OBJECT ORIENTED PROGRAMMING

U23CST34 | L:3 T:0 P:0 C:3 | 45 Periods

Complete Unit-wise Study Notes with Examples, Tables & Code

Unit	Title	Periods
Unit I	Introduction to OOP and Java	9
Unit II	Inheritance, Packages and Interfaces	9
Unit III	Exception Handling and Multithreading	9
Unit IV	I/O, Generics, String Handling	9
Unit V	JavaFX Event Handling, Controls and Components	9

CO	Course Outcome
CO1	Develop Java programs using Object Oriented Programming principles
CO2	Explain Java programs with inheritance and interface concepts
CO3	Build Java applications using exceptions
CO4	Build Java applications with I/O and generics classes
CO5	Develop interactive Java programs using JavaFX event handling
CO6	Understand the Concept of Controls components

UNIT I
INTRODUCTION TO OOP AND JAVA

1.1 Overview of OOP — Object Oriented Programming Paradigms

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data (objects) rather than functions and logic. An object is an instance of a class that bundles state (fields) and behavior (methods) together.

Paradigm	Focus	Example Languages
Procedural	Functions / procedures	C, Pascal
Object-Oriented	Objects and classes	Java, C++, Python
Functional	Mathematical functions	Haskell, Lisp
Logic	Rules and facts	Prolog

Four Pillars of OOP

- **Encapsulation:** Wrapping data and methods in a single unit (class); hiding internal state using access modifiers
- **Inheritance:** A class (child) acquiring properties and behaviors of another class (parent)
- **Polymorphism:** Same method name behaving differently depending on context (overloading / overriding)
- **Abstraction:** Hiding implementation details and showing only essential features

1.2 Features of Object Oriented Programming

- Class: Blueprint/template for creating objects
- Object: Instance of a class with state and behavior
- Message Passing: Objects communicate by calling each other's methods
- Dynamic Binding: Method to execute is determined at runtime
- Reusability: Classes can be reused in multiple programs via inheritance

Note: Java is a purely OOP language (except for primitive types). Everything lives inside a class.

1.3 Java Buzzwords

Buzzword	Meaning
Simple	Easy syntax, no pointers, automatic garbage collection
Object-Oriented	Everything is an object (except primitives)
Platform Independent	Compiled to bytecode, runs on any JVM — Write Once Run Anywhere
Robust	Strong type checking, exception handling, garbage collection
Secure	No explicit pointer arithmetic; security manager; sandboxed execution
Multithreaded	Built-in support for concurrent execution via threads
Architecture Neutral	Bytecode runs on any OS without recompilation

Portable	Same behavior on all platforms due to strict data type sizes
High Performance	Just-In-Time (JIT) compiler improves runtime speed
Distributed	Built-in network APIs (java.net); supports TCP/IP, HTTP, FTP

1.4 Overview of Java

Java was created by James Gosling at Sun Microsystems (1995). The Java platform consists of:

- JDK (Java Development Kit): Compiler (javac), tools, JRE
- JRE (Java Runtime Environment): JVM + class libraries
- JVM (Java Virtual Machine): Executes bytecode

```
// Hello World in Java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

1.5 Data Types, Variables and Arrays

Category	Type	Size	Example
Integer	byte	8-bit	byte b = 127;
Integer	short	16-bit	short s = 32000;
Integer	int	32-bit	int x = 42;
Integer	long	64-bit	long l = 100000L;
Floating	float	32-bit	float f = 3.14f;
Floating	double	64-bit	double d = 3.14159;
Character	char	16-bit Unicode	char c = 'A';
Boolean	boolean	1-bit	boolean flag = true;

Arrays

```
int[] arr = new int[5];          // single-dim array
int[][] matrix = new int[3][3];  // 2D array
String[] names = {"Alice","Bob","Charlie"};
```

□ *Note: Arrays in Java are objects stored on the heap. They have a fixed size once created.*

1.6 Operators in Java

Category	Operators	Example
Arithmetic	+ - * / % ++ --	a + b, x++
Relational	== != ><>= <=	a == b
Logical	&& || !	a && b
Bitwise	& | ^ ~ <<>>	a & b

Assignment	= += -= *= /=	x += 5
Ternary	condition ? val1 : val2	x = (a>b) ? a : b
instanceof	Checks type	obj instanceof String

1.7 Control Statements

Statement	Syntax	Purpose
if-else	if(cond){...}else{...}	Conditional branching
switch	switch(var){case x:...}	Multi-way branching
for	for(init;cond;update){...}	Fixed iteration
while	while(cond){...}	Pre-test loop
do-while	do{...}while(cond);	Post-test loop (runs at least once)
break	break;	Exit loop/switch
continue	continue;	Skip to next iteration
return	return value;	Return from method

1.8 Programming Structures in Java

A Java program has the following structure:

1. Package declaration (optional)
2. Import statements
3. Class declaration
4. Fields (variables)
5. Constructors
6. Methods

```
package com.example;
import java.util.Scanner;
public class Student {
    String name;  int age;          // fields
    Student(String n, int a) {       // constructor
        name = n; age = a;
    }
    void display() {                 // method
        System.out.println(name + "" + age);
    }
}
```

1.9 Defining Classes in Java

A class is defined using the class keyword. Key components:

- **Fields:** Variables that hold the state of an object
- **Constructor:** Special method called when object is created; same name as class; no return type
- **Methods:** Functions that define behavior of an object
- **this keyword:** Refers to the current object instance

```
class Circle {
    double radius;                          // field
    Circle(double r) { this.radius = r; }  // constructor
    double area() { return Math.PI * radius * radius; } // method
}
// Creating object:
Circle c = new Circle(5.0);
System.out.println(c.area());  // 78.539...
```

1.10 Access Specifiers

Modifier	Same Class	Same Package	Subclass	Everywhere
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

1.11 Static Members and Java Doc Comments

Static members belong to the class, not to any instance. They are shared across all objects.

```
class MathUtil {
    static int count = 0;                   // static field
    static int square(int n) { return n*n; } // static method
}
// Access without creating object:
int result = MathUtil.square(5); // 25
```

JavaDoc Comments use /** ... */ syntax and generate HTML documentation:

```
/**
 * Calculates area of rectangle.
 * @param length the length of the rectangle
 * @param width the width of the rectangle
 * @return area as double
 */
public double area(double length, double width) {
    return length * width;
}
```

UNIT II
INHERITANCE, PACKAGES AND INTERFACES

2.1 Overloading Methods

Method Overloading: defining multiple methods with the same name but different parameter lists in the same class. The compiler selects the method based on arguments passed.

```
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
    int add(int a, int b, int c) { return a + b + c; }
}
```

 Note: Return type alone cannot differentiate overloaded methods. Overloading is resolved at compile-time (static/early binding).

2.2 Objects as Parameters and Returning Objects

Objects can be passed to methods and returned from methods just like primitives.

```
class Box {
    double vol;
    Box(double v) { vol = v; }
    Box combine(Box other) {              // return object
        return new Box(this.vol + other.vol);
    }
}
```

2.3 Static, Nested and Inner Classes

Class Type	Description	Access to Outer
Static Nested	Declared static inside outer class	Only static members
Inner (Non-static)	Non-static class inside outer class	All members (static + instance)
Local Inner	Class inside a method	Local final variables
Anonymous Inner	Class with no name, defined and used once	Outer members

```
class Outer {
    int x = 10;
    class Inner {                 // inner class
        void show() { System.out.println(x); }  // accesses outer x
    }
}
Outer.Inner obj = new Outer().new Inner();
```

2.4 Inheritance: Basics

Inheritance allows a class (subclass/child) to inherit fields and methods from another class (superclass/parent). In Java, inheritance is implemented using the extends keyword.

- Code reuse — no need to rewrite common functionality
- Represents IS-A relationship (Dog IS-A Animal)
- Java supports single inheritance for classes

- Multiple inheritance achieved via interfaces

```
class Animal {
    String name;
    void eat() { System.out.println(name + " eats"); }
}
class Dog extends Animal {
    void bark() { System.out.println(name + " barks"); }
}
```

2.5 Types of Inheritance

Type	Description	Java Support
Single	One child, one parent	✓ Yes
Multilevel	A→B→C chain of inheritance	✓ Yes
Hierarchical	Multiple children from one parent	✓ Yes
Multiple	One child from multiple parents	✗ Classes: No; ✓ Interfaces: Yes
Hybrid	Combination of above	✓ Via Interfaces

2.6 super Keyword

The super keyword is used to:

7. Call the parent class constructor: super() or super(args)
8. Call overridden methods of parent: super.methodName()
9. Access hidden fields of parent: super.fieldName

```
class Vehicle {
    String brand = "Generic";
    void display() { System.out.println("Brand: " + brand); }
}
class Car extends Vehicle {
    String brand = "Toyota";
    void display() {
        super.display();  // calls parent method
        System.out.println("Car Brand: " + brand);
    }
}
```

2.7 Method Overriding

Method Overriding: subclass provides its own implementation for a method already defined in the parent class. Rules:

- Same method name, same parameters, same return type (or covariant)
- Cannot reduce access (public overriding private is OK; not reverse)
- Overriding resolved at runtime (dynamic dispatch / late binding)
- Use @Override annotation for compile-time safety

2.8 Dynamic Method Dispatch

When an overridden method is called through a parent class reference, Java determines the actual method to call at runtime based on the object's type — not the reference type. This is runtime polymorphism.

```
Animal a = new Dog();   // parent reference, child object
a.sound();              // calls Dog's sound() — decided at runtime
```

2.9 Abstract Classes

An abstract class is a class that cannot be instantiated. It may contain abstract methods (declared but not implemented) that subclasses must override.

```
abstract class Shape {
    abstract double area();       // abstract method
    void print() { System.out.println("Area: " + area()); }
}
class Circle extends Shape {
    double r;
    Circle(double r) { this.r = r; }
    double area() { return Math.PI * r * r; } // must override
}
```

Note: If a class has even one abstract method, it must be declared abstract. A class can be abstract without abstract methods.

2.10 final with Inheritance

Context	Meaning
final variable	Value cannot be changed (constant)
final method	Cannot be overridden by subclass
final class	Cannot be subclassed (e.g., String class is final)

2.11 Packages

A package is a namespace that groups related classes and interfaces. It provides access protection and avoids name conflicts.

```
// Creating a package
package com.university.student;
public class Student { ... }
```

```
// Using a package
import com.university.student.Student;
import java.util.*;     // import all classes from java.util
```

Built-in Package	Contents
java.lang	Core classes: String, Math, System, Integer (auto-imported)
java.util	Collections, Scanner, Date, Random
java.io	File I/O, streams
java.net	Networking — Socket, URL
java.awt / javax.swing	GUI components
javafx.*	Modern JavaFX UI framework

2.12 Interfaces

An interface is a contract that defines methods a class must implement. Interfaces achieve full abstraction and multiple inheritance in Java.

- All methods are public and abstract by default (before Java 8)
- Java 8+: interfaces can have default and static methods
- Java 9+: interfaces can have private methods
- All fields are public static final

```
interface Drawable {
    void draw();                              // abstract
    default void print() {                   // default method (Java 8+)
        System.out.println("Drawable object");
    }
}
class Circle implements Drawable {
    public void draw() { System.out.println("Drawing Circle"); }
}
```

Feature	Abstract Class	Interface
Instantiation	Cannot instantiate	Cannot instantiate
Methods	Abstract + concrete	Abstract + default + static
Variables	Any type	public static final only
Constructor	Has constructor	No constructor
Multiple inheritance	Single	Multiple (implements A, B, C)
Use case	Shared base behavior	Capability contracts

UNIT III
EXCEPTION HANDLING AND MULTITHREADING

3.1 Exception Handling Basics

An exception is an event that disrupts the normal flow of a program. Java has a robust exception-handling mechanism using try, catch, finally, throw, and throws.

Term	Meaning
Exception	Unexpected event at runtime that disrupts execution
try block	Encloses code that might throw an exception
catch block	Handles the specific exception type thrown
finally block	Always executes regardless of exception (cleanup code)
throw	Explicitly throw an exception object
throws	Declares that a method may throw exceptions

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    System.out.println("Always executes");
}
```

3.2 Multiple catch Clauses

A single try block can have multiple catch blocks to handle different exception types:

```
try {
    int[] arr = new int[5];
    arr[10] = 1;                    // ArrayIndexOutOfBoundsException
    int x = Integer.parseInt("abc"); // NumberFormatException
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array error: " + e);
} catch (NumberFormatException e) {
    System.out.println("Number error: " + e);
} catch (Exception e) {             // generic catch — always last
    System.out.println("General: " + e);
}
```

 Note: More specific exceptions must come before general ones. Java 7+ supports multi-catch: catch(AEx | BEx e).

3.3 Nested try Statements

try blocks can be nested inside other try blocks. Inner exceptions not handled by inner catch propagate to outer catch:

```
try {                           // outer try
    try {                       // inner try
        int x = 10/0;
```

```
    } catch(NullPointerException e) {
        System.out.println("NPE caught");
    }
} catch(ArithmeticException e) {  // catches from inner too
    System.out.println("Arithmetic: " + e);
}
```

3.4 Java's Built-in Exceptions

Exception	Cause
ArithmeticException	Division by zero
ArrayIndexOutOfBoundsException	Accessing array beyond its size
NullPointerException	Using a null reference
ClassCastException	Invalid type cast
NumberFormatException	Parsing invalid string as number
StackOverflowError	Infinite recursion
OutOfMemoryError	Heap memory exhausted
IllegalArgumentException	Illegal method argument
IOException	File / network I/O failure
FileNotFoundException	File not found

Hierarchy	Description
Throwable	Root of all exceptions and errors
Error	Serious JVM issues (StackOverflow, OutOfMemory) — don't catch
Exception	Recoverable problems — should handle
Checked	Must be caught or declared (IOException, SQLException)
Unchecked (RuntimeException)	Not required to catch (NPE, ArrayIndex)

3.5 User Defined Exceptions

Custom exceptions extend Exception (checked) or RuntimeException (unchecked):

```
class InvalidAgeException extends Exception {
    InvalidAgeException(String msg) { super(msg); }
}
class Voter {
    static void vote(int age) throws InvalidAgeException {
        if(age < 18) throw new InvalidAgeException("Age < 18!");
        System.out.println("Vote cast.");
    }
    public static void main(String[] args) {
        try { vote(15); }
        catch(InvalidAgeException e) { System.out.println(e.getMessage()); }
```

```
    }
}
```

3.6 Multithreaded Programming

A thread is an independent path of execution within a program. Java supports multithreading natively, allowing multiple threads to run concurrently.

Thread State	Description
New	Thread created but not started (start() not called)
Runnable	Thread is ready to run (after start(), waiting for CPU)
Running	Thread is currently executing on CPU
Blocked/Waiting	Thread waiting for a lock or I/O to complete
Terminated	Thread has finished execution

3.7 Java Thread Model — Creating Threads

Two ways to create a thread in Java:

Method 1: Extend Thread class

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread: " + getName() + " running");
    }
}
MyThread t = new MyThread();
t.start(); // do NOT call run() directly
```

Method 2: Implement Runnable interface (preferred)

```
class Task implements Runnable {
    public void run() {
        System.out.println("Task running");
    }
}
Thread t = new Thread(new Task());
t.start();
```

 Note: Implementing Runnable is preferred as Java does not support multiple class inheritance — the class can still extend another class.

3.8 Thread Priorities

Java threads have priorities (1–10) that influence scheduler decisions:

```
Thread.MIN_PRIORITY  = 1
Thread.NORM_PRIORITY = 5  (default)
Thread.MAX_PRIORITY  = 10
t.setPriority(Thread.MAX_PRIORITY);
int p = t.getPriority();
```

3.9 Synchronization

When multiple threads access shared data, race conditions can occur. Synchronization ensures only one thread accesses a critical section at a time.

```
class Counter {
    int count = 0;
    synchronized void increment() {  // only one thread at a time
        count++;
    }
}
```

synchronized block (more granular control):

```
synchronized(this) {
    // critical section
}
```

3.10 Inter Thread Communication

Threads can communicate using wait(), notify(), notifyAll() — must be called from synchronized context:

Method	Description
wait()	Causes current thread to wait until notify() is called
notify()	Wakes up ONE thread that called wait() on same object
notifyAll()	Wakes up ALL waiting threads

3.11 Suspending, Resuming and Stopping Threads

Modern Java deprecated suspend()/resume()/stop() due to deadlock risks. Use flags instead:

```
class SafeThread extends Thread {
    volatile boolean running = true;
    public void run() {
        while(running) {
            // do work
        }
    }
    public void stopThread() { running = false; }
}
```

3.12 Multithreading Wrappers and Auto Boxing

Auto-boxing: Automatic conversion between primitives and their wrapper classes.

Primitive	Wrapper Class
int	Integer
double	Double
boolean	Boolean
char	Character
long	Long
float	Float

```
int x = 5;
```

```
Integer obj = x;          // auto-boxing (int → Integer)
int y = obj;              // auto-unboxing (Integer → int)
Integer.parseInt("42");  // parse string to int
Integer.MAX_VALUE;        // 2147483647
```

UNIT IV
I/O, GENERICS, STRING HANDLING

4.1 I/O Basics

Java I/O (Input/Output) is based on streams. A stream is a sequence of data. Java provides two types:

Stream Type	Data	Base Classes
Byte Streams	Raw bytes (images, audio, binary)	InputStream, OutputStream
Character Streams	Text characters (Unicode)	Reader, Writer

4.2 Reading and Writing Console I/O

```
// Reading from console
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
String s = sc.next();
String line = sc.nextLine();
```

```
// Writing to console
System.out.print("No newline");
System.out.println("With newline");
System.out.printf("Formatted: %d %.2f%n", n, 3.14);
```

4.3 Reading and Writing Files

File I/O using FileReader/FileWriter (character) and FileInputStream/FileOutputStream (byte):

```
// Writing to a file
FileWriter fw = new FileWriter("output.txt");
BufferedWriter bw = new BufferedWriter(fw);
bw.write("Hello File!");
bw.newLine();
bw.close();
```

```
// Reading from a file
FileReader fr = new FileReader("output.txt");
BufferedReader br = new BufferedReader(fr);
String line;
while((line = br.readLine()) != null) {
    System.out.println(line);
}
br.close();
```

□ *Note: Always close streams in a finally block or use try-with-resources (Java 7+): try(BufferedReader br = ...) { }*

4.4 Generics: Generic Programming

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. Benefits:

- Type safety at compile time — prevents ClassCastException at runtime
- Code reuse — write once, use with any type
- Eliminates casting

```
// Generic class
class Box<T> {
    T value;
    Box(T v) { value = v; }
    T get() { return value; }
}
Box<Integer> intBox = new Box<>(42);
Box<String>  strBox = new Box<>("Hello");
```

4.5 Generic Classes and Generic Methods

```
// Generic method
class Util {
    public static <T> void swap(T[] arr, int i, int j) {
        T temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
    }
}
Integer[] nums = {1,2,3,4};
Util.swap(nums, 0, 3);  // {4,2,3,1}
```

Multiple Type Parameters

```
class Pair<K, V> {
    K key; V value;
    Pair(K k, V v) { key=k; value=v; }
}
Pair<String,Integer> p = new Pair<>("Age", 25);
```

4.6 Bounded Types

Bounded type parameters restrict what types can be used as type arguments:

```
// Upper bound — T must be Number or its subclass
class Statistics<T extends Number> {
    T num;
    double square() { return num.doubleValue() * num.doubleValue(); }
}
```

```
// Wildcard — unknown type
void printList(List<?> list) { ... }           // any type
void sum(List<? extends Number> list) { ... }  // Number or subtype
void add(List<? super Integer> list) { ... }   // Integer or supertype
```

4.7 Restrictions and Limitations of Generics

- Cannot instantiate generic type: new T() is not allowed
- Cannot create arrays of generic types: new T[10] is not allowed
- Cannot use primitives as type arguments: Box<int> is invalid, use Box<Integer>
- Static members cannot use type parameters
- Cannot use instanceof with generic type: obj instanceof T is illegal

Note: Generics are erased at runtime (type erasure). Generic type information exists only at compile time.

4.8 Strings: Basic String Class

String in Java is an immutable sequence of characters (java.lang.String):

```
String s1 = "Hello";                    // string literal (pool)
String s2 = new String("Hello");         // new object on heap
String s3 = s1 + " World";               // concatenation
```

4.9 String Methods

Method	Description	Example
length()	Returns length	"Hello".length() → 5
charAt(i)	Char at index	"Hi".charAt(0) → 'H'
substring(i,j)	Substring from i to j-1	"Hello".substring(1,3) → "el"
indexOf(s)	First occurrence index	"Hello".indexOf('l') → 2
toUpperCase()	Convert to upper case	"hi".toUpperCase() → "HI"
toLowerCase()	Convert to lower case	"HI".toLowerCase() → "hi"
trim()	Remove leading/trailing spaces	" hi ".trim() → "hi"
replace(a,b)	Replace all occurrences	"aaa".replace('a','b') → "bbb"
split(regex)	Split into array	"a,b".split(",") → ["a","b"]
equals(s)	Content comparison	"hi".equals("hi") → true
compareTo(s)	Lexicographic comparison	"a".compareTo("b") → negative
contains(s)	Check substring	"Hello".contains("ell") → true
isEmpty()	Check if empty	"".isEmpty() → true
valueOf(x)	Convert to String	String.valueOf(42) → "42"

4.10 StringBuffer Class

StringBuffer is a mutable sequence of characters (thread-safe). Use when string is modified frequently:

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");   // Hello World
sb.insert(5, ",");     // Hello, World
sb.delete(5, 6);       // Hello World
sb.reverse();          // dlroW olleH
sb.replace(0,5,"Hi"); // Hi World
```

Class	Mutable	Thread-safe	Speed
String	No	Yes (immutable)	Fast for read
StringBuffer	Yes	Yes (synchronized)	Slower (thread-safe)
StringBuilder	Yes	No	Fastest (single-thread)

UNIT V
JAVAFX EVENT HANDLING, CONTROLS AND COMPONENTS

5.1 Introduction to JavaFX

JavaFX is Java's modern GUI framework, replacing Swing for building rich desktop applications. It uses a scene graph — a hierarchical tree of nodes representing UI elements.

Concept	Description
Stage	Top-level window (like JFrame in Swing)
Scene	Container for all UI content; added to Stage
Node	Any UI element (Button, Label, TextField, etc.)
Scene Graph	Tree of nodes that makes up the UI
Application	Base class for all JavaFX apps (extend this)

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
public class HelloFX extends Application {
    public void start(Stage stage) {
        Label lbl = new Label("Hello JavaFX!");
        StackPane root = new StackPane(lbl);
        Scene scene = new Scene(root, 400, 300);
        stage.setTitle("My App");
        stage.setScene(scene);
        stage.show();
    }
    public static void main(String[] a) { launch(a); }
}
```

5.2 JavaFX Events and Controls

JavaFX uses an event-driven programming model. User interactions generate events that are handled by event handlers.

Event Type	Description
ActionEvent	Button click, menu item selection, key press (Enter)
MouseEvent	Mouse click, move, drag, press, release
KeyEvent	Key pressed, released, typed
WindowEvent	Window open, close, minimize
DragEvent	Drag and drop operations

```
Button btn = new Button("Click Me");
```

```
btn.setOnAction(event -> {
    System.out.println("Button clicked!");
});
```

5.3 Event Basics — Handling Key and Mouse Events

```
// Key Event
scene.setOnKeyPressed(e -> {
    System.out.println("Key: " + e.getCode());
});
```

```
// Mouse Event
pane.setOnMouseClicked(e -> {
    System.out.println("X:" + e.getX() + " Y:" + e.getY());
});
```

5.4 Controls

Checkbox

```
CheckBox cb = new CheckBox("Agree to Terms");
cb.setOnAction(e -> {
    System.out.println(cb.isSelected() ? "Checked" : "Unchecked");
});
```

Toggle Button

```
ToggleButton tb = new ToggleButton("ON/OFF");
tb.setOnAction(e -> System.out.println(tb.isSelected()));
```

Radio Buttons

```
RadioButton rb1 = new RadioButton("Male");
RadioButton rb2 = new RadioButton("Female");
ToggleGroup group = new ToggleGroup();
rb1.setToggleGroup(group);
rb2.setToggleGroup(group);
```

5.5 More Controls

Control	Class	Key Methods
List View	ListView<T>	getSelectionModel(), getItems()
Combo Box	ComboBox<T>	getValue(), getItems()
Choice Box	ChoiceBox<T>	getValue(), getItems()
Text Controls	TextField, TextArea, PasswordField	getText(), setText()
Scroll Controls	ScrollPane	setContent(), setFitToWidth()
Slider	Slider	getValue(), setMin(), setMax()
ProgressBar	ProgressBar	setProgress(0.0 to 1.0)
Label	Label	setText(), setGraphic()

Button	Button	setOnAction()
Hyperlink	Hyperlink	setOnAction()

```
// ComboBox example
ComboBox<String> combo = new ComboBox<>();
combo.getItems().addAll("Java","Python","C++");
combo.setValue("Java");  // default selection
String selected = combo.getValue();
```

5.6 Layouts

Layout	Description	Key Properties
FlowPane	Flows nodes in rows/columns like text	hgap, vgap, orientation
HBox	Horizontal row of nodes	spacing, alignment
VBox	Vertical column of nodes	spacing, alignment
BorderPane	5 regions: Top, Bottom, Left, Right, Center	setTop(), setCenter()
StackPane	Stacks nodes on top of each other	alignment
GridPane	Grid of rows and columns	add(node,col,row), hgap, vgap
AnchorPane	Anchors children to edges	setTopAnchor(), setLeftAnchor()
TilePane	Uniform tiles	hgap, vgap, prefColumns

```
// GridPane example
GridPane grid = new GridPane();
grid.setHgap(10); grid.setVgap(10);
grid.add(new Label("Name:"), 0, 0);
grid.add(new TextField(), 1, 0);
grid.add(new Label("Age:"),  0, 1);
grid.add(new TextField(), 1, 1);
```

5.7 Menus

```
MenuBar menuBar = new MenuBar();
Menu fileMenu = new Menu("File");
MenuItem openItem = new MenuItem("Open");
MenuItem saveItem = new MenuItem("Save");
MenuItem exitItem = new MenuItem("Exit");
exitItem.setOnAction(e -> Platform.exit());
fileMenu.getItems().addAll(openItem, saveItem,
    new SeparatorMenuItem(), exitItem);
menuBar.getMenus().add(fileMenu);
```

Menu Class	Purpose
MenuBar	Top-level container for menus
Menu	Drop-down menu (has sub-items)

MenuItem	Clickable item inside a Menu
CheckMenuItem	Menu item with checkbox
RadioMenuItem	Menu item that can be selected (radio-style)
SeparatorMenuItem	Horizontal separator between items
ContextMenu	Right-click popup menu

5.8 Full Sample JavaFX Application

```
public class LoginApp extends Application {
    public void start(Stage stage) {
        GridPane grid = new GridPane();
        grid.setPadding(new Insets(20));
        grid.setHgap(10); grid.setVgap(10);
        Label userLbl = new Label("Username:");
        TextField userTF = new TextField();
        Label passLbl = new Label("Password:");
        PasswordField passTF = new PasswordField();
        Button loginBtn = new Button("Login");
        Label msg = new Label();
        loginBtn.setOnAction(e -> {
            if(userTF.getText().equals("admin")
&& passTF.getText().equals("1234"))
                msg.setText("Login Successful!");
            else msg.setText("Invalid credentials!");
        });
        grid.add(userLbl,0,0); grid.add(userTF,1,0);
        grid.add(passLbl,0,1); grid.add(passTF,1,1);
        grid.add(loginBtn,1,2); grid.add(msg,1,3);
        stage.setScene(new Scene(grid,300,200));
        stage.setTitle("Login"); stage.show();
    }
    public static void main(String[] a){ launch(a); }
}
```

QUICK REVISION — ALL UNITS

Important OOP Concepts at a Glance

Concept	Definition	Java Keyword
Encapsulation	Hiding data inside a class	private, get/set
Inheritance	Child class inherits from parent	extends
Polymorphism	Same method, different behavior	overload/override
Abstraction	Hide complexity, show essentials	abstract, interface
Interface	Contract for classes to follow	implements
Package	Group of related classes	package, import
Exception	Runtime error that can be handled	try/catch/throw
Thread	Independent execution path	Thread, Runnable
Generic	Type-safe reusable code	<T>
JavaFX	Modern UI framework	Application, Stage

Java Keywords Summary

Keyword	Purpose
class / interface	Define a class or interface
extends / implements	Inheritance / interface implementation
abstract / final	Abstract method/class; prevent override/extend
static / this / super	Class member; current/parent object reference
new	Create object instance
try/catch/finally/throw/throws	Exception handling
synchronized	Thread-safe code block/method
import / package	Namespace management
public/private/protected	Access control
void / return	No return / return value from method

Common Exam Questions & Answers

Question	Short Answer
Difference between overloading and overriding?	Overloading: same class, different params; Overriding: parent-child, same signature
What is dynamic dispatch?	Runtime selection of overridden method based on actual object type

Checked vs Unchecked exception?	Checked: must handle (IOException); Unchecked: optional (NullPointerException)
Why Runnable over Thread?	Java has single inheritance; Runnable class can still extend another
What is type erasure?	Generic type info is removed at compile time; all generics become Object at runtime
StringBuffer vs StringBuilder?	StringBuffer: thread-safe (synchronized); StringBuilder: faster, not thread-safe
What is JavaFX Scene Graph?	Hierarchical tree of UI nodes (Stage → Scene → Layout → Controls)
Interface default methods?	Java 8+ allows concrete methods in interfaces using default keyword
What is UTXO? (wrong course!)	N/A — check you're studying the right material!
What is auto-boxing?	Auto conversion between primitive (int) and wrapper class (Integer)

— End of Study Material | U23CST34 Object Oriented Programming | 45 Periods —