ZOLAXTECH

Java for PC App Development

From Zero to Java Mastery to Shipped Desktop Software

A Beginner-to-Advanced Roadmap for Java, Swing, JavaFX, Databases & Packaging

---

zolaxtech.com.et

Free Learning Resource

Table of Contents

Welcome

This guide takes you from complete beginner to a confident Java desktop developer, capable of designing, building, and shipping real Windows, macOS, and Linux applications. As with any serious Java skill, everything rests on a solid grasp of the Java language itself so this guide opens with a complete, practical foundation in Java syntax, object-oriented programming, collections, and exception handling before writing a single window.

Once your Java foundation is solid, the guide covers two ways to build desktop user interfaces: the classic Swing toolkit and the modern JavaFX framework, followed by advanced professional topics reading and writing files, talking to databases with JDBC, keeping your UI responsive with multithreading, and finally packaging your finished app into an installer user can double-click to run.

> **How to use this guide?**
> Read it in order the first time through each chapter builds on the last. Type out every code example yourself in an IDE rather than copy-pasting; that repetition is what turns syntax into instinct. Each chapter closes with a short practice exercise don't skip these.

What you will be able to build by the end

- Classic desktop applications using Java Swing forms, dialogs, menus, and tables
- Modern, styled desktop applications using JavaFX and CSS
- Apps that read and write files, and remember data between runs
- Apps connected to a real database using JDBC and SQL
- Responsive apps that run long tasks in the background without freezing
- A finished app packaged as a native installer (.exe / .dmg / .deb) ready to distribute

Prerequisites

None. This guide assumes no prior programming experience. If you have coded before, feel free to skim Part 1 and move faster through the Java fundamentals.

PART ONE

Java Foundations

Every great desktop application starts with solid Java fundamentals. This part is a self-contained crash course in the language itself work through it carefully, even if you have dabbled in programming before.

CHAPTER 1

Getting Started with Java

Setting up your environment and writing your first program

1.1 Installing the JDK

Java programs run on the Java Virtual Machine (JVM), and you write them using the Java Development Kit (JDK). Download the latest JDK (17 or newer) from Oracle or use the free OpenJDK distribution (Adoptium/Temurin is a good choice). After installing, confirm it from a terminal:

```
java -version
javac -version
```

If both commands print version numbers, you are ready to compile and run Java code. For desktop development, install an IDE such as IntelliJ IDEA (Community Edition is free) or Eclipse both have first-class support for building and packaging desktop apps.

1.2 Your First Program

Every Java application starts with a class containing a main method this is the entry point the JVM looks for when it runs your program.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, ZolaxTech!");
    }
}
```

Save this as HelloWorld.java, then compile and run it:

```
javac HelloWorld.java
java HelloWorld
```

Breaking it down

- public class HelloWorld declares a class; the filename must match the class name.
- public static void main(String[] args) the method the JVM calls first.
- System.out.println(...) prints text to the console, followed by a new line.

✎ Try it yourself

1. Modify the program to print your own name instead of “ZolaxTech”.

2. Add a second println statement that prints today’s date as plain text.

CHAPTER 2

Variables, Data Types & Operators

Storing and manipulating data in Java

2.1 Declaring Variables

A variable is a named container for a value. Java is statically typed, meaning you must declare the type of data a variable will hold before you use it.

```
int age = 25;
double price = 19.99;
char grade = 'A';
boolean isActive = true;
String name = "Selam";
```

2.2 Primitive Data Types

Type	Size	Example	Used for
int	4 bytes	int count = 10;	Whole numbers
double	8 bytes	double total = 3.75;	Decimal numbers
boolean	1 bit	boolean isDone = false;	True/false values
char	2 bytes	char letter = 'z';	Single characters
long	8 bytes	long fileSize = 9000000000L;	Very large whole numbers

2.3 Operators

Java supports the standard set of arithmetic, comparison, and logical operators:

```
int sum = 5 + 3;        // arithmetic: + - * / %
boolean isEqual = (5 == 5);   // comparison: == != > < >= <=
boolean result = (true && false); // logical: && || !
```

Common beginner mistake

Use == to compare primitive values, but never use == to compare the contents of two String objects use .equals() instead. This trips up almost every new Java developer at least once.

✎ Try it yourself

1. Declare variables for a product name, price, and quantity, then calculate and print the total cost.
2. Write an expression that checks whether a number is even using the % operator.

CHAPTER 3

Control Flow

Making decisions and repeating actions

3.1 Conditional Statements

```
int score = 82;

if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else {
    System.out.println("Grade: C or below");
}
```

3.2 Switch Statements

```
int day = 3;
switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
    default: System.out.println("Another day");
}
```

3.3 Loops

Loops let you repeat a block of code. Java has three main loop types:

```
// for loop   known number of repetitions
for (int i = 0; i < 5; i++) {
    System.out.println("Count: " + i);
}

// while loop   repeats while a condition is true
int n = 0;
while (n < 3) {
    System.out.println("n is " + n);
    n++;
}
```

```
// do-while   runs the body at least once
int x = 0;
do {
   System.out.println("x = " + x);
   x++;
} while (x < 2);
```

✎ Try it yourself

1. Write a loop that prints all even numbers between 1 and 20.
2. Write a program that uses if/else to categorize a temperature as Cold, Mild, or Hot.

CHAPTER 4

Object-Oriented Programming

Classes, objects, and the four pillars of OOP

Object-oriented programming (OOP) is the backbone of Java and of desktop development, where every window, button, and data record you work with is an object built from a class. Master this chapter and Swing/JavaFX code will make far more sense.

4.1 Classes and Objects

A class is a blueprint; an object is an instance created from that blueprint.

```
public class Product {
    // Fields (state)
    String name;
    double price;

    // Constructor
    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // Method (behavior)
    public void displayInfo() {
        System.out.println(name + ": $" + price);
    }
}

// Creating and using an object
Product item = new Product("Keyboard", 45.00);
item.displayInfo();
```

4.2 The Four Pillars of OOP

Encapsulation

Bundling data and the methods that operate on it inside a class, and restricting direct access to that data using private fields with public getter/setter methods.

```
public class Account {
    private double balance;
```

```
    public double getBalance() { return balance; }
    public void deposit(double amount) {
        if (amount > 0) balance += amount;
    }
}
```

Inheritance

A class can inherit fields and methods from another class using extends, letting you reuse and specialize behavior including specialized window types in Swing and JavaFX, which you'll extend directly.

```
public class Vehicle {
    public void start() { System.out.println("Starting..."); }
}

public class Car extends Vehicle {
    public void openTrunk() { System.out.println("Trunk open"); }
}

Car myCar = new Car();
myCar.start();      // inherited from Vehicle
myCar.openTrunk();  // defined in Car
```

Polymorphism

The same method name behaves differently depending on the object calling it, typically through method overriding.

```
public class Shape {
    public double area() { return 0; }
}
public class Circle extends Shape {
    private double radius;
    public Circle(double radius) { this.radius = radius; }
    @Override
    public double area() { return Math.PI * radius * radius; }
}
```

Abstraction

Hiding implementation detail behind a simple interface, using abstract classes or interfaces so callers only need to know what a method does, not how.

```
public interface Payable {
   double calculatePayment();
}

public class Employee implements Payable {
   private double hours, rate;
   public Employee(double hours, double rate) {
      this.hours = hours; this.rate = rate;
   }
   @Override
   public double calculatePayment() { return hours * rate; }
}
```

Why this matters for desktop apps?

Every window you build extends a Swing JFrame or JavaFX Application class (inheritance). You'll implement listener interfaces like ActionListener (abstraction/polymorphism). And you'll encapsulate application data inside model classes constantly. OOP is the daily grammar of desktop Java code.

✎ Try it yourself

1. Create a Shape hierarchy with Circle, Rectangle, and Triangle, each overriding an area() method.
2. Create a BankAccount class with a private balance, and public deposit() and withdraw() methods that validate input.

CHAPTER 5

Collections & Data Structures

Storing groups of data efficiently

5.1 Arrays

```
int[] numbers = {10, 20, 30, 40};
String[] names = new String[3];
names[0] = "Abel";
System.out.println(numbers[1]); // 20
```

5.2 Array List

Unlike arrays, an ArrayList can grow and shrink dynamically the collection you will use constantly, including to back Swing JTable and JavaFX TableView data models later in this guide.

```
import java.util.ArrayList;
import java.util.List;

List<String> tasks = new ArrayList<>();
tasks.add("Design the main window");
tasks.add("Connect to the database");
tasks.remove("Design the main window");

for (String task : tasks) {
    System.out.println(task);
}
```

5.3 HashMap

Stores key-value pairs for fast lookups by key.

```
import java.util.HashMap;
import java.util.Map;

Map<String, Double> prices = new HashMap<>();
prices.put("Keyboard", 45.00);
prices.put("Mouse", 20.00);
System.out.println(prices.get("Keyboard")); // 45.0
```

✎ Try it yourself

1. Build an ArrayList of customer names and print them using a for-each loop.
2. Build a HashMap that maps product names to prices, then print the total value of all products.

CHAPTER 6

Exception Handling

Writing code that fails gracefully

Real desktop apps encounter unexpected situations a missing file, invalid user input, a lost database connection. Java's try/catch mechanism lets you handle these gracefully instead of crashing on your user.

```
try {
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[5]); // out of bounds
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    System.out.println("This always runs.");
}
```

You can also define and throw your own exceptions to enforce rules in your own code:

```
public void setAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("Age cannot be negative");
    }
    this.age = age;
}
```

Why this matters for desktop apps?

You'll use try/catch constantly reading files, parsing user input from text fields, and running SQL queries all throw checked exceptions in Java. Handling them well is the difference between an app that crashes and one that shows the user a friendly error dialog.

PART TWO

Desktop UIs with Swing

Swing is Java's original, mature GUI toolkit. It ships with every JDK, requires no extra dependencies, and remains widely used in real production software an excellent place to learn the concepts behind desktop UI programming.

CHAPTER 7

Setting Up for Desktop Development

Project structure and tools

7.1 Creating a Project

1. Open your IDE (IntelliJ IDEA or Eclipse) and create a new Java project.
2. Make sure the Project SDK points to your installed JDK (17 or newer).
3. Create a package such as com.zolaxtech.desktopapp to keep your classes organized.
4. Create your first class with a main method as the entry point.

7.2 Running from the Command Line

You do not need an IDE to build desktop apps this also works from a plain terminal:

```
javac -d out src/com/zolaxtech/desktopapp/Main.java
java -cp out com.zolaxtech.desktopapp.Main
```

7.3 Swing vs. JavaFX at a Glance

	Swing	JavaFX
Ships with JDK	Yes	No (separate SDK since Java 11)
Look and feel	Native-ish, dated by default	Modern, fully CSS-stylable
Layout defined in	Java code	FXML (XML) or Java code
Best for	Simple tools, legacy maintenance	New, polished applications

CHAPTER 8

Your First Swing Window

JFrame, JPanel, and basic components

8.1 A Minimal Window

```
import javax.swing.*;

public class MainWindow {
    public static void main(String[] args) {
        JFrame frame = new JFrame("ZolaxTech Desktop App");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);

        JLabel label = new JLabel("Welcome to your first Swing app!");
        frame.add(label);

        frame.setVisible(true);
    }
}
```

Breaking it down

- JFrame the top-level window that holds your entire application UI.
- setDefaultCloseOperation tells the app to exit fully when the window is closed.
- frame.add(...) places a component inside the window.
- setVisible(true) must be called last, after all components are added.

> **Best practice**
> Always create and update Swing UI on the Event Dispatch Thread using SwingUtilities.invokeLater(...). This avoids subtle threading bugs as your app grows.

```
SwingUtilities.invokeLater(() -> {
    JFrame frame = new JFrame("ZolaxTech Desktop App");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
});
```

> ✎ **Try it yourself**
> 1. Change the window title and size, and add a second JLabel below the first.

2. Look up JButton in the Swing documentation and add one to your window.

CHAPTER 9

Layout Managers & Components

Arranging your UI properly

Swing never lets you place components at fixed pixel coordinates by default — instead, a LayoutManager decides positioning, so your UI adapts cleanly when the window is resized.

Layout Manager	Behavior
BorderLayout	Five regions: North, South, East, West, Center
FlowLayout	Places components left-to-right, wrapping as needed
GridLayout	Arranges components in an even grid of rows and columns
BoxLayout	Stacks components in a single row or column
GridBagLayout	Flexible, precise grid-based layout for complex forms

9.1 A Simple Form

```
JPanel panel = new JPanel(new GridLayout(2, 2, 8, 8));
panel.add(new JLabel("Name:"));
panel.add(new JTextField());
panel.add(new JLabel("Email:"));
panel.add(new JTextField());

frame.setLayout(new BorderLayout());
frame.add(panel, BorderLayout.CENTER);
```

9.2 Common Components

- JLabel — displays static text or an icon
- JTextField / JPasswordField — single-line text input
- JTextArea — multi-line text input
- JButton — a clickable button
- JCheckBox / JRadioButton — toggle and choice controls
- JComboBox — a dropdown selection list

- JTable displays tabular data, backed by a TableModel

> ✎ **Try it yourself**
>
> 1. Build a registration form with fields for name, email, and password using GridLayout.
> 2. Add a JComboBox listing three cities and print the selected value when a button is clicked.

CHAPTER 10

Events, Menus & Dialogs

Making your app interactive

10.1 Handling Button Clicks

```
JButton saveButton = new JButton("Save");
saveButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(frame, "Saved successfully!");
    }
});
```

Modern Java lets you write the same listener far more concisely with a lambda expression:

```
saveButton.addActionListener(e -> JOptionPane.showMessageDialog(frame, "Saved successfully!"));
```

10.2 Menu Bars

```
JMenuBar menuBar = new JMenuBar();
JMenu fileMenu = new JMenu("File");
JMenuItem exitItem = new JMenuItem("Exit");
exitItem.addActionListener(e -> System.exit(0));
fileMenu.add(exitItem);
menuBar.add(fileMenu);
frame.setJMenuBar(menuBar);
```

10.3 Dialogs

```
int choice = JOptionPane.showConfirmDialog(frame, "Discard unsaved changes?",
        "Confirm", JOptionPane.YES_NO_OPTION);
if (choice == JOptionPane.YES_OPTION) {
    frame.dispose();
}
```

✎ Try it yourself

1. Add a File menu with New, Open, and Exit items.
2. Show a confirmation dialog when the user clicks a Delete button, and only proceed if they confirm.

PART THREE

Modern UIs with JavaFX

JavaFX is the modern successor to Swing built for richer graphics, smooth animation, and full CSS styling. Most new, polished Java desktop applications are built with JavaFX today.

CHAPTER 11

Introduction to JavaFX

Scenes, stages, and your first app

11.1 Adding JavaFX to Your Project

Since Java 11, JavaFX ships separately from the JDK. Download the JavaFX SDK from openjfx.io, or add it as a Maven/Gradle dependency:

```
<!-- Maven pom.xml -->
<dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>21</version>
</dependency>
```

11.2 Your First JavaFX App

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class MainApp extends Application {
    @Override
    public void start(Stage stage) {
        Label label = new Label("Welcome to JavaFX!");
        StackPane root = new StackPane(label);

        Scene scene = new Scene(root, 400, 300);
        stage.setTitle("ZolaxTech Desktop App");
        stage.setScene(scene);
        stage.show();
    }

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

Breaking it down

- Stage the top-level window, provided to start() by the JavaFX runtime.
- Scene the container for all visible content, with a defined width and height.
- Application the base class every JavaFX app extends; start() is your entry point.

> **Swing vs. JavaFX terms**
> A Swing JFrame roughly corresponds to a JavaFX Stage. A Swing JPanel roughly corresponds to a JavaFX layout pane such as VBox or StackPane.

✎ Try it yourself

1. Change the Scene size and window title.
2. Replace the StackPane with a VBox and add two Label controls stacked vertically.

CHAPTER 12

FXML & Scene Builder

Separating design from logic

FXML is an XML format for defining JavaFX layouts declaratively, similar in spirit to Android's XML layouts. It keeps your visual design separate from your Java logic, and can be edited visually with the free Scene Builder tool.

12.1 A Simple FXML Layout

```
<!-- login.fxml -->
<VBox xmlns:fx="http://javafx.com/fxml"
      fx:controller="com.zolaxtech.desktopapp.LoginController"
      spacing="10" style="-fx-padding: 20;">
    <TextField fx:id="usernameField" promptText="Username" />
    <PasswordField fx:id="passwordField" promptText="Password" />
    <Button text="Log In" onAction="#handleLogin" />
</VBox>
```

12.2 The Controller Class

```
public class LoginController {
    @FXML private TextField usernameField;
    @FXML private PasswordField passwordField;

    @FXML
    private void handleLogin() {
        String user = usernameField.getText();
        System.out.println("Logging in as: " + user);
    }
}
```

12.3 Loading FXML at Startup

```
@Override
public void start(Stage stage) throws Exception {
    Parent root = FXMLLoader.load(getClass().getResource("login.fxml"));
    stage.setScene(new Scene(root));
    stage.show();
}
```

CHAPTER 13

Styling with CSS

Giving your app a polished look

JavaFX supports CSS styling almost identically to web development, letting you theme buttons, backgrounds, fonts, and more without touching layout code.

```
/* style.css */
.button {
    -fx-background-color: #1B3A5E;
    -fx-text-fill: white;
    -fx-font-size: 14px;
    -fx-padding: 8 16;
    -fx-background-radius: 6;
}

.button:hover {
    -fx-background-color: #F2A93C;
}
```

Attach the stylesheet to your scene:

```
scene.getStylesheets().add(getClass().getResource("style.css").toExternalForm());
```

> **Tip**
> You can also target a specific control with an fx:id selector in CSS, e.g. #usernameField { -fx-border-color: #1B3A5E; } for precise, per-control styling.

CHAPTER 14

MVC Architecture in JavaFX

Structuring apps that stay maintainable

As an app grows past a single screen, mixing UI code and business logic becomes unmanageable. The Model-View-Controller (MVC) pattern keeps each concern separate and testable.

Layer	Responsibility
Model	Plain Java classes representing your data (e.g. Customer, Order) and business rules
View	FXML files defining what the user sees
Controller	Java classes (@FXML-annotated) that respond to user actions and update the Model

```
public class Customer {
    private final String name;
    private final String email;

    public Customer(String name, String email) {
        this.name = name; this.email = email;
    }
    public String getName() { return name; }
    public String getEmail() { return email; }
}

public class CustomerController {
    @FXML private TableView<Customer> customerTable;
    private final ObservableList<Customer> customers = FXCollections.observableArrayList();

    @FXML
    public void initialize() {
        customerTable.setItems(customers);
    }

    public void addCustomer(Customer c) {
        customers.add(c);
    }
}
```

Why this matters

ObservableList automatically refreshes the TableView whenever items are added or removed no manual UI refresh code required. This kind of data binding is one of JavaFX's biggest advantages over Swing.

✎ Try it yourself

1. Create a Customer model, an FXML view with a TableView, and a controller that adds a sample customer on startup.
2. Add two TableColumn objects bound to the name and email properties and confirm they display correctly.

PART FOUR

Data, Concurrency & Shipping

A finished desktop application needs to save data, stay responsive, and reach real users. This part covers the professional skills that turn a working prototype into shippable software.

CHAPTER 15

File I/O

Reading and writing data on disk

15.1 Writing a Text File

```
import java.io.*;

try (PrintWriter writer = new PrintWriter(new FileWriter("notes.txt"))) {
    writer.println("Meeting at 10am");
    writer.println("Buy office supplies");
} catch (IOException e) {
    e.printStackTrace();
}
```

15.2 Reading a Text File

```
try (BufferedReader reader = new BufferedReader(new FileReader("notes.txt"))) {
    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

15.3 Java's FileChooser (JavaFX)

```
FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("Open File");
File selected = fileChooser.showOpenDialog(stage);
if (selected != null) {
    System.out.println("Selected: " + selected.getAbsolutePath());
}
```

> **Best practice**
> Always use try-with-resources (as shown above) for files and streams it guarantees the file is closed automatically, even if an exception occurs.

CHAPTER 16

Connecting to Databases with JDBC

Persisting data in a real database

JDBC (Java Database Connectivity) is Java's standard API for talking to relational databases. It works the same way whether you connect to a lightweight embedded SQLite database or a full MySQL/PostgreSQL server.

16.1 Connecting

```
String url = "jdbc:sqlite:app_data.db";

try (Connection conn = DriverManager.getConnection(url)) {
    System.out.println("Connected to the database.");
} catch (SQLException e) {
    e.printStackTrace();
}
```

16.2 Creating a Table

```
String sql = "CREATE TABLE IF NOT EXISTS customers (" +
             "id INTEGER PRIMARY KEY AUTOINCREMENT, " +
             "name TEXT NOT NULL, " +
             "email TEXT)";

try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
    stmt.execute(sql);
}
```

16.3 Inserting & Querying Data

```
String insertSql = "INSERT INTO customers (name, email) VALUES (?, ?)";
try (Connection conn = DriverManager.getConnection(url);
     PreparedStatement ps = conn.prepareStatement(insertSql)) {
    ps.setString(1, "Selam Tesfaye");
    ps.setString(2, "selam@example.com");
    ps.executeUpdate();
}

String querySql = "SELECT * FROM customers";
```

```
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(querySql)) {
    while (rs.next()) {
        System.out.println(rs.getInt("id") + ": " + rs.getString("name"));
    }
}
```

Security tip

Always use PreparedStatement with? placeholders for any value coming from user input, as shown above. Building SQL by concatenating strings directly exposes your app to SQL injection.

✎ Try it yourself

1. Create a products table and write a method that inserts a new product record.
2. Write a method that queries all products with a price above a given value and prints them.

CHAPTER 17

Multithreading & Responsive UIs

Keeping your app from freezing

Every GUI toolkit has a single UI thread responsible for drawing and responding to input. Run a slow task a large database query, a file download directly on that thread, and your entire window freezes.

17.1 Swing: SwingWorker

```
SwingWorker<Void, Void> worker = new SwingWorker<>() {
    @Override
    protected Void doInBackground() {
        // slow work here, off the UI thread
        loadLargeReport();
        return null;
    }
    @Override
    protected void done() {
        statusLabel.setText("Report loaded!"); // back on the UI thread
    }
};
worker.execute();
```

17.2 JavaFX: Task

```
Task<Void> task = new Task<>() {
    @Override
    protected Void call() {
        loadLargeReport(); // background thread
        return null;
    }
};
task.setOnSucceeded(e -> statusLabel.setText("Report loaded!")); // UI thread
new Thread(task).start();
```

Rule of thumb

Never touch a UI component from a background thread. Both SwingWorker and JavaFX's Task exist specifically to hop back to the UI thread safely once background work finishes.

CHAPTER 18

Packaging & Distribution

Turning your app into installable software

18.1 Building a Runnable JAR

A JAR (Java ARchive) bundles your compiled classes into a single file. With Maven, the Shade or Assembly plugin produces a self-contained “fat JAR” including all dependencies:

```
mvn clean package
java -jar target/my-app-1.0.jar
```

18.2 Native Installers with jpackage

Since Java 14, the jpackage tool (bundled with the JDK) creates native installers no separate Java installation required by your end users.

```
jpackage --input target/ \
        --name ZolaxDesktopApp
\
        --main-jar  my-app-1.0.jar
\
        --main-class
com.zolaxtech.desktopapp.Main
\
        --type exe
```

Target OS	--type value	Result
Windows	exe or msi	A native Windows installer
macOS	dmg or pkg	A native macOS installer
Linux	deb or rpm	A native Linux package

> **Tip**
> jpackage can bundle a private Java runtime with your app using --runtime-image, so users never need to install Java separately at all.

CHAPTER 19

Testing & Best Practices

Writing dependable desktop software

19.1 Unit Testing with JUnit

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {
    @Test
    void additionIsCorrect() {
        Calculator calc = new Calculator();
        assertEquals(4, calc.add(2, 2));
    }
}
```

19.2 Practical Best Practices

- Keep business logic out of your Controller/UI classes so it can be unit tested without launching the UI
- Never block the UI thread always use SwingWorker or JavaFX Task for slow operations
- Validate all user input before touching a file or database
- Log errors to a file, not just the console, so you can diagnose issues after your app ships
- Version your app and use a changelog so users (and you) know what changed between releases

APPENDIX

Where to Go Next

Suggested Learning Path

1. Rebuild every code example in this guide by hand in your IDE.
2. Build three small practice apps: a notes app with file saving, a contacts manager with a database, and a simple calculator.
3. Learn Scene Builder in depth to speed up JavaFX layout design.
4. Explore advanced JavaFX: animations, charts (LineChart, PieChart), and custom controls.
5. Learn a build tool thoroughly Maven or Gradle for dependency management and packaging.
6. Package and share a small but complete app with a friend using jpackage.

Quick Reference: Java Concept → Desktop App Usage

Java Concept	Where You'll Use It
Classes & Objects	Every window, controller, and data Model class
Inheritance	extends JFrame, extends Application
Interfaces	ActionListener, Runnable, Callback
Collections (List/Map)	TableView/JTable data, caching, form state
Exception Handling	File I/O, JDBC queries, input parsing
Multithreading	SwingWorker, JavaFX Task, background jobs

Thank you for learning with ZolaxTech

More free resources at zolaxtech.com.et