, while the Behavior is responsible for handling all input events and using that information to modify the control state. Also, the `Cell` is styled from CSS just like any other control. However, it is not necessary to implement a skin for most uses of a `Cell`.

Cell Factory

- To specialize the `Cell` used for the `TableView`, you must provide an implementation of the `cellFactory` callback function defined on the `TableView`.
- The cell factory is called by the platform whenever it determines that a new cell needs to be created.
- The implementation of the cell factory is responsible for creating a `Cell` instance and also configuring that `Cell` so that it reacts to changes in its state.



It is the responsibility of the various virtualized containers' skins to render the default representation of the `Cell` item. For example, the `ListView` by default converts the item to a String and calls `Labeled.setText(java.lang.String)` with this value. If you want to specialize the `Cell` used for the `ListView` (for example), then you must provide an implementation of the `cellFactory` callback function defined on the `ListView`. A similar API exists on most controls that use Cells (for example, `TreeView`, `TableView`, `TableColumn`, and `ListView`).

The cell factory is called by the platform whenever it determines that a new cell needs to be created. For example, perhaps your ListView has 10 million items. Creating all 10 million cells would be prohibitively expensive. So instead, the `ListView` skin implementation might only create just enough cells to fit the visual space. If the `ListView` is resized to be larger, the system will determine that it needs to create some additional cells. In this case, it will call the `cellFactory` callback function (if one is provided) to create the `Cell` implementation that should be used. If no cell factory is provided, the built-in default implementation will be used.

The implementation of the cell factory is then responsible not just for creating a `Cell` instance, but also for configuring that `Cell` so that it reacts to changes in its state. In the example in the slide, the cell data is positioned vertically and horizontally within the cell using the `textAlignmentProperty` inherited from the `Labeled` class.

Custom Cell Factory: FXML Example

```
<TableView fx:id="salesTable" >
    <columns>
        <TableColumn text="Date" prefWidth="100">
           <cellValueFactory><PropertyValueFactory
    property="date" /></cellValueFactory>
           <cellFactory>
               <FormattedTableCellFactory alignment="center">
                     <format><DateFormat
    fx:factory="getDateInstance"/></format>
               </FormattedTableCellFactory>
           </cellFactory>
        </TableColumn>
        <TableColumn text="Product" prefWidth="180">
           <cellValueFactory><PropertyValueFactory
    property="productId" /></cellValueFactory>
           <cellFactory><FormattedTableCellFactory
    alignment="left"/></cellFactory>
        </TableColumn>
```


This example represents cellFactory alignment for the columns.

Custom Cell: Example

```
public class FormattedTableCellFactory<S,T> implements
   Callback<TableColumn<S,T>, TableCell<S,T>> {
   private TextAlignment alignment;
   private Format format;

   public TextAlignment getAlignment() {
       return alignment;
   }

   public void setAlignment(TextAlignment alignment) {
       this.alignment = alignment;
   }

   public Format getFormat() {
       return format;
   }

   public void setFormat(Format format) {
       this.format = format;
   }
```


This example represents an implementation of the Callback class and creates instances of the TextAlignment and Format classes using these parameters:

- S: The type of the TableView generic type (that is, S == TableView<S>)
- T: The type of the content in all cells in this TableColumn

Table Data Model

`ObservableList` is the underlying data model for the `TableView`.

```
ObservableList<Person> teamMembers = getTeamMembers();
 table.setItems(teamMembers);

 TableColumn<Person,String> firstNameCol =
   new TableColumn<Person,String>("First Name");
   firstNameCol.setCellValueFactory
   (new PropertyValueFactory("firstName"));
 TableColumn<Person,String> lastNameCol =
   new TableColumn<Person,String>("Last Name");
   lastNameCol.setCellValueFactory
   (new PropertyValueFactory("lastName"));

 table.getColumns().setAll(firstNameCol, lastNameCol);
```

ORACLE

The `TableView` instance is defined as follows:

```
TableView<Person> table = new TableView<Person>();
```

With the basic table defined, look at the data model. This example uses an `ObservableList`. You can set a list directly into the `TableView`:

```
ObservableList<Person> teamMembers = getTeamMembers();
 table.setItems(teamMembers);
```

With the items set, `TableView` automatically updates whenever the `teamMembers` list changes. If the items list is available before the `TableView` is instantiated, it is possible to pass it directly into the constructor.

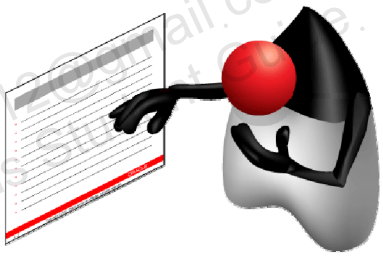
At this point, you have a `TableView` connected to observe the `teamMembers observableList`. The missing ingredient now is the means of splitting out the data contained within the model and representing it in one or more `TableColumn` instances. To create a two-column `TableView` to show the `firstName` and `lastName` properties, you extend the preceding code sample as follows:

```
ObservableList<Person> teamMembers = ...;
 table.setItems(teamMembers);
 TableColumn<Person,String> firstNameCol = new
TableColumn<Person,String>("First Name");
 firstNameCol.setCellValueFactory(new
PropertyValueFactory("firstName"));
 TableColumn<Person,String> lastNameCol = new
TableColumn<Person,String>("Last Name");
 lastNameCol.setCellValueFactory(new
PropertyValueFactory("lastName"));
 table.getColumns().setAll(firstNameCol, lastNameCol);
```

With this code, you have fully defined the minimum properties required to create a `TableView` instance. Running the code results in a `TableView` displayed with two columns: `firstName` and `lastName`. No other properties of the `Person` class are shown because no `TableColumns` are defined.

Topics

- Develop a table and custom table cell
- Add CSS to a table
- Describe the BrokerTool application interface



Benefits of Using CSS with Tables

The benefits of using CSS to style a table (or other control):

- Both time-efficient and memory-efficient for large data sets
- Easy to build and use libraries for custom cells
- Easy to customize cell visuals
- Easy to customize display formatting (for example, 12.34 as $12.34 or 1234%)
- Easy to extend for custom visuals
- Easy to have "panels" of data for the visuals
- Easy to animate the cell size or other properties

CSS and Tables

Use CSS to set cell colors:

- Each cell can be styled directly from CSS. To change the default background of every cell in a TableView to white, you can use the following CSS:

```
.table-cell {
  -fx-padding: 3 3 3 3;
  -fx-background-color: white;
}
```

- To set the color of selected TableView cells to blue, you can add this to your CSS file:

```
.table-cell:selected {
  -fx-background-color: blue;
}
```


Each Cell can be styled directly from CSS.

For `table-cell:selected` to work, you must have `cellSelectionEnabled` set to `true`.

Most Cell implementations extend from `IndexedCell` rather than `Cell`. `IndexedCell` adds two other pseudoclass states: "odd" and "even." With these, you can obtain alternate row striping by using something like the following in your CSS file:

```
.table-cell:odd {
   -fx-background-color: grey;
}
```

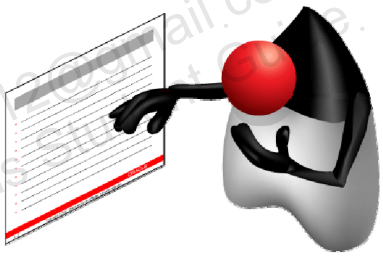
None of these examples requires code changes. Simply update your CSS file to alter the colors. You can also use "hover" and other pseudoclasses in CSS (as with other controls).

One approach to formatting a list of numbers is to use style classes. Suppose you have an `ObservableList` of Numbers to display in a `ListView` and you want to color all of the negative values red and all positive or 0 values black. One way to achieve this is with a custom `cellFactory` that changes the `styleClass` of the `Cell` based on whether the value is negative or positive. This is as simple as adding code to test if the number in the cell is negative, and adding a "negative" `styleClass`. If the number is not negative, the "negative" string should be removed. With this approach, the colors can be defined from CSS, thus enabling simple customization. The CSS file would include something like the following:

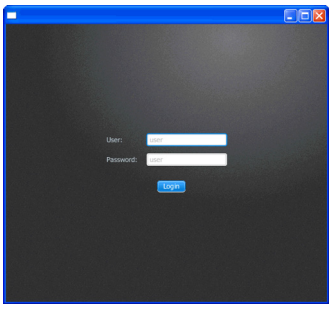
```
.table-cell {
   -fx-text-fill: black;
}
.table-cell .negative {
   -fx-text-fill: red;
}
```

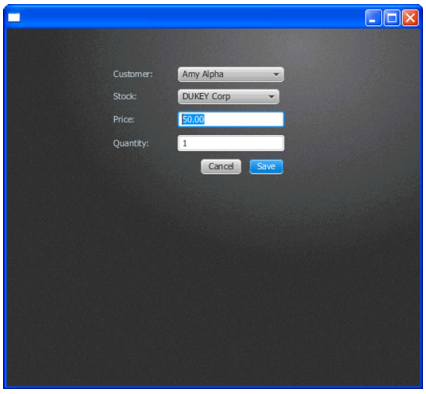
Topics

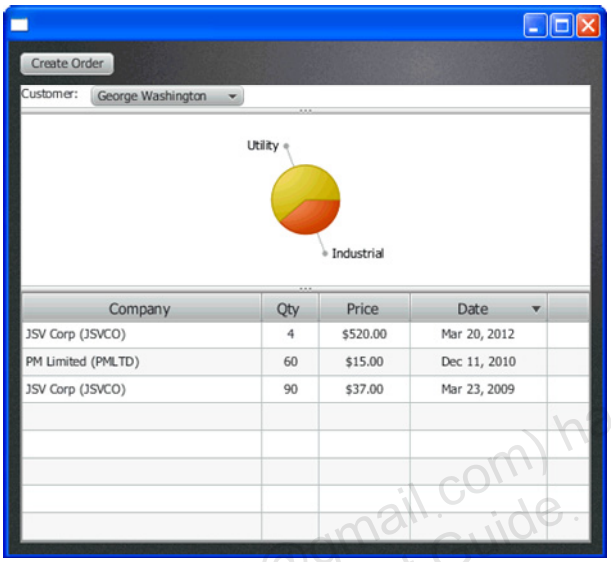
- Develop a table and custom table cell
- Add CSS to a table
- Describe the BrokerTool application interface



BrokerTool Application







- Login window
- Main window
- Order form

JavaFX Development Practices

- Avoid mutable properties that are also updated by the skin.
- Use POJOs as the control's model whenever possible.
- To define style, you should choose CSS rather than explicit API.
- Try to be "deceptively simple" and emphasize content rather than graphics.

Using POJOs as the control's model helps keep the processes separate from the UI.

When you choose CSS rather than explicit API, it is much easier to maintain the CSS styling if you have to make changes later.

Keep the graphics simple and intuitive so the UI is easier for the user to understand.

Application in MVC Terms

- **Model:** JPA
  - Contains the Broker, Customer, Shares, and Stock classes
- **View:** FXML files
- **Controller:** Java files

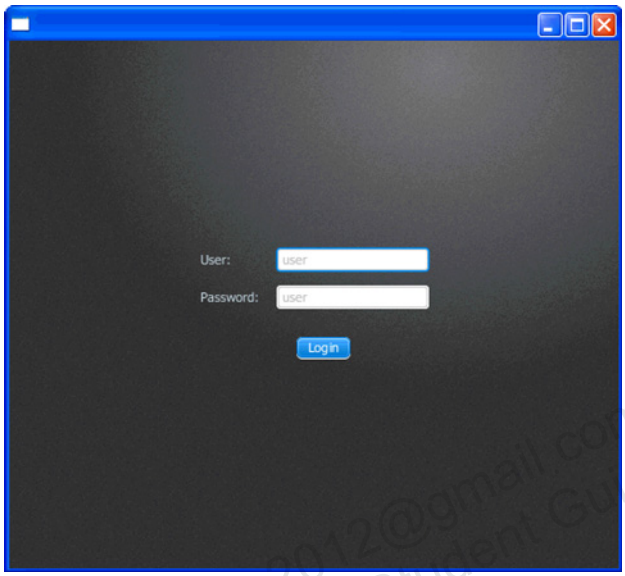
The BrokerTool application follows the Model View Controller design pattern. The model consists of the JPA model, the view is all of the FXML classes, and the controllers are all of the Java classes.

Login Window

View: `login.fxml`

Controls

- Label
- Button
- TextField



The login window is created in `login.fxml`.

- `login.java`
- `AuthenticationException.java`
- `AnimatedPageView.java`

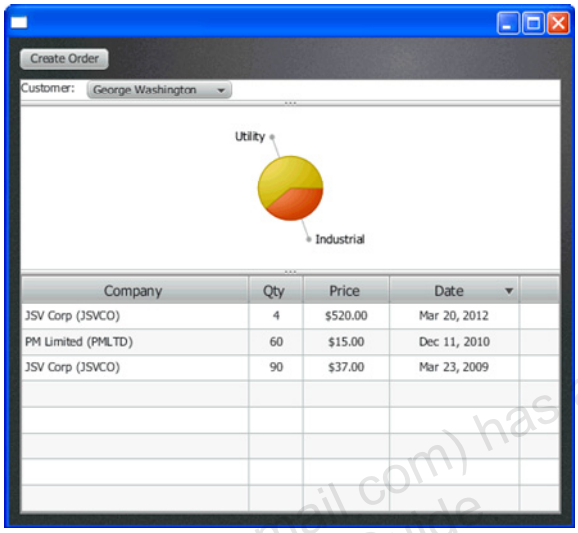
Main Window

View

- BrokerDashboard.fxml
- BrokerToolClient.fxml
- BrokerToolTop.fxml

Controller

- BrokerDashboard.java
- BrokerToolClient.java
- BrokerToolTop.java
- BrokerToolClientApp.java
- …



The BrokerTool application main view is created in a combination of three classes:

- BrokerDashboard.fxml
- BrokerToolClient.fxml
- BrokertToolTop.fxml

The controllers are created in several classes, including:

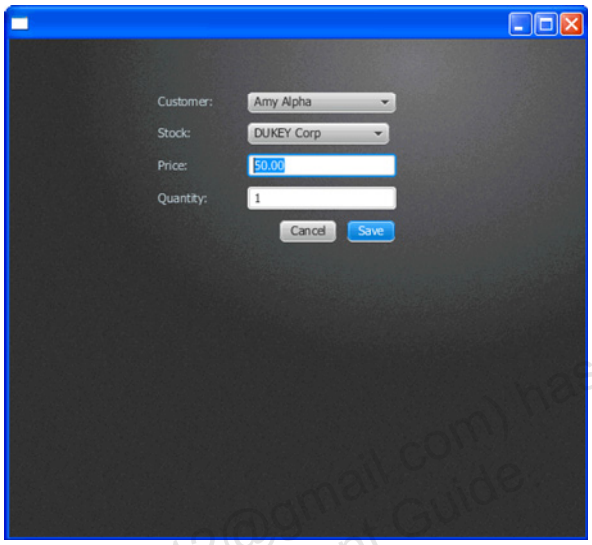
- BrokerDashboard.java
- BrokerToolClient.java
- BrokertToolTop.java
- BrokerToolClientApp.java
- GetSharesTask.java
- GetSharesService.java
- FormattedTableCellFactory.java
- StockTableCellFactory.java
- AnimatedPageView.java

Order Form Window

View: `OrderForm.fxml`

Controls

- Label
- Button
- ChoiceBox
- TextField



The `OrderForm.fxml` class is the view for the Order Form. The controllers for `OrderForm.fxml` include:

- `OrderForm.java`
- `AnimatedPageView.java`
- `BrokerToolClient.java`
- `BrokerToolClientApp.java`
- `GetSharesService.java`
- `GetSharesTask.java`

Quiz

Which are considered the three most important classes for creating tables in JavaFX?

a. `TableView`
b. `TableColumn`
c. `TableCell`
d. `TableRow`
e. `Labeled`

Answer: a, b, c

Summary

In this lesson, you should have learned how to:

- Create a table and custom table cell
- Apply CSS to a table
- Recognize JavaFX development practices
- Describe the BrokerTool application interface
- Identify the JavaFX components and charts to use in the BrokerTool interface

Practice 7: Overview

- 7-1: Creating a Simple Table View
- 7-2: Styling a Smart Table
- 7-3: Creating a Complete BrokerTool Interface

8

JavaFX Concurrency and Binding

ORACLE

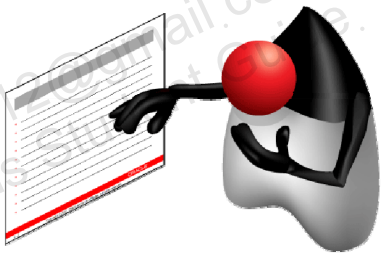
Objectives

After completing this lesson, you should be able to:

- Describe and implement JavaFX concurrency
- Describe binding and properties in JavaFX, including simple binding and bi-directional binding

Topics

- Working with JavaFX concurrency
- Binding in JavaFX



Concurrency and JavaFX

The `javafx.concurrent` package handles multithreaded code that interacts correctly with the UI and ensures that this interaction happens on the correct thread.

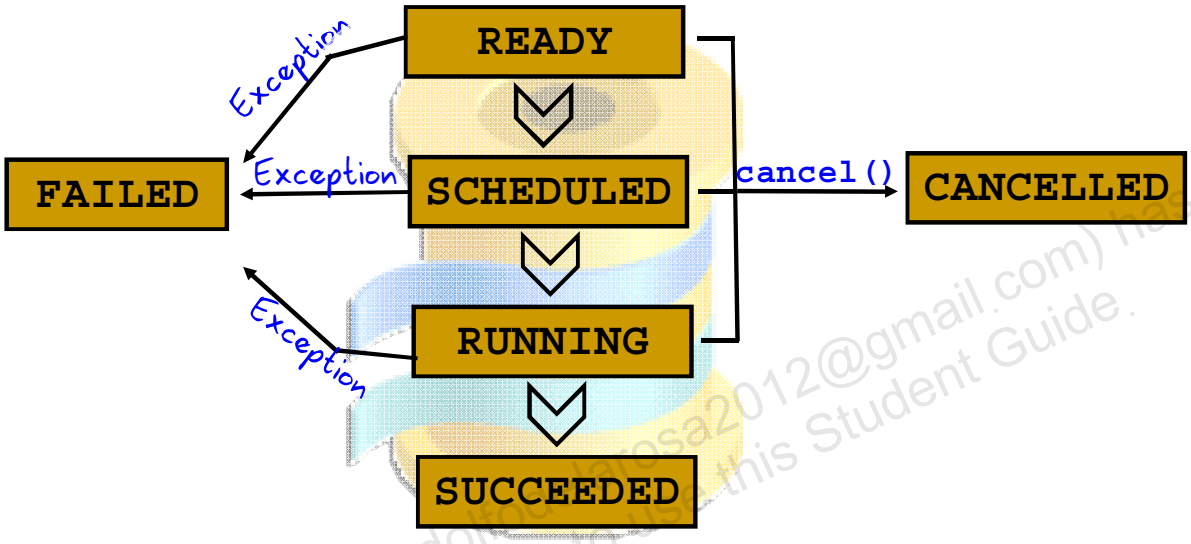
- The `Worker` interface
  - **`Task`:** Is a fully observable implementation of the `java.util.concurrent.FutureTask` class; enables implementation of asynchronous tasks
  - **`Service`:** Executes tasks

The Java platform provides a complete set of concurrency libraries available through the `java.util.concurrent` package. The `javafx.concurrent` package leverages the existing API by taking into account the JavaFX Application thread and other constraints faced by GUI developers.

The `javafx.concurrent` package consists of the Worker interface and two basic classes, Task and Service, both of which implement the Worker interface. The Worker interface provides APIs that are useful for a background worker to communicate with the UI. The Task class is a fully observable implementation of the `java.util.concurrent.FutureTask` class. The Task class enables developers to implement asynchronous tasks in JavaFX applications. The Service class executes tasks.

Worker Interface

A `Worker` is an object that performs work in background threads. The `Worker` main life cycle includes `READY`, `SCHEDULED`, and `RUNNING`.



ORACLE

The Worker interface defines an object that performs some work in one or more background threads. The state of the Worker object is observable and usable from the JavaFX Application thread.

The life cycle of the Worker object is defined as follows. When created, the Worker object is in the READY state. After being scheduled for work, the Worker object transitions to the SCHEDULED state. After that, when the Worker object is performing the work, its state becomes RUNNING. Note that even when the Worker object is immediately started without being scheduled, it first transitions to the SCHEDULED state and then to the RUNNING state. The state of a Worker object that completes successfully is SUCCEEDED, and the value property is set to the result of this Worker object. Otherwise, if any exceptions are thrown during the execution of the Worker object, its state becomes FAILED and the exception property is set to the type of the exception that occurred. In any state, the Worker object can be interrupted using the cancel method, which puts the Worker object in the CANCELLED state.

The progress of the work being done by the Worker object can be obtained through three different properties such as totalWork, workDone, and progress.

For more information about the range of the parameter values, see the API documentation.

`Task` Class

`Task` can be started in one of the following three ways (the first two listed are the preferred methods):

- Using the ExecutorService API: `ExecutorService.submit(task);`
- Using the run method: `task.run();`
- Starting a thread with the given task as a parameter: `new Thread(task).start();`

The `Task` class defines a thread and task that cannot be reused.

- You can call the `Task` object directly by using `FutureTask.run()`.
  - `public abstract class Task<V> extends java.util.concurrent.FutureTask<V> implements Worker<V>, EventTarget`

Tasks are used to implement the logic of work that needs to be done on a background thread. You first need to extend the `Task` class. Your implementation of the `Task` class must override the `call` method.

Note that the `Task` class fits into the Java concurrency libraries because it inherits from the `java.utils.concurrent.FutureTask` class, which implements the `Runnable` interface. For this reason, a `Task` object can be used within the Java concurrency Executor API and also can be passed to a thread as a parameter. You can call the `Task` object directly by using the `FutureTask.run()` method, which enables calling this task from another background thread.

Example: Create the `Task`

```
import javafx.concurrent.Task;

public class CounterTask extends Task<Void>{

    @Override
    public Void call(){
        final int max = 10000000;
        updateProgress(0, max);
        for (int i=1; i<=max; i++){
            updateProgress(i, max);
        }

        return null;
    }

}
```

Overrides the `call()` method invoked on the background thread

In the example in the slide, the task is created by overriding the `call()` method. The `call()` method is invoked on the background thread; therefore, this method can only manipulate states that are safe to read and write from a background thread. For example, manipulating an active scene graph from the `call` method throws runtime exceptions. On the other hand, the `Task` class is designed to be used with JavaFX GUI applications, and it ensures that any changes to public properties, change notifications for errors, event handlers, and states occur on the JavaFX Application thread. Inside the `call` method, you can use the `updateProgress`, `updateMessage`, `updateTitle` methods, which update the values of the corresponding properties on the JavaFX Application thread.

The example is located in `D:\labs\08-FXConcurrency\examples\CounterBar`.

Example: Run the `Task`

```
ExecutorService es = Executors.newSingleThreadExecutor();

. . .

  @Override
  public void handle(ActionEvent event) {
      System.out.println("Count Started");
      bar.progressProperty().bind(
                        countTask.progressProperty());
      es.execute(countTask);
  }
});
```

ExecutorService

In the example in the slide, the task is executed by `ExecutorService`.

The example is located in `D:\labs\08-FXConcurrency\examples\CounterBar`.

`Service` Class

- The `Service` class is designed to execute a `Task` object on one or several background threads.
- `Service` class methods and states must only be accessed on the JavaFX application thread.
- It helps developers implement correct interactions between background threads and the JavaFX Application thread.
- You can start, stop, cancel, and restart a `Service` as needed.
- Use `Service.start()` to start a `Service`.
- `Service` can run a task more than once.

Using the Service class, you can observe the state of the background work and (optionally) cancel it. Later, you can reset the service and restart it. Thus, the service can be defined declaratively and restarted on demand.

`Service` Class Execution

The `Service` can be executed in one of the following three ways:

- By an `Executor` object, if it is specified for the given service
- By a daemon thread, if no `Executor` is specified
- By a custom executor such as a `ThreadPoolExecutor`

If a `java.util.concurrent.Executor` is specified on the Service, it will be used to actually execute the background worker. Otherwise, a daemon thread will be created and executed. If you want to create non-daemon threads, specify a custom `Executor` (for example, you could use a `ThreadPoolExecutor`).

Example: Create the `Service`

```
public class CounterService extends Service<Void>{
   //Create an instance of Task that returns the CounterTask.
    @Override
    protected Task<Void> createTask(){
        CounterTask ct = new CounterTask();
        return ct;
    }

}
```


In the example in the slide, the `Service` is built and creates an instance of `Task` that returns `CounterTask`.

The example is located in `D:\labs\08-FXConcurrency\examples`.

Example: Run the `Service`

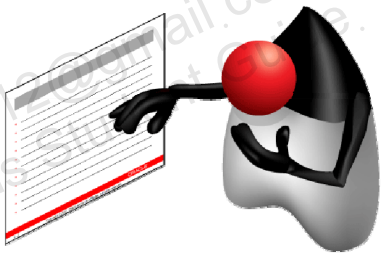
```
CounterService cs = new CounterService();

...

@Override
  public void handle(ActionEvent event) {
     System.out.println("Count Started");

     bar.progressProperty().bind(cs.progressProperty());
     if (cs.getState() == State.READY){
         cs.start();
     }
  }
});
```

Starts the service

In the example in the slide, the Service is started by the `start()` method.
The example is located in `D:\labs\08-FXConcurrency\examples`.

Topics

- Working with JavaFX concurrency
- Binding in JavaFX



Data Binding in JavaFX

- Simplifies the task of synchronizing the view state and domain data model
- Enables enterprise design patterns
- Enables rapid application development in visual tools
- A binding observes its list of dependencies for changes, and then updates itself automatically after a change has been detected.
- The binding API uses a High-Level API that provides a simple way to create bindings for the most common use cases.

ORACLE

Binding is a powerful mechanism for expressing direct relationships between variables. When objects participate in bindings, changes made to one object will automatically be reflected in another object. This can be useful in a variety of applications. For example, binding can be used in a bill invoice tracking program, where the total of all bills is automatically updated whenever an individual bill is changed; or, binding can be used in a GUI that automatically keeps its display synchronized with the application's underlying data.

Data binding lends itself to enterprise design patterns such as the Presentation Model and Supervising Controller, which are patterns similar to the MVC pattern. The Supervising Controller leverages the data-binding facilities of the view framework, thus leading to fewer methods in the view that need to be mocked up.

Bindings are assembled from one or more sources called *dependencies*. A binding observes its list of dependencies for changes, and then updates itself automatically after a change has been detected.

The High-Level API provides a simple way to create bindings for the most common use cases. Its syntax is easy to learn and use, especially in environments that provide code completion, such as the NetBeans IDE. It works for functions such as arithmetic operations, boolean operations, calculating minimum and maximum values, comparisons, and so on.

Data Binding in JavaFX

In the following example, `bind` is used to tie the progress bar to the `cs` (CounterService).

```
CounterService cs = new CounterService();

...

@Override
  public void handle(ActionEvent event) {
     System.out.println("Count Started");
     bar.progressProperty().bind(cs.progressProperty());
     if (cs.getState() == State.READY){
         cs.start();
     }
  }
});
```

Bind to the cs.progressProperty

In the example in the slide, `bind` is used to tie the progress bar to the counter service. As the counter service is invoked, the progress bar is invoked.

The example is located in `D:\labs\08-FXConcurrency\examples\CounterBar`.

Quiz

Which of the following is an interface that receives notifications of changes to an `ObservableMap`?

a. `ObservableMap`
b. `javafx.collections`
c. `MapChangeListener`
d. `ObservableList`

Answer: c

Summary

In this lesson, you should have learned how to:

- Describe and implement JavaFX concurrency
- Describe binding and properties in JavaFX, including simple binding and bi-directional binding

ORACLE

Practice 8: Overview

- Practice 8-1: Displaying Service State Information
- Practice 8-2: Adding a Second Service to Your Application

9

Java Persistence API (JPA)

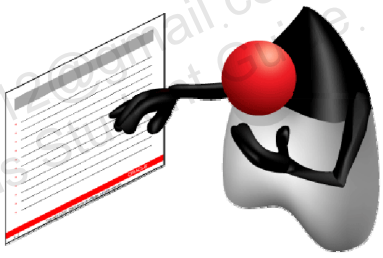
Objectives

After completing this lesson, you should be able to:

- Describe the Java Persistence API (JPA)
- Define Object-Relational Mapping (ORM) and how JPA provides a framework to support ORM
- Use JPA to create, read, update, and delete database entities
- Create and use queries in JPA

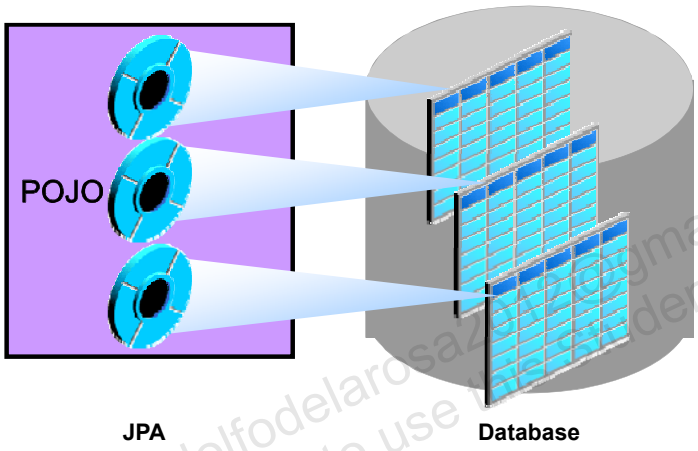
Topics

- What is JPA?
- Components of JPA architecture
- Transactions
- Entity operations and queries



Java Persistence API: Overview

The Java Persistence API (JPA) is a lightweight framework that leverages Plain Old Java Objects (POJOs) for persisting Java objects that represent relational data (typically in a database).



The Java Persistence API (1.0) began as a part of the Enterprise JavaBeans 3.0 specification (JSR-220) to standardize a model from Object-Relational Mapping.

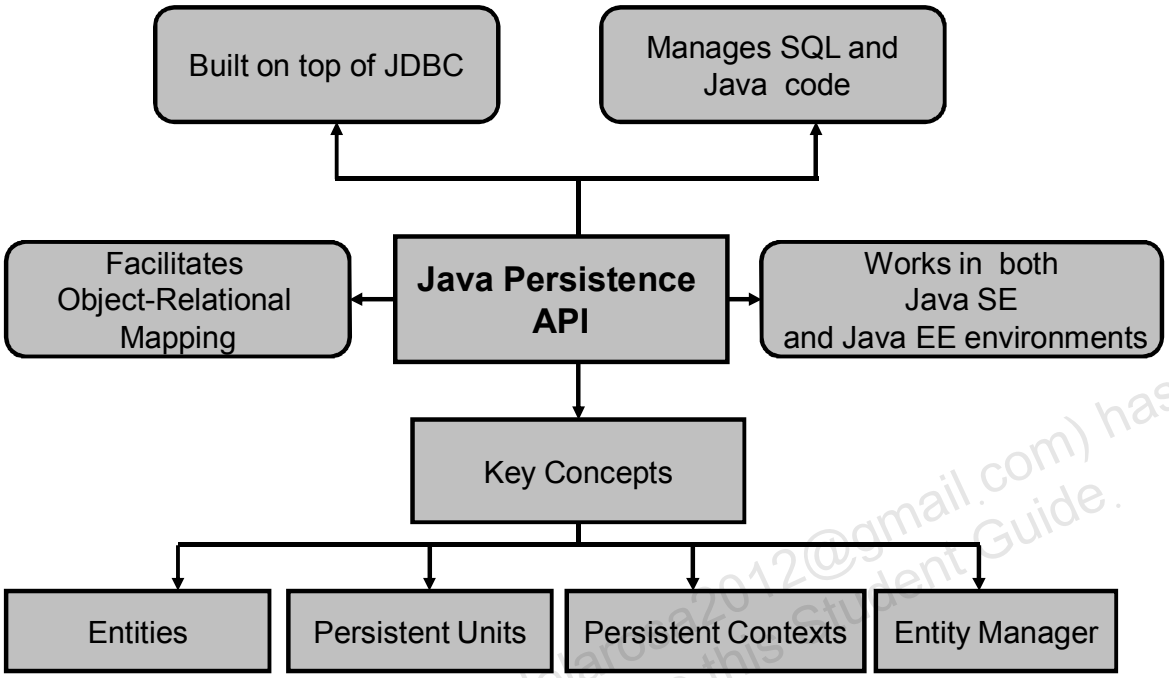
JPA 2.0 (JSR-317) set out to improve on the original JPA specification and was defined in its own JSR.

POJO: This term was added in the EJB 3.0 specification. It is a Java object that does not extend specific classes or implement specific interfaces as required by the EJB framework. Therefore, all normal Java objects are POJOs. POJO is another way of describing normal non-enterprise Java objects.

Benefits of Using JPA

- You do not need to create complex data access objects (DAO).
- The API helps you manage transactions.
- You write standards-based code that interacts with any relational database, freeing you from vendor-specific code.
- You can avoid SQL and instead use a query language that uses your class names and properties.
- You can use and manage POJOs.
- You can use JPA for desktop application persistence.

Features of JPA



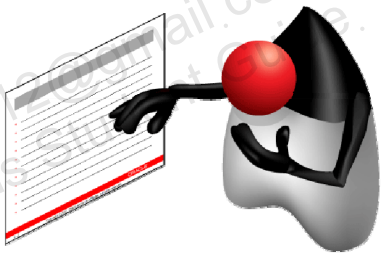
JDBC was the first mechanism that Java developers used for persistent storage of data. However, working with JDBC requires that you understand how to map a Java object to a database table and maintain the set of SQL queries used to transform relational data into Java objects. JPA provides a framework for this Object-Relational Mapping while preserving the ability to manipulate databases directly.

The key JPA concepts in this lesson are the starting points for writing JPA applications:

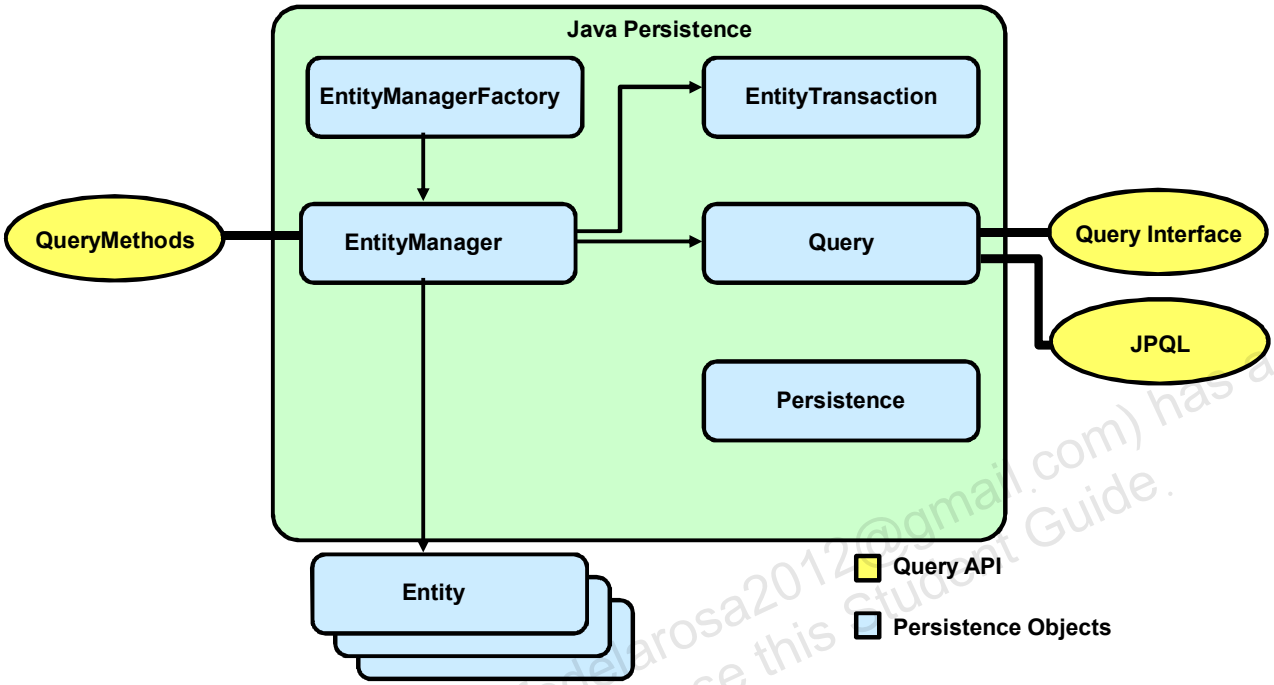
- An entity can be used to represent a relational table by a Java object. (This is a one-to-one mapping.)
- A persistence unit defines the set of all classes that are related or grouped by the application, and which must be co-located in their mapping to a single database.
- A persistence context is a set of entity instances in which there is a unique entity instance for any persistent entity identity.
- An entity manager does the work of creating, reading, and writing entities.

Topics

- What is JPA?
- Components of JPA architecture
- Transactions
- Entity operations and queries



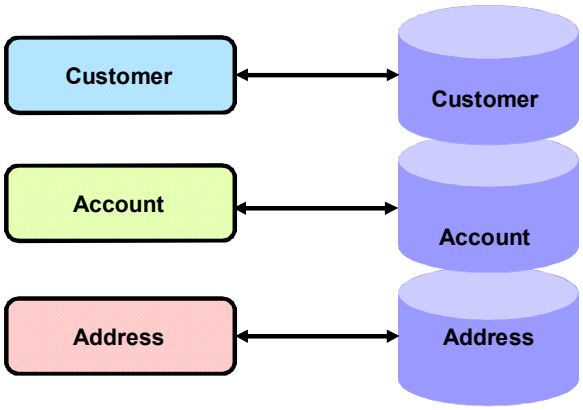
JPA Components



The figure in the slide shows the components of the JPA architecture.

- **Entity:** The persistence object that represents one record in the database table. It is a (Plain Old Java Object) POJO class with annotations.
- **EntityManager:** The `EntityManager` interface provides the API for interacting with the entity. There are methods to persist the entity in the database, update the state of the entity, or remove the entity instance.
- **EntityManagerFactory:** The EntityManagerFactory is used to create an instance of EntityManager. When an application no longer uses the EntityManagerFactory, it is necessary to close the instance of EntityManagerFactory. After the EntityManagerFactory is closed, all its EntityManagers are also closed.

Object-Relational Mapping (ORM)



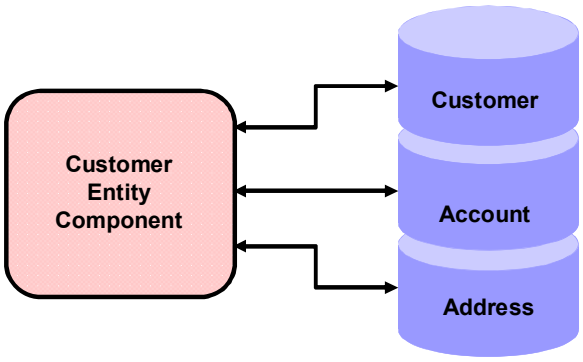
The storage mechanism that is most often used by applications is a relational database. A relational database is typically configured to store data in tables that have a relationship to one another. However, an object-oriented application design may not have the same organization in the same structure. A Java domain object can encompass partial data from a single database table, or it can include data from multiple tables depending on the normalization of the relational database.

Persistence mechanisms are the code that enable programmers to persist data (store) in a relational database. JDBC and JPA are two examples of persistence mechanisms.

Writing code to translate a relational schema to an object-oriented domain schema (or vice versa) can be time-consuming and error-prone. Object-Relational Mapping (ORM) software frameworks attempt to manage the mapping so that object-oriented programmers have to do very little coding. EclipseLink and Hibernate are examples of ORM software.

In the diagram in the slide, we see the most basic ORM mapping. This simple mapping of Java objects to database tables is a one-to-one mapping.

Object-Relational Mapping (ORM)



The diagram in the slide shows a more complex ORM mapping, where a Java domain object is composed of data from multiple tables where there is a relationship across the tables. This is a one-to-many mapping.

This lesson focuses on the basics of using JPA in a simple one-to-one mapping. For more complex relationships, and a more in-depth coverage of JPA, enroll in the Oracle course titled *Building Database-Driven Applications with JPA*.

Note: EclipseLink is the continuation of Oracle's open-source version of TopLink, which Oracle donated to the Eclipse Foundation. EclipseLink is the reference implementation for the Java Persistence API 2.0 specification. For more information about EclipseLink, see http://www.eclipse.org/eclipselink/.

JPA Entities

A JPA entity describes a concept or concrete thing that can be represented by attributes (data) that have possible relationships to other entities.

- Entities are meant to be persisted—to be stored in a relational database.
- An entity:
  - Is a POJO created by using the `new` keyword
  - Supports inheritance and polymorphism
  - Is serializable; can be used as detached objects
- Entities can be queried.
- Entities are managed at runtime through the Entity Manager API.

The JPA entity is the Java object that represents something to be persisted—a customer, an employee, a book, and so on. Basically, anything that is typically represented in a relational database can be represented by a POJO that defines the elements of the entity, and includes methods to set and get the values of each element.

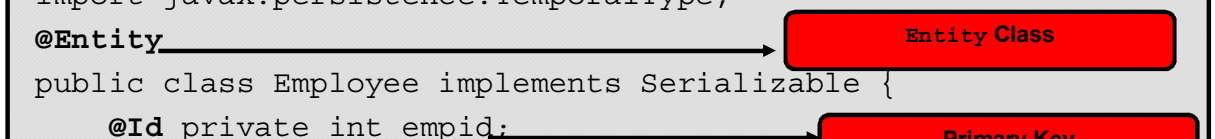
The JPA characteristics of entities include:

- **Persistence:** Entities that are created can be stored in a database (persisted), but they are treated as objects.
- **Identity:** Each entity has a unique object identity that is usually associated with a key in the persistent store (database). Typically, the key is the database primary key.
- **Transactions:** JPA requires a transactional context for operations that commit changes to the database.
- **Granularity:** JPA entities can be as fine-grained or coarse-grained as a developer wants, but are intended to represent a single row in a table.

Entities are managed by the `EntityManager` API, which is covered later in this lesson.

Creating an Entity

```
//...
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
@Entity
public class Employee implements Serializable {
    @Id private int empid;
    private String firstName;
    private String lastName;
    @Temporal(TemporalType.DATE) private Date birthDate;
    private double salary;
    public Employee() {}
    public Employee(int id, String firstName, String lastName,
                    Date birthDate, double salary) { ... }
    // ...
}
```



An entity can be created from a POJO by using annotations. In the example in the slide, you can see the `@Entity` annotation to declare the Employee class as an entity class to manage.

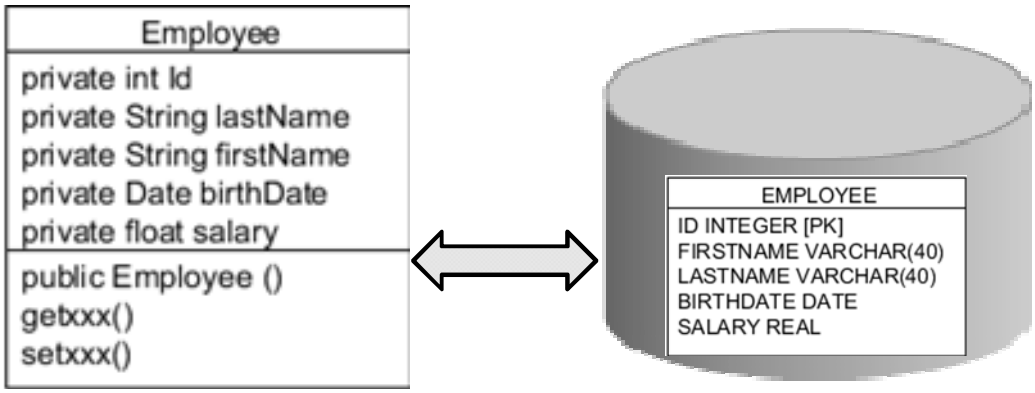
The `@Id` annotation declares the primary key. (Primary keys are covered in detail later in this lesson.)

The `@Temporal` annotation is required for any persisted field that is of `java.util.Date` or `java.util.Calendar` type. Because databases store dates in different ways (some as TimeStamp types and some as Date types), the `@Temporal` annotation indicates that you want Java to manage the conversion to and from the Java Date or Calendar type to the appropriate type in the database.

The requirements for an entity class include the following:

- The entity class must be annotated with the `@Entity` annotation.
- The entity class must have a no-arg constructor. The entity class may have other constructors as well.
- The no-arg constructor must be `public` or `protected`.
- The entity class must be a top-level class. An `enum` or `interface` must not be designated as an entity.
- The entity class must not be `final`. No methods or persistent instance variables of the entity class may be `final`.
- If an entity instance is to be passed by value as a detached object (for example, through a remote interface), the entity class must implement the `java.io.Serializable` interface.

Entity Mapping



ID (PK)	FIRSTNAME	LASTNAME	BIRTHDATE	SALARY
110	Troy	Hammer	1965-03-31	102109.15
123	Michael	Walton	1986-08-25	93400.2
201	Thomas	Fitzpatrick	1961-09-22	75123.45
101	Abhijit	Gopali	1956-06-01	89345.0
120	Rajiv	Sudahari	1969-12-22	68400.22

ORACLE

In a simple entity mapping, the default name of the table used to store an entity is the name of the entity class.

Each row in the table is represented by an instance of an object that maps each of the elements of the table to a field in the class by name. So the table element FIRSTNAME is mapped to the `firstName` field in the Employee class.

You can modify the default table and field names by using annotations. You will learn this later in the lesson.

Primary Keys

Every entity must have a primary key. The primary key is used to distinguish one entity instance from another.

- The primary key provides the identity of an entity and is used by the entity manager.
- The `@Id` annotation is used to identify the primary key.
- The most common primary key is an integer field or a string field, but it can be a custom class that corresponds to several database columns (compound key).

```
@Id private int id;
```

- Primary keys can be generated automatically.

```
@Id @GeneratedValue (strategy=GenerationType.AUTO)
@Column(name="ID")
private int id;
```

ORACLE

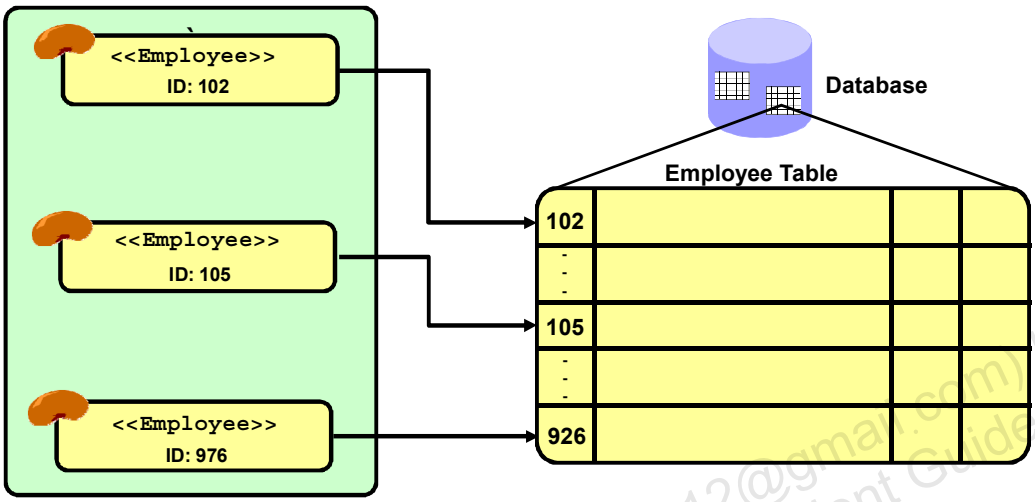
As shown in the previous slide, the primary key uniquely identifies each row in the Employee table.

Note that when you generate keys automatically, the primary key must be a simple key type, and the persistence provider is responsible for determining how to generate the key:

- **AUTO:** The persistence provider chooses the best strategy for key generation.
- **TABLE:** The persistence provider uses an underlying database table for key generation.
- **SEQUENCE, IDENTITY:** The persistence provider uses a sequence or identity column.

The entity's primary key is the `id` property and is correctly marked with the `@Id` annotation. Primary keys can be auto-generated values. The reference implementation will generate a key automatically if you add the `GeneratedValue` annotation to the primary key.

Entity Component Primary Key Association



The figure in the slide shows the relationship between entity instances and table rows.

Overriding Mapping

- `@Table`: Is used to override the default mapping between a class and table

```
@Entity
@Table(name="EMPLOYEE") //EMPLOYEE is the database table name
public class Emp {
    //...
}
```

- `@Column`: Is used to override column name mapping

```
public class Emp {
    @Column(name="FIRSTNAME")
    private String fName;
    //...
}
```


JPA Annotations

Without overriding a table or column name, JPA uses introspection to determine the table and column names.

In the example shown in the slide, the name of the column in the EMPLOYEE table is FIRSTNAME and is mapped to a field named `fName`. So, while you write code that gets and sets or somehow changes the value of `fName`, JPA always maps this field data to FIRSTNAME in the database.

Transient Fields

`@Transient`: Like the `transient` keyword, this annotation indicates that a field is not persistent.

```
public class Cart {
   @Id int customerId;
   //...
   @Transient float cartTotal;
//...
}
```


`@Transient` Versus `transient`

The JPA annotation `@Transient` provides an indication to the persistence provider not to persist a field. However, `@Transient` does not prevent a field from being serialized.

The code example in the slide shows a Cart class with a field cartTotal that is not required to be persisted.

Persistent Fields Versus Persistent Properties

Entity classes have their state synchronized with a database. The state of an entity is obtained either from its fields or from its accessor methods (properties).

- Field-based or property-based access is determined by the placement of annotations.
- An annotation placed on one field (variable) means that the persistence state of the entity class is field-based.

```
@Entity
public class Employee implements Serializable {
    @Id private int id;
    @Temporal(TemporalType.DATE) private Date birthDate;
    //...
}
```


Field-based and Property-based Persistent State

JPA provides two types of access to the data of a persistent class:

- **Field access:** Maps the instance variables (fields) to columns in the database table
- **Property access:** Uses the getters to determine the property names that will be mapped to the table

The access type depends on where you put the `@Id` annotation. For field-based access, the annotation is placed in the `id` field. For property-based access, the annotation is placed in the `getId()` method. You cannot use both access types in the same class.

Persistent Field Access

- Persistent fields cannot be `public`.
- Persistent fields should not be read by clients directly.
- Unless annotated with the `@Transient` annotation or `transient` keyword, all fields are persisted. (The use of the `@Column` annotation defines only the column name to use.)

Note: Accessor methods can be present in an entity class that uses field-based access and may be useful for other types of operations, but they are ignored by the persistence provider.

Persistent Fields Versus Persistent Properties

An annotation placed on a getter method means that the persistence state is property-based.

```
@Entity
public class Employee implements Serializable {
    private int id;
    private String firstName;

    @Id @Column(name="ID") public int getId() {
        return id;
    }
    @Temporal(TemporalType.DATE) public Date getBirthDate() {
        return birthDate;
    }
    //...
}
```


Persistent Properties

When you use persistent properties, the persistence provider will retrieve an object's state by calling the accessor methods of the entity class.

- Methods must be declared `public` or `protected`.
- Methods must follow the JavaBeans naming convention.
- `@Column` may also be used to define the column name in the database.
- Persistence annotations can be placed only on getter methods.

Developers prefer filed access over property access for several reasons:

- With field access, your annotated persistent fields are all neatly organized near the top of your class.
- With field access, the state is well encapsulated.
- With property access, you must define getter methods for every field just for use by JPA even if your code will never call them.

However, as the application design grows, property access might be preferable for performance or security reasons.

Persistence Data Types

Persistence fields or properties can be of the following data types:

- Java primitive types
- Java wrappers, such as `java.lang.Integer`
- `java.lang.String`
- `byte[]` and `Byte[]`
- `char[]` and `Character[]`
- Any serializable types including but not limited to:
  - `java.util.Date`
  - `java.sql.Date`
  - `java.sql.TimeStamp`
  - User-defined types

Note

Complex data types that do not have simple mapping (such as a `java.lang.String` to a `VARCHAR` or `CHAR` column) might require additional annotations. These annotations are described in the course titled *Building Database-Driven Applications with Java Persistence API.*

Entity Manager

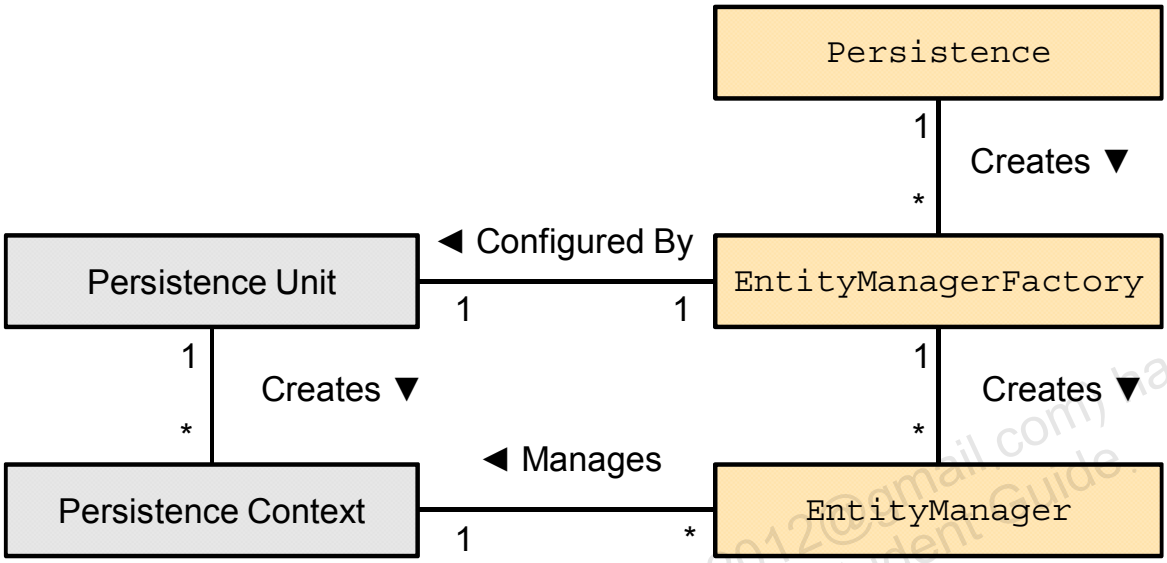
The `EntityManager` interface performs the work of persisting entities.

- Entities that an entity manager instance holds references to are *managed* by the entity manager.
- A set of entities in an entity manager at any given time is called its *persistence context*.
- Entities in the persistence context are unique. For example, there is only one Java instance with an ID of 110.
- A *persistence provider* creates the implementation details to read and write to a given database.
- An instance of an entity manager is obtained from a factory of the type `EntityManagerFactory`.

Entities do not add or remove themselves from the database when they are created or deleted. It is the logic of the application that must manipulate entities to manage their persistent life cycle. JPA provides the EntityManager interface for this purpose enable applications to manage and search for entities in the relational database.

A persistence unit is a named configuration of entity classes. A persistence context is a managed set of entity instances. Every persistence context is associated with a persistence unit, restricting the classes of the managed instances to the set defined by the persistence unit. Saying that an entity instance is managed means that it is contained in a persistence context and it can be acted on by an entity manager. This is why we say that an entity manager manages a persistence context.

Persistence Unit



JPA is a specification for the API and life-cycle behavior. It is not ORM software by itself. To use the JPA to perform ORM operations, you must first obtain an instance of a JPA provider implementation.

A persistence unit provides the concrete classes and object-relational mapping information. There is a one-to-one relationship between the persistence unit and its concrete `EntityManagerFactory`. A persistence unit:

- Maps to a single database
- Defines the scope for queries and relationships
- Defines the configuration information for the persistence provider
- Is specified by using a `persitence.xml` file

Although there can be multiple `EntityManager` instances, they will all point to a single persistence context.

The `Persistence` class is a bootstrap class for obtaining an `EntityManagerFactory` instance in Java SE environments. In Java EE environments, an `EntityManager` can be obtained using dependency injection.

Persistence Unit: Definition

The `persistence.xml` file is used to describe a persistence unit for an application.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ...>
  <persistence-unit name="EmployeePU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>examples.model.Employee</class>
    <validation-mode>NONE</validation-mode>
    <properties>
      <property name="javax.persistence.jdbc.url"
value="jdbc:derby://localhost:1527/EmployeeDB"/>
      <property name="javax.persistence.jdbc.password" value="tiger"/>
      <property name="javax.persistence.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
      <property name="javax.persistence.jdbc.user" value="public"/>
    </properties>
  </persistence-unit>
</persistence>
```


A persistence context is a grouping of unique entity instances that are managed by the persistence provider at runtime. A similar term is *persistence unit*, which is the set of all entity classes that an application might use. A persistence unit defines a group of entities mapped to a single database.

Obtaining an Entity Manager

An entity manager instance is obtained from an `EntityManagerFactory`.

In Java SE environments, this instance is obtained through the `Persistence` bootstrap class:

```
    EntityMangerFactory emf =
    Persistence.createEntityManagerFactory
("EmployeePU");
```

- The string passed to the method identifies the name of the persistence unit configuration.
- From the factory, you can obtain an entity manager instance supported by the provider:

```
EntityManager em = emf.createEntityManager();
```


EntityManager Instances

The persistence unit configuration is defined in the `persistence.xml` file. You will see this file later in this lesson.

Using JPA in a Java SE Application

1. Create the database in JavaDB or any database server.
2. Add the required libraries to your Java application.
   - EclipseLink (JPA 2.0)
   - `DerbyClient.jar`
3. Define Entities in your application.
4. Create `persistence.xml` and configure it.
   a. Define the persistent unit and transaction type,
   b. Provide the names of the Entity classes.
   c. Define the JDBC connection properties.
5. Create instances of `EntityManagerFactory` and `EntityManager`.
6. Write code to perform persistent entity operations using the entity manager instance.

ORACLE

You have now learned about all the JPA components. This slide lists the steps to start developing a JPA application.

The Reference Implementation for the Java Persistence API is called EclipseLink and is an open-source and freely available Eclipse project derived from the Oracle TopLink product code base. EclipseLink can be downloaded from java.net and is part of the NetBeans IDE.

Steps 3 through 5 are easily generated if we use NetBeans.

Subsequent slides provide details about step 6.

Quiz

To create an entity from a POJO, you must:

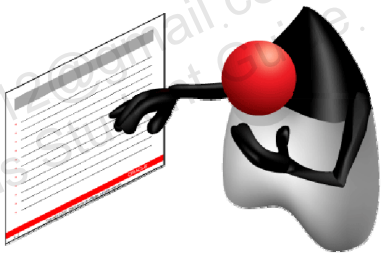
a. Implement the `javax.persistence.Entity` interface
b. Create a `persistence.xml` file
c. Annotate the class with `@Entity`
d. Declare the class as `final`

Answer: c

To define an POJO as an Entity class, you must annotate the class with `@Entity` (`javax.persistence.Entity`).

Topics

- What is JPA?
- Components of JPA architecture
- Transactions
- Entity operations and queries



What Is a Transaction?

- A transaction is a mechanism to handle groups of operations as though they were one operation.
- Either all operations in a transaction occur or none occur at all.
- The operations involved in a transaction might rely on multiple databases.

A typical example of using a transaction is as follows. Suppose that a client application needs to make a service request that might involve multiple read and write operations to a database. If any one invocation is unsuccessful, any state that is written (either in memory or, more typically, to a database) must be rolled back.

Consider an interbank fund transfer application in which money is transferred from one bank to another.

The transfer operation requires the server to make the following invocations:

1. Invoking the debit method on one account at the first bank
2. Invoking the credit method on another account at the second bank

If the credit invocation on the second bank fails, the banking application must roll back the previous debit invocation on the first bank.

ACID Properties of a Transaction

A transaction is formally defined by the set of properties that is known by the acronym **ACID**.

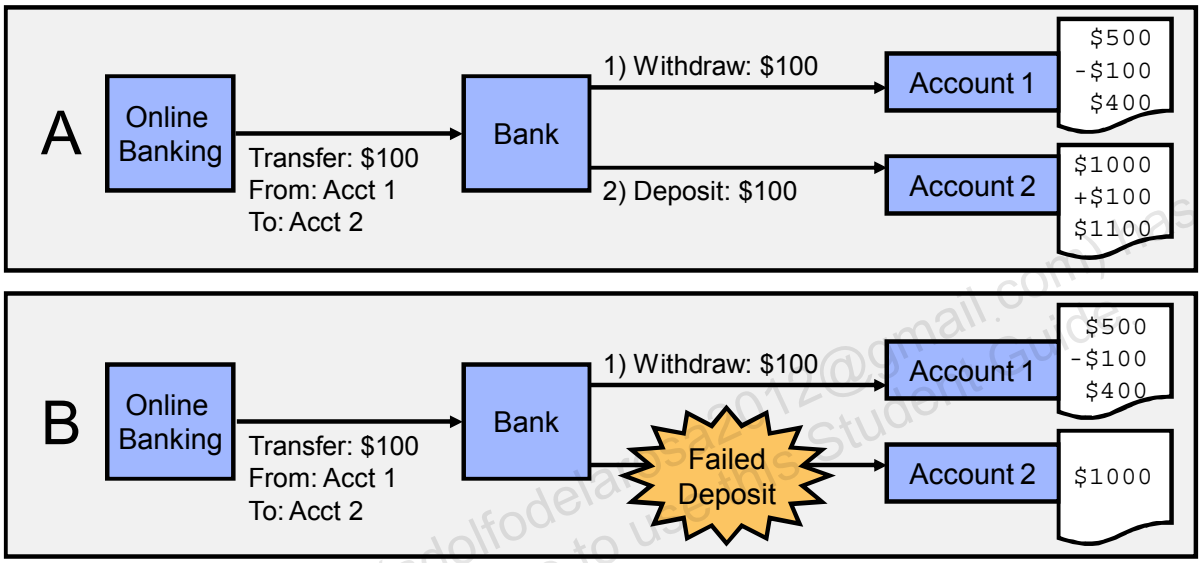
- **A**tomicity: A transaction is done or undone completely. In the event of a failure, all operations and procedures are undone, and all data rolls back to its previous state.
- **C**onsistency: A transaction transforms a system from one consistent state to another consistent state.
- **I**solation: Each transaction occurs independently of other transactions that occur at the same time.
- **D**urability: Completed transactions remain permanent, even during system failure.

Transactions should have the following ACID properties:

- **Atomicity:** All or nothing; all operations involved in the transaction are implemented or none are.
- **Consistency:** The database must be modified from one consistent state to another. If the system or database fails during the transaction, the original state is restored (rolled back).
- **Isolation:** An executing transaction is isolated from other executing transactions in terms of the database records it is accessing.
- **Durability:** After a transaction is committed, it can be restored to this state if a system or database failure occurs.

Transferring Without Transactions

A. Successful transfer
B. Unsuccessful transfer (Accounts remain in an inconsistent state.)



Consider an online banking application. Users can make online money transfers from one account to another account. In such a transfer scenario, one account should get debited and another account should get credited with the amount. The application must ensure that both operations happen successfully. If one operation fails, the other operation should also be nullified. This is easily managed in a transaction.

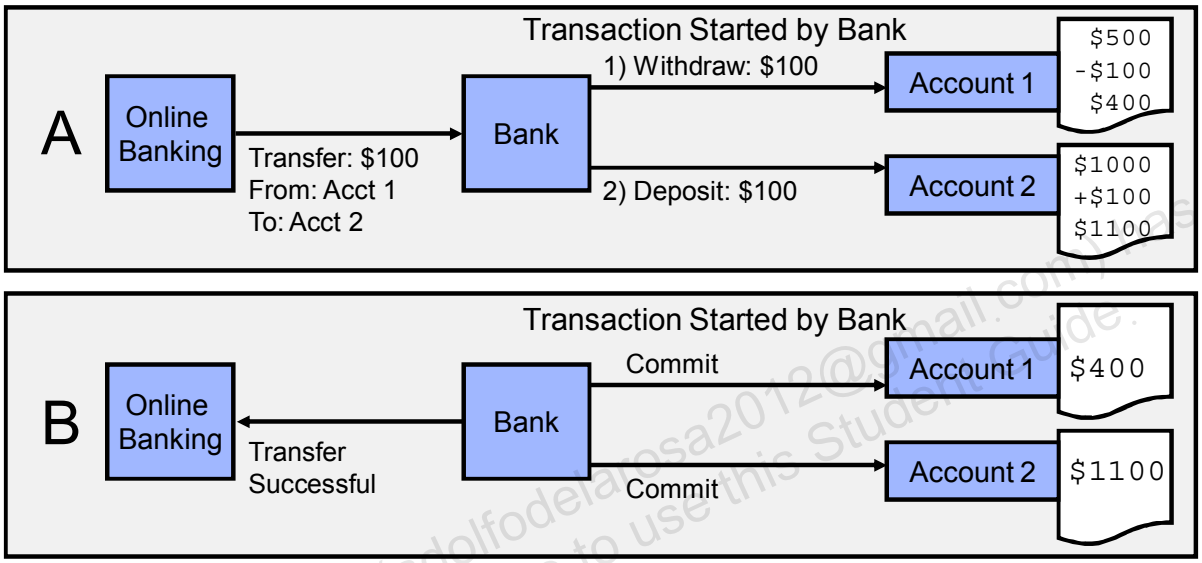
Transactions are appropriate in this online banking scenario.

A client application must converse with an object that is managed, and it must make multiple invocations on a specific object instance. The conversation can be characterized by one or more of the following steps:

1. Data is cached in memory or written to a database during or after each successive invocation.
2. Data is written to a database at the end of the conversation.
3. The client application requires that the object maintain an in-memory context between each invocation; each successive invocation uses the data that is maintained in memory.
4. At the end of the conversation, the client application requires the capability to cancel all the database write operations that may have occurred during or at the end of the conversation.

Successful Transfer with Transactions

A. Changes in a transaction are buffered.
B. If a transfer is successful, changes are committed (made permanent).



If the transaction is successful, the buffered changes are committed (that is, made permanent).

Within the scope of one client invocation on an object, the object performs multiple changes to the data in a database. If one change fails, the object must roll back all the changes.

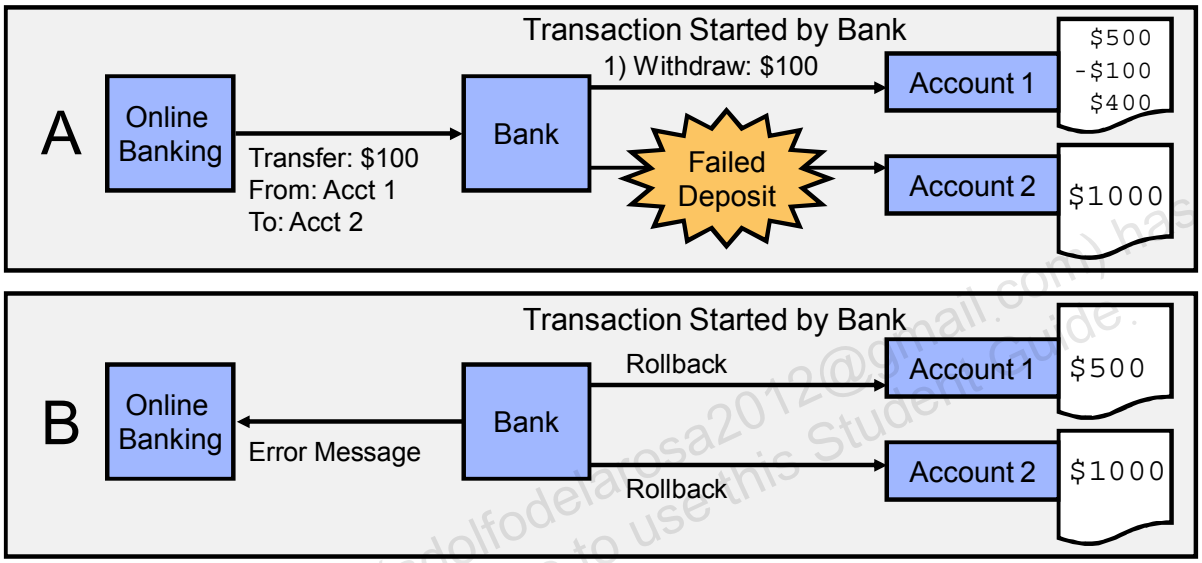
Consider a banking application. The client invokes the transfer operation on a teller object. The operation requires the teller object to make the following invocations on the bank database:

1. Invoking the debit method on one account
2. Invoking the credit method on another account

If the credit invocation on the bank database fails, the banking application must roll back the previous debit invocation.

Unsuccessful Transfer with Transactions

A. Changes in a transaction are buffered.
B. If a problem occurs, the transaction is rolled back to the previous consistent state.



If the transaction is unsuccessful, the buffered changes are thrown out and the database is rolled back to its previous consistent state.

Using JPA for Transactions

Entity operations are transactional. That is, they are required to be part of a transaction.

- Java SE applications can use resource-local transactions.
- The `EntityTransaction` interface supports resource local transactions.
- An `EntityTransaction` instance is obtained from `EntityManager`:

```
EntityTransaction et = em.getTransaction();
```

- Typical transaction methods include:
  - `begin()`: Starts a new resource transaction context
  - `commit()`: Completes the current transaction context and writes any unflushed changes to the database
  - `rollback()`: Rolls back the current transaction

ORACLE

There are two transactional models supported by JPA: resource-local transactions and JTA transactions.

- Resource local transactions are native transactions supported by the JDBC drivers referenced in the persistence unit.
- JTA transactions are part of a Java EE server.

There are three other methods in the `EntityTransaction` interface:

- **`setRollbackOnly`:** Sets the current transaction so that the only possible outcome is a rollback. This is especially appropriate for deleting a current long-running (atomic) transaction.
- **`getRollbackOnly`:** Returns a `boolean` indicating whether the current transaction is marked for rollback
- **`isActive`:** Returns a `boolean` indicating whether the resource transaction is in process

Quiz

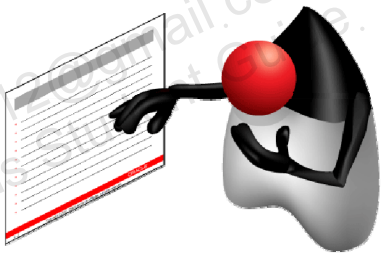
The transaction type in Java SE applications that use JPA is:

a. RESOURCE_LOCAL
b. JTA transactions
c. NULL
d. LOCAL

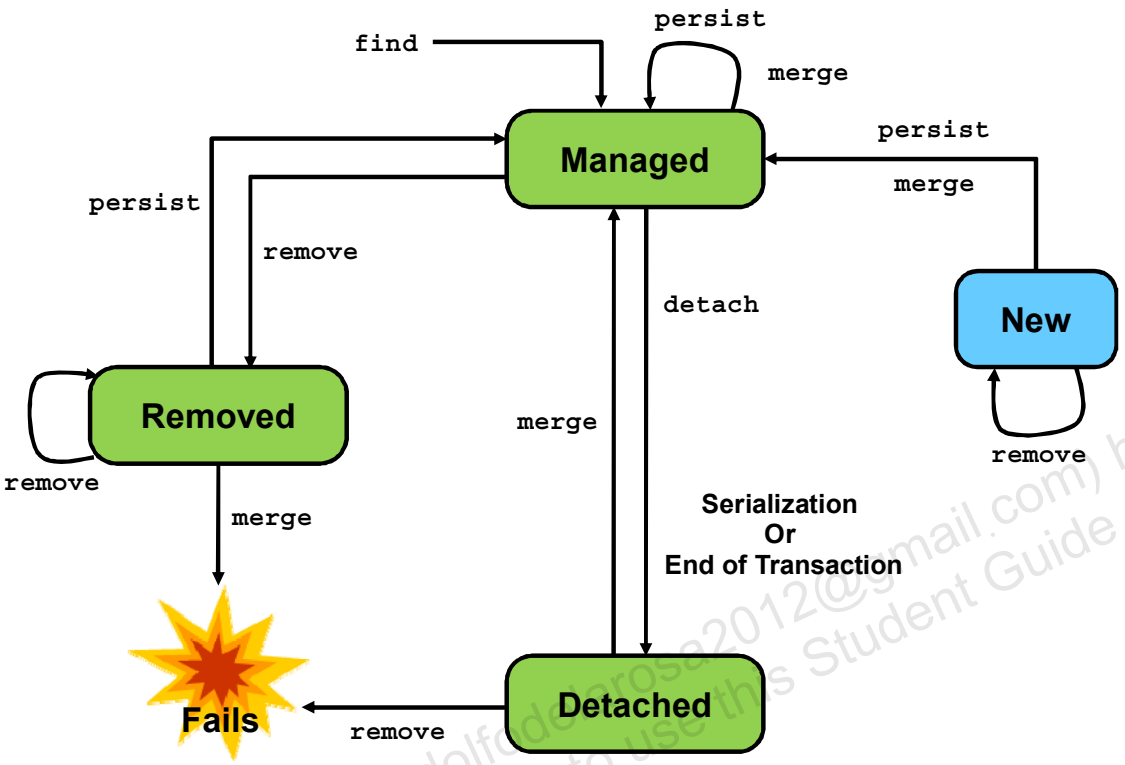
Answer: a

Topics

- What is JPA?
- Components of JPA architecture
- Transactions
- Entity operations and queries



Entity Instance Life Cycle



The life cycle of an entity is depicted in the slide. Entity instances are in one of four states: new, managed, detached, or removed.

- **New:** An entity object instance that has no persistent identity and is not yet associated with a persistence context (for example, when an Employee object is created using the new keyword)
- **Managed:** An instance with a persistent identity that is currently associated with a persistence context
- **Detached:** An instance with a persistent identity that is not (or is no longer) associated with a persistence context
- **Removed:** An instance with a persistent identity, associated with a persistence context, that will be removed from the database on transaction commit

You manage entity instances by invoking operations on the entity by means of an EntityManager instance.

The `EntityManager` interface defines the methods to manage entities, including:

- **Persisting an entity:** Taking a transient or one that has no persistent representation in the database and storing its state. After it is persisted, an entity is also in the managed state. Any further changes to this entity (before the transaction commit) are also persisted in the database.
- **Finding an entity:** Locating an entity in the persistent store. If an entity is successfully located in the persistent store, the entity is returned by the find method and managed by the entity manager.
- **Removing an entity:** Removing an entity from the persistent store (using a `DELETE` statement)
- **Updating an entity:** Locating an existing entity in the persistent store and making changes that are then persisted

These operations are the basic CRUD (Create, Read, Update, and Delete) operations performed on databases.

Persisting an Entity

Persisting an entity is the operation of writing an entity to the database. It is the equivalent of a SQL `INSERT` statement.

```
public static void main(String[] args) {
     EntityManagerFactory emf =
         Persistence.createEntityManagerFactory("EmployeePU");
     EntityManager em = emf.createEntityManager();

     Employee emp = new Employee (158, "John", "Doe", new
GregorianCalendar(1967, 11, 23).getTime(), 45000);
     em.getTransaction().begin();
     em.persist (emp);
     em.getTransaction().commit();

     em.close();
     emf.close();
}
```


In the code fragment in the slide, an `EntityManagerFactory` is obtained using the `Persistence` class and `EmployeePU` named configuration.

From the factory, an `EntityManager` is obtained.

Note that using the `EntityManager`, the resource-local transaction is also started and then committed immediately after persisting the new Employee entity.

Finding an Entity

To locate an entity in the database, the EntityManager will locate a row in the database based on the primary key:

```
Employee emp = em.find (Employee.class, 110);
```

- The `find` method uses the entity class passed in the first argument to determine what type to use for the primary key and for the return type. (So an extra cast is not required.)
- When the call returns, the employee returned is now a managed entity.
- If no employee with the ID 158 is found, a null is returned.

Using the `EntityManager.find` Method

This method is equivalent to a SQL `SELECT` statement.

Note that the second argument of the `find` method is `java.lang.Object`. Java automatically converts the integer primitive to an `Integer` class.

Updating an Entity

- After an entity is managed, you can make changes to the entity object without explicitly persisting it.
- This update is the equivalent of a SQL `UPDATE ... WHERE` statement.

```
emp = em.find(Employee.class, 158);
em.getTransaction().begin();
emp.setSalary(emp.getSalary() + 1000);
em.getTransaction().commit();
```


Perhaps the most interesting aspect of JPA is that after an entity is managed, changes made to the entity are automatically reflected in the persistent store (in a transaction context).

You may also want to be able to update an entity that is not managed. To do this, you can use the `merge` method. You will see this in later slides.

Deleting an Entity

The removal of an entity from the database is the equivalent of a SQL `DELETE` statement. To remove an entity, the entity must first be managed:

```
emp = em.find(Employee.class, 158);
em.getTransaction().begin();
em.remove(emp);
em.getTransaction().commit();
```

Note: If the value returned by find is `null`, the remove method throws an `java.lang.IllegalArgumentException`.

ORACLE

After the transaction is committed, the entity is removed (`DELETE`) from the persistent store. However, the Java object is still in memory.

Detach and Merge

A managed entity requires a transaction context. But there may be times when you want to “unmanage” an entity to make changes before returning the entity to managed status.

```
em.detach(emp);
emp.setLastName("Jones");
// ... in some other part of the code
em.getTransaction().begin();
emp = em.merge(emp);
emp.setSalary(emp.getSalary() + 20000);
em.getTransaction().commit();
```


ORACLE

In this example, the employee entity is detached from the entity manager. After it is detached, the entity can be operated on as if it were a Java object rather than a database record. A detached entity is not deleted from the persistent store.

After the changes are determined to be correct, the entity can then be merged back into the entity manager. As the example shows, the employee object returned from the merge method is managed, so additional changes can be made and the appropriate updates can be committed to the database.

Queries with JPQL

The JPA framework:

- Makes it possible to operate on individual entities
- Provides Java Persistence Query Language (JPQL) to query over entities, as in the following example:

```
TypedQuery<Employee> query =
    em.createQuery("SELECT e FROM Employee e",
Employee.class);
List<Employee> emps = query.getResultList();
for (Employee e : emps)
System.out.println("Employee: " + e);
```

- Queries can be dynamic or static.
  - Static queries are defined with an annotation:
    - `@NamedQuery, @NamedNativeQuery`

Note: JPQL is a rich language with many features. A full discussion of JPQL is beyond the scope of this lesson.

In the code fragment in the slide, `TypedQuery<Employee>` indicates that you want to create a query that returns entities that are Employee objects.

Here are a few features that JPQL supports:

- The results can be of single type or multiple type.
- Sorting and grouping
- Aggregate functions, expressions with conditions, and subqueries
- Syntax with joins
- Queries that allow bulk delete or updates
- Capture results in classes that are nonpersistent

`Query` and `TypedQuery` interfaces can be used to write queries.

A query may either be dynamically specified at runtime or configured in persistence unit metadata (annotation or XML) and referenced by name. Dynamic queries are nothing more than strings, and therefore may be defined as needed. Named queries are static and unchangeable. They are more efficient to execute because the persistence provider can translate the JPQL string to SQL once when the application starts as opposed to every time the query is executed.

Example: @NamedQuery

- Defining a named query:

```
@NamedQuery(name="Employee.findByName",
query="SELECT e FROM Employee e WHERE
e.name = :name")
```

- Executing a named query:

```
public Employee findEmployeeByName(String name) { return
    em.createNamedQuery("Employee.findByName", Employee.class)
    .setParameter("name", name) .getSingleResult();
```


A named query is defined using the `@NamedQuery` annotation, which may be placed on the class definition for any entity.

The annotation defines the name of the query as well as the query text.

If more than one named query is to be defined on a class, they must be placed inside of a `@NamedQueries` annotation, which accepts an array of one or more `@NamedQuery` annotations.

In the code snippet, `em` is an `EntityManager` instance. The following code indicates how it was created from the `EntityManagerFactory` instance:

```
EntityMangerFactory emf =  Persistence.createEntityManagerFactory
("EmployeePU");
EntityManager em = emf.createEntityManager();
```

Quiz

If an entity does not exist in the persistent store, using the `remove` method on the entity will throw an exception.

a. True
b. False

Answer: a

To remove an entity, it must first be managed using the `find` method. If the `find` method returns a null (the entity was not found with the primary key passed to the `find` method), the `remove` method throws a `java.lang.IllegalArgumentException`.

Quiz

Suppose that you are creating a new Employee entity. Which of the following steps is a best practice for persisting this entity?

a. Use the `update` method to add the record.
b. Use `find` to locate the record, and then `update` it.
c. Use `find` to see if the entity exists and, if not, `persist` it.
d. Use `find` to see if the entity exists, and then `merge` to update it.

ORACLE

Answer: c

It is a good practice to use `find` to determine if the entity exists in the persistent store. Another option would be to attempt to persist the Employee entity and then catch `EntityExistsException` to handle situations where the entity that is to be added already exists.

Summary

In this lesson, you should have learned how to:

- Describe the Java Persistence API (JPA)
- Define Object-Relational Mapping (ORM) and how JPA provides a framework to support ORM
- Use JPA to create, read, update, and delete database entities
- Create and use queries in JPA

Practice 9: Overview

- Practice 9-1: Creating Entity Classes by Using JPA
- Practice 9-2: Implementing CRUD Operations by Using JPA