

PROGRAMMING IN JAVA

1.Introduction to Java

Java Architecture : –



1. Java Development Kit (JDK)

- **Compiler (javac):** Converts Java source code into bytecode.
- **Debugger (jdb):** Helps in debugging Java programs.
- **JavaDoc:** Generates documentation from Java source code.
- **Other Tools:** Additional tools for developing, debugging, and monitoring Java applications.

2. Java Runtime Environment (JRE)

- **Libraries:** JRE includes essential libraries and APIs required to run Java applications.
- **JVM:** JRE contains the JVM and other components necessary to run Java programs.
-

3. Java Virtual Machine (JVM)

- **Bytecode Execution:** Java programs are compiled into bytecode, which the JVM interprets and executes. This allows Java programs to be platform-independent.
- **Garbage Collection:** JVM automatically manages memory allocation and deallocation, cleaning up unused objects to free memory.
- **Security:** JVM enforces a strict security model, isolating Java programs from the underlying system and each other.

Java Features : –

1. Platform Independence

- Java programs are compiled into bytecode, which can run on any platform with a compatible JVM.

2. Object-Oriented

- Java follows the principles of object-oriented programming (OOP) such as inheritance, encapsulation, polymorphism, and abstraction.

3. Simple

- Java has a simpler syntax compared to languages like C++, making it easier to learn and use.

4. Secure

- Java provides a secure execution environment with features like bytecode verification, sandboxing, and an extensive set of APIs for security management.

5. Robust

- Java has strong memory management, exception handling, and type-checking mechanisms.

6. Multithreaded

- Java supports multithreading, allowing concurrent execution of two or more threads.

7. Distributed

- Java has a rich set of APIs for networking, making it suitable for distributed computing.

8. High Performance

- Java's performance is enhanced by Just-In-Time (JIT) compilers and efficient garbage collection.

Syntax Differences between C++ and Java : –

1. Basic Structure

- C++: Uses header files and the `main()` function.
- Java: Uses packages and the `public static void main(String[] args)` method.

```
// C++
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, World!";
    return 0;
}
```

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

2.Memory Management

- C++: Manual memory management using `new`, `delete`, `malloc`, and `free`.
- Java: Automatic garbage collection.

3.Inheritance

- C++: Supports multiple inheritance.
- Java: Supports single inheritance, with multiple inheritance achieved through interfaces.

4.Exception Handling

- C++: Uses `try`, `catch`, and `throw`, but exception handling is less emphasized.
- Java: Exception handling is a core feature with a robust hierarchy of exception classes.

5.Pointers

- C++: Extensive use of pointers.
- Java: No direct use of pointers; references are used instead.

6.Templates and Generics

- C++: Uses templates for generic programming.
- Java: Uses generics for type-safe collections and other generic programming features.

Semantic Differences between Java and C++ : -

Understanding the semantic differences between Java and C++ helps in grasping how the same constructs behave differently in these languages.

1. Memory Management

- **C++:** Memory management is manual. Developers use `new` and `delete` for dynamic memory allocation and deallocation. This can lead to memory leaks if not handled correctly.
- **Java:** Memory management is automatic through garbage collection, which periodically removes unused objects, reducing the risk of memory leaks.

2. Inheritance

- **C++:** Supports multiple inheritance, allowing a class to inherit from more than one base class. This can lead to complexity and issues like the diamond problem.
- **Java:** Does not support multiple inheritance of classes. Instead, it uses interfaces to achieve similar functionality, simplifying the inheritance structure and avoiding problems like the diamond problem.

3. Exception Handling

- **C++:** Exception handling is available but is less emphasized and often not used consistently across different codebases.
- **Java:** Exception handling is a core feature, with a well-defined hierarchy of exception classes and enforced use of try-catch blocks for error handling.

4. Object Orientation

- **C++:** Is a multi-paradigm language that supports both procedural and object-oriented programming. This allows flexibility but can lead to mixed programming styles.
- **Java:** Is purely object-oriented, meaning almost everything is an object. This enforces a consistent object-oriented approach throughout the code.

5. Runtime Environment

- **C++:** Programs are compiled directly into machine code, making them platform-dependent. The behaviour of the program is directly tied to the underlying hardware and operating system.
- **Java:** Programs are compiled into bytecode, which is interpreted by the JVM. This makes Java programs platform-independent, as the JVM abstracts the underlying hardware and operating system.

6. Type System

- **C++**: Offers more control over types, including the ability to use low-level constructs like pointers and perform explicit type casting.
- **Java**: Enforces strict type checking and does not allow explicit pointer manipulation, enhancing security and reducing bugs related to type errors.

Semantics in simple terms is MEANINGFUL Sentence, which is formed by using correctly well structured syntax.

For Example :

In normal English, if you say: “ She is a boy “ . Now this sentence is syntactically correct , but does not give any proper meaning. It's ambiguous.

Similarly, in programming language if you write `string name= 5 ; x ="a" +1; int num="arjun"`

These are semantically wrong. Since it makes no sense .

Compiling and Executing a Java Program : –

Convert Java source code (`.java`) to bytecode (`.class`) and Run the bytecode using the JVM.

1. Writing a Java Program

Java programs are written in `.java` files. Here's an example of a simple Java program:

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

2. Compiling the Java Program

Before running a Java program, you must compile it to convert the source code into bytecode.

- **Command Line:** Open the terminal or command prompt, navigate to the directory where your `.java` file is located, and run:

```
javac HelloWorld.java
```

- **Output:** This command will generate a `HelloWorld.class` file containing the bytecode.

3. Executing the Java Program

After compiling, you can execute the program:

- **Command Line:** Run the following command:

```
java HelloWorld
```

- **Output:** This will execute the bytecode on the JVM, and you should see the output:

```
Hello, World!
```

Variables : –

Variables are containers for storing data values. In Java, each variable has a type that determines what kind of data it can hold.

- Syntax:

```
java  
  
dataType variableName = value;
```

Example:

```
java  
  
int age = 20;  
String name = "John";
```

Types of Variables:

- **Local Variables:** Declared inside a method or block and accessible only within it.
- **Instance Variables:** Declared inside a class but outside any method, and they belong to an instance of the class.
- **Static Variables:** Declared with the `static` keyword and belong to the class rather than any instance.

Constants : –

Constants are variables whose values cannot be changed once assigned. They are declared using the `final` keyword.

- Syntax:

```
java  
  
final dataType CONSTANT_NAME = value;
```

Example:

```
java  
  
final int MAX_VALUE = 100;
```

Keywords: –

Keywords are reserved words in Java that have a predefined meaning and cannot be used as identifiers (variable names, function names, etc.).

Examples of Keywords:

Control flow: `if`, `else`, `switch`, `case`, `for`, `while`, `do`, `break`, `continue`

Access modifiers: `public`, `private`, `protected`

Class-related: `class`, `interface`, `extends`, `implements`, `abstract`

Data types: `int`, `float`, `char`, `double`, `boolean`, `void`

Others: `static`, `final`, `try`, `catch`, `finally`, `throw`, `throws`, `return`, `this`, `new`

Data Types : –

Data types specify the type of data a variable can hold. Java has two main categories of data types:

1. Primitive Data Types

2. Reference Data Types.

1. Primitive Data Types : –

Java has eight primitive data types, which are the most basic data types.

Integer Types:

- **byte**: 1 byte, range: -128 to 127
- **short**: 2 bytes, range: -32,768 to 32,767
- **int**: 4 bytes, range: -2^{31} to $2^{31}-1$
- **long**: 8 bytes, range: -2^{63} to $2^{63}-1$

Floating-Point Types:

- **float**: 4 bytes, single-precision floating point
- **double**: 8 bytes, double-precision floating point

Character Type:

- **char**: 2 bytes, stores a single character/letter or ASCII values

Boolean Type:

- **boolean**: 1 bit, stores **true** or **false**

2. Reference Data Types

Reference data types refer to objects and arrays.

- **Examples:**
 - **Strings:** `String name = "John";`
 - **Arrays:** `int[] numbers = {1, 2, 3};`
 - **Classes:** `MyClass obj = new MyClass();`
 -

Example of Using Variables and Data Types

```
public class DataTypesExample {
    public static void main(String[] args) {
        int age = 25; // Integer variable
        float salary = 5000.75f; // Floating-point variable
        char grade = 'A'; // Character variable
        boolean isPassed = true; // Boolean variable
        String name = "Alice"; // String variable

        final int MAX_LIMIT = 100; // Constant

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Salary: " + salary);
        System.out.println("Grade: " + grade);
        System.out.println("Passed: " + isPassed);
        System.out.println("Max Limit: " + MAX_LIMIT);
    }
}
```

Operators (Arithmetic, Logical and Bitwise) and Expressions : –

- 1.Arithmetic Operator
- 2.Relational Operator
- 3.Logical Operator
- 4.Bitwise Operator
- 5.Assignment Operator
- 6.Unary Operator

1) Arithmetic Operators

Used for basic mathematical operations.

Operator	Description	Example (a=10, b=5)	Result
+	Addition	a + b	15
-	Subtraction	a - b	5
*	Multiplication	a * b	50
/	Division	a / b	2
%	Modulus (remainder)	a % b	0

```
public class ArithmeticOperators {
    public static void main(String[] args) {
        int a = 10, b = 5;
        System.out.println("Addition: " + (a + b));           // 15
        System.out.println("Subtraction: " + (a - b));        // 5
        System.out.println("Multiplication: " + (a * b));      // 50
        System.out.println("Division: " + (a / b));            // 2
        System.out.println("Modulus: " + (a % b));             // 0
    }
}
```

(2) Logical Operators

Used to combine boolean expressions

Used with boolean expressions.

Operator	Description	Example	Result
<code>&&</code>	Logical AND	<code>(a > b && b > 0)</code>	true
<code>&&</code>			Logical OR
<code>!</code>	Logical NOT	<code>!(a > b)</code>	false

```
public class LogicalOperators {  
    public static void main(String[] args) {  
        int a = 10, b = 5;  
        boolean x = (a > b);  
        boolean y = (a < b);  
  
        System.out.println(x && y); // false  
        System.out.println(x || y); // true  
        System.out.println(!x);    // false  
    }  
}
```

(3) Bitwise Operators

Operate at the binary level. Work on bits (0 and 1).

Operator	Description	Example (a=5, b=3)	Result (in binary)
<code>&</code>	AND	<code>a & b</code>	1 (0101 & 0011 = 0001)
<code> </code>	OR	<code>a b</code>	7 (0101 0011 = 0111)
<code>^</code>	XOR	<code>a ^ b</code>	6 (0101 ^ 0011 = 0110)
<code>~</code>	NOT	<code>~a</code>	-6 (in 2's complement)
<code><<</code>	Left shift	<code>a << 1</code>	10 (0101 << 1 = 1010)
<code>>></code>	Right shift	<code>a >> 1</code>	2 (0101 >> 1 = 0010)

```
public class BitwiseOperators {
    public static void main(String[] args) {
        int a = 5; // 0101 in binary
        int b = 3; // 0011 in binary

        System.out.println("a & b = " + (a & b)); // 1
        System.out.println("a | b = " + (a | b)); // 7
        System.out.println("a ^ b = " + (a ^ b)); // 6
        System.out.println("~a = " + (~a)); // -6 (2's complement)
        System.out.println("a << 1 = " + (a << 1)); // 10
        System.out.println("a >> 1 = " + (a >> 1)); // 2
    }
}
```

(4) Unary Operators

Operate on one operand.

```

public class UnaryOperators {
    public static void main(String[] args) {
        int a = 10;
        System.out.println("Post-increment: " + (a++)); // 10
        System.out.println("After post-increment: " + a); // 11
        System.out.println("Pre-increment: " + (++a)); // 12

        int b = -a;
        System.out.println("Unary minus: " + b); // -12
    }
}

```

(5) Assignment Operators

Assign values and perform operations in one step.

```

public class AssignmentOperators {
    public static void main(String[] args) {
        int a = 10;

        a += 5; // a = a + 5
        System.out.println(a); // 15

        a -= 3; // a = a - 3
        System.out.println(a); // 12

        a *= 2; // a = a * 2
        System.out.println(a); // 24

        a /= 4; // a = a / 4
        System.out.println(a); // 6

        a %= 5; // a = a % 5
        System.out.println(a); // 1
    }
}

```

(6) Relational (Comparison) Operators

Compare two values.

```
public class RelationalOperators {  
    public static void main(String[] args) {  
        int a = 10, b = 5;  
        System.out.println(a == b); // false  
        System.out.println(a != b); // true  
        System.out.println(a > b); // true  
        System.out.println(a < b); // false  
        System.out.println(a >= b); // true  
        System.out.println(a <= b); // false  
    }  
}
```

Comments in Java : –

1. Used to make code readable.
2. For Not Running specific line or multi lines we use comments

Single-line comment: *// This is a comment*

Multi-line comment:

```
/* This is a  
multi-line comment */
```

Basic Java Program Structure : –

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

Decision Making Constructs: –

(1) if Statement

```
if (a > b) {  
    System.out.println("a is greater");  
}
```

(2) if-else Statement

```
if (a > b) {  
    System.out.println("a is greater");  
} else {  
    System.out.println("b is greater");  
}
```

(3) if-else-if Ladder

```
if (a > b) {  
    // code  
} else if (a == b) {  
    // code  
} else {  
    // code  
}
```

Loops and Nesting: –

1.for loop

```
for (int i = 1; i <= 5; i++) {  
    System.out.println(i);  
}
```


2.while loop

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}
```

3.do-while loop

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

4.Nested Loops

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

Java Methods: –

- A method is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a method.
- Methods are used to perform certain actions, and they are also known as functions.
- Why use methods? To reuse code: define the code once, and use it many times.

(a) Defining a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses (). Java provides some pre-defined methods, such as `System.out.println()`, but you can also create your own methods to perform certain actions:

```
public class Main {  
    static void myMethod() {  
        // code to be executed  
    }  
}
```

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main class.
- `void` means that this method does not have a return value.

(b) Calling a Method

To call a method in Java, write the method's name followed by two parentheses () and a semicolon;

In the following example, `myMethod()` is used to print a text (the action), when it is called:

Example

Inside `main`, call the `myMethod()` method:

```
public class Main {
    static void myMethod() {
        System.out.println("I just got executed!");
    }

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

// Outputs "I just got executed!"
```

A method can also be called multiple times

```
public class Main {
    static void myMethod() {
        System.out.println("I just got executed!");
    }

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

// I just got executed!
// I just got executed!
// I just got executed!
```

Java Scope: –

In Java, variables are only accessible inside the region they are created. This is called scope.

Scope determines the **visibility or accessibility** of a variable.

♦ 1. Local Variables

➤ Definition:

A **local variable** is declared **inside a method, constructor, or block** and is accessible **only within that block**.

➤ Key Points:

- Declared inside methods or loops.
- Cannot be accessed outside the method/block.
- No default value – must be initialized before use.

```
public class LocalExample {  
    public void display() {  
        int x = 10; // local variable  
        System.out.println("Local variable x: " + x);  
    }  
  
    public static void main(String[] args) {  
        LocalExample obj = new LocalExample();  
        obj.display();  
    }  
}
```

♦ 2. Instance Variables

➤ Definition:

An **instance variable** is declared **inside a class but outside any method**. It belongs to the object of the class.

➤ Key Points:

- Created when an object is created.
- Has a default value (e.g., 0 for int, null for objects).
- Each object gets its **own copy** of the variable.

```
public class InstanceExample {  
    int a = 5; // instance variable  
  
    public void show() {  
        System.out.println("Instance variable a: " + a);  
    }  
  
    public static void main(String[] args) {  
        InstanceExample obj1 = new InstanceExample();  
        obj1.show(); // prints 5  
  
        InstanceExample obj2 = new InstanceExample();  
        obj2.a = 10;  
        obj2.show(); // prints 10  
        obj1.show(); // still prints 5 (different copy)  
    }  
}
```

♦ 3. Static Variables

➤ Definition:


A **static variable** is declared with the `static` keyword. It belongs to the **class** rather than any object.

➤ Key Points:

- Shared among all objects of the class.
- Only one copy exists in memory.
- Can be accessed using **class name** or object.

```
public class StaticExample {  
    static int count = 0; // static variable  
  
    StaticExample() {  
        count++;  
        System.out.println("Count is: " + count);  
    }  
  
    public static void main(String[] args) {  
        StaticExample obj1 = new StaticExample(); // Count is 1  
        StaticExample obj2 = new StaticExample(); // Count is 2  
        StaticExample obj3 = new StaticExample(); // Count is 3  
    }  
}
```

◆ Comparison Table

Feature	Local Variable	Instance Variable	Static Variable	
Defined in	Inside method/block	Inside class (outside method)	Inside class (with <code>static</code>)	
Scope	Limited to method/block	Accessible by object	Shared among all objects	
Default Value	None (must be initialized)	Yes (0, null, etc.)	Yes	
Memory Allocation	On method call stack	When object is created	When class is loaded	
Accessed by	Method/block only	Object (<code>obj.var</code>)	Class or object (<code>Class.var</code>)	

✓ Summary:

- Use **local variables** for temporary values within methods.
- Use **instance variables** when each object should have its own data.
- Use **static variables** for values shared by all instances (like counters).

Passing and Returning Arguments: –

◆ Parameters in Java

- Parameters are variables listed **inside the parentheses in a method definition**.
- They are used to receive values when the method is called.

```
public class Example {  
    // This method has one parameter: 'name'  
    public static void greet(String name) {  
        System.out.println("Hello, " + name);  
    }  
}
```

♦ Arguments in Java

- Arguments are the **actual values** passed to the method **when it is called**.

```
public class Main {  
    public static void main(String[] args) {  
        Example.greet("Alice"); // "Alice" is an argument  
        Example.greet("Bob");   // "Bob" is another argument  
    }  
}
```

Java uses Pass by Value for both primitive and reference types.

♦ 1. Passing Arguments to Methods

Java passes **a copy** of the variable's value. Changes inside the method **do not affect** the original value.

```
public class PassPrimitive {  
    public static void changeValue(int x) {  
        x = 100;  
        System.out.println("Inside method: " + x);  
    }  
  
    public static void main(String[] args) {  
        int a = 50;  
        changeValue(a);  
        System.out.println("Outside method: " + a);  
    }  
}
```


♦ 2. Returning Values from Methods

```
public class ReturnExample {  
    public static int square(int n) {  
        return n * n;  
    }  
  
    public static void main(String[] args) {  
        int result = square(5);  
        System.out.println("Square is: " + result);  
    }  
}
```

Type Conversion in Java: –

Type conversion refers to changing data from one type to another.

✓ Two Types of Type Conversion:

♦ A. Implicit Type Conversion (Widening Conversion)

- Happens **automatically** by Java.
- Converts **smaller type to larger type**.
- **No data loss**.

```

public class ImplicitConversion {
    public static void main(String[] args) {
        int a = 10;
        double b = a; // int to double
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
}

```

Output:

```

ini

a = 10
b = 10.0

```

♦ B. Explicit Type Conversion (Narrowing Conversion / Type Casting)

- Performed **manually** by the programmer.
- Converts **larger type to smaller type**.
- May cause **data loss** or **overflow**.

```

public class ExplicitConversion {
    public static void main(String[] args) {
        double d = 9.7;
        int i = (int) d; // double to int
        System.out.println("d = " + d);
        System.out.println("i = " + i);
    }
}

```

Output:

```
ini  
  
d = 9.7  
i = 9
```

Type checking ensures **type safety** — that variables and expressions are used with compatible types.

♦ A. Compile-time Type Checking

Java is a **statically-typed language** — it checks data types **at compile time**.

✓ Example:

```
java  
  
int a = "hello"; // ✗ Compile-time error: incompatible types
```

♦ B. Runtime Type Checking using **instanceof** operator

Used to **check the actual type of an object at runtime** before casting it.

Example

```
public class TypeCheck {  
    public static void main(String[] args) {  
        String s = "Java";  
        if (s instanceof String) {  
            System.out.println("s is a String");  
        }  
    }  
}
```

Output:

vbnet

s is a String

Built in Java Class Method

Predefined methods available in **Java API classes** (like `Math`, `String`, `Integer`, etc.).

Help you write efficient and cleaner code without reinventing logic.

1. Math Class (java.lang.Math)

✓ Common Methods:

Method	Description	Example	Output
<code>Math.abs(x)</code>	Absolute value	<code>Math.abs(-10)</code>	10
<code>Math.sqrt(x)</code>	Square root	<code>Math.sqrt(16)</code>	4.0
<code>Math.pow(a, b)</code>	a raised to b	<code>Math.pow(2, 3)</code>	8.0
<code>Math.max(a, b)</code>	Maximum of two	<code>Math.max(3, 5)</code>	5
<code>Math.min(a, b)</code>	Minimum of two	<code>Math.min(3, 5)</code>	3
<code>Math.round(x)</code>	Rounds to nearest	<code>Math.round(4.6)</code>	5

♦ 2. String Class (`java.lang.String`)

Provides methods to manipulate text (strings).

✓ Common Methods:

Method	Description	Example	Output
<code>length()</code>	Length of string	<code>"Hello".length()</code>	5
<code>charAt(index)</code>	Character at index	<code>"Java".charAt(1)</code>	'a'
<code>toUpperCase()</code>	Convert to upper case	<code>"java".toUpperCase()</code>	"JAVA"
<code>toLowerCase()</code>	Convert to lower case	<code>"JAVA".toLowerCase()</code>	"java"
<code>substring(start, end)</code>	Extract substring	<code>"hello".substring(1, 4)</code>	"ell"
<code>equals()</code>	Compare content	<code>"hi".equals("hi")</code>	true
<code>compareTo()</code>	Lexical comparison	<code>"a".compareTo("b")</code>	-1

♦ 3. Wrapper Classes (Integer, Double, etc.)

Used to wrap primitive types into objects and provide utility methods.

✓ Common Integer Methods:

Method	Description	Example	Output
<code>parseInt(String)</code>	String to int	<code>Integer.parseInt("123")</code>	123
<code>toString(int)</code>	Int to String	<code>Integer.toString(456)</code>	"456"
<code>compare(a, b)</code>	Compare two ints	<code>Integer.compare(4, 2)</code>	1

♦ 4. Character Class

Provides utility methods for working with characters.

✓ Common Methods:

Method	Description	Example	Output
<code>isDigit(char)</code>	Checks if digit	<code>Character.isDigit('5')</code>	true
<code>isLetter(char)</code>	Checks if letter	<code>Character.isLetter('A')</code>	true
<code>toUpperCase(char)</code>	Converts to upper	<code>Character.toUpperCase('a')</code>	'A'

♦ 5. System Class

Used for basic system-related functions.

✓ Common Methods:

Method	Description
<code>System.out.println()</code>	Prints with newline
<code>System.currentTimeMillis()</code>	Returns current time in ms
<code>System.exit(0)</code>	Terminates the program

2. Arrays, Strings and I/O

1. Creating & Using Arrays

An **array** is a data structure that stores multiple values of the same type in a single variable, instead of declaring separate variables for each value.

◆ 1.1 One-Dimensional Arrays

➤ Declaration of an Array

```
int[] numbers; // Recommended
// OR
int numbers[]; // Also valid
```

➤ Instantiation (Memory Allocation)

```
numbers = new int[5]; // Creates an array that can hold 5 integers
```

➤ Declaration + Instantiation

```
int[] numbers = new int[5];
```

➤ Initialization

```
numbers[0] = 10;  
numbers[1] = 20;  
numbers[2] = 30;  
numbers[3] = 40;  
numbers[4] = 50;
```

➤ Combined Declaration, Instantiation & Initialization

```
int[] numbers = {10, 20, 30, 40, 50};
```

➤ Accessing Array Elements

```
System.out.println(numbers[2]); // Output: 30
```

```
public class OneDArrayExample {  
    public static void main(String[] args) {  
        int[] marks = {90, 85, 88, 92, 76};  
  
        System.out.println("Student Marks:");  
        for (int i = 0; i < marks.length; i++) {  
            System.out.println("Student " + (i+1) + ": " + marks[i]);  
        }  
    }  
}
```

◆ 1.2 Multi-Dimensional Arrays

A multi-dimensional array is essentially an array of arrays.

➤ Declaration and Instantiation


```
int[][] matrix = new int[2][3]; // 2 rows, 3 columns
```

➤ Initialization

```
matrix[0][0] = 1;  
matrix[0][1] = 2;  
matrix[0][2] = 3;  
matrix[1][0] = 4;  
matrix[1][1] = 5;  
matrix[1][2] = 6;
```

➤ Combined Declaration & Initialization

```
int[][] matrix = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

```

public class MultiDArrayExample {
    public static void main(String[] args) {
        int[][] table = {
            {1, 2, 3},
            {4, 5, 6}
        };

        System.out.println("2D Array:");
        for (int i = 0; i < table.length; i++) {
            for (int j = 0; j < table[i].length; j++) {
                System.out.print(table[i][j] + " ");
            }
            System.out.println();
        }
    }
}

```

♦ 1.3 Key Points

- Arrays are zero-indexed: the first element is at index 0.
- The length of the array is fixed once declared.
- Use `.length` to find the size of the array.
- Arrays can store primitive types (like `int`, `char`, etc.) and objects (like `String`, `Student`, etc.).

2. Referencing Arrays Dynamically

Arrays in Java are **objects** created at runtime.

Example :

```
int size = 5;  
int[] arr = new int[size]; // dynamic referencing
```

Size can be determined based on user input or logic in the program.

3. Java Strings

a) The Java String Class

Part of `java.lang` package.

Strings are immutable sequences of characters.

```
String s = "Hello";
```

b) Creating & Using String Objects

Using String Literals:

```
String str1 = "Java";
```

Using the new keyword:

```
String str2 = new String("Java");
```

c) Manipulating Strings

Common methods:

- `length()`, `charAt(index)`, `substring()`
- `toUpperCase()`, `toLowerCase()`, `trim()`
- `indexOf()`, `lastIndexOf()`, `replace()`

```
String s = "Java Programming";  
System.out.println(s.substring(5)); // prints "Programming"
```

d) String Immutability & Equality

♦ String Immutability in Java

✓ What is Immutability?

A **String** is **immutable** in Java, which means:

Once a **String** object is created, its value cannot be changed.

✓ Why are Strings immutable?

1. **Security** – Strings are used in sensitive operations (e.g., file paths, network connections).
2. **Thread-safety** – Immutable objects are inherently thread-safe.
3. **Caching** – Java optimizes memory by reusing String objects in the **String pool**.

✓ Example of String Immutability:

```
public class StringImmutability {  
    public static void main(String[] args) {  
        String s1 = "Hello";  
        s1.concat(" World");  
  
        System.out.println(s1); // Output: Hello  
    }  
}
```

Explanation:

- `s1.concat(" World")` creates a **new String object**, but does **not modify** the original `s1`.
- To store the changed value, you must assign it:

```
s1 = s1.concat(" World"); // Now s1 = "Hello World"
```

♦ String Equality in Java

✓ 1. `==` Operator (Reference Comparison)

- Checks if **two string references point to the same object** in memory.

✓ 2. `.equals()` Method (Content Comparison)

- Compares **actual content/values** of the strings.

```
public class StringEquality {
    public static void main(String[] args) {
        String a = "Java";
        String b = "Java";
        String c = new String("Java");

        System.out.println(a == b);      // true (same reference in string pool)
        System.out.println(a == c);      // false (different object in heap)
        System.out.println(a.equals(c)); // true (same content)
    }
}
```

Explanation:

- `"Java"` literals `a` and `b` refer to the same object in the string pool.
- `new String("Java")` creates a new object in the heap memory.
- So:
 - `==` checks reference, and

- `.equals()` checks value.

```
String s1 = "hello";
String s2 = "Hello";

System.out.println(s1.equals(s2));           // false
System.out.println(s1.equalsIgnoreCase(s2)); // true
```

✓ Summary Table:

Comparison Type	Method	Checks What?	Example
Reference	<code>==</code>	Memory location	<code>a == b</code>
Content	<code>.equals()</code>	Character sequence	<code>a.equals(b)</code>
Ignore Case	<code>.equalsIgnoreCase()</code>	Content (case-insensitive)	<code>a.equalsIgnoreCase(b)</code>

4 Passing Strings To & From Methods

✓ Key Concept:

- In Java, strings are objects, but they are passed to methods just like primitive types — using pass-by-value.
- Since String is immutable, changes made to the string inside a method do not affect the original string outside the method.

✓ 1. Passing String to a Method

```

public class StringPassExample {
    // Method that takes a string argument
    static void greet(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public static void main(String[] args) {
        String user = "Alice";
        greet(user); // Output: Hello, Alice!
    }
}

```

✓ 2. Returning a String from a Method

```

public class ReturnStringExample {
    // Method that returns a greeting string
    static String getGreeting(String name) {
        return "Welcome, " + name + "!";
    }

    public static void main(String[] args) {
        String message = getGreeting("Bob");
        System.out.println(message); // Output: Welcome, Bob!
    }
}

```

StringBuffer Class

Unlike `String`, `StringBuffer` is mutable.

Common methods:

- `append()`, `insert()`, `replace()`, `delete()`, `reverse()`

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb); // Hello World
```

4.Simple I/O Using System.out and Scanner Class

- **System.out** is used for output.

```
java

System.out.println("Enter your name:");
```

- **Scanner** is used for input:

```
java

import java.util.Scanner;

Scanner sc = new Scanner(System.in);
String name = sc.nextLine();
System.out.println("Hello, " + name);
```

5.Byte and Character Streams

Java I/O is based on **streams** (sequences of data).

Byte Streams: Handle binary data.

- Classes: `InputStream`, `OutputStream`
- Example:

```
FileInputStream fis = new FileInputStream("data.txt");  
int data = fis.read();  
fis.close();
```

Character Streams: Handle character data.

- Classes: `Reader`, `Writer`
- Example:

```
FileReader fr = new FileReader("data.txt");  
int data = fr.read();  
fr.close();
```

6. Reading/Writing from Console and Files

Console Input/Output

- Use `Scanner` for reading and `System.out` for writing.
- Example:

```
Scanner sc = new Scanner(System.in);  
int age = sc.nextInt();  
System.out.println("Age is: " + age);
```

File Input/Output

- Reading from file:

```
FileReader fr = new FileReader("file.txt");
int ch;
while((ch = fr.read()) != -1) {
    System.out.print((char) ch);
}
fr.close();
```

Writing to file:

```
FileWriter fw = new FileWriter("output.txt");
fw.write("Hello, File!");
fw.close();
```

3. Inheritance, Interfaces, Packages, Enumerations, Autoboxing and Metadata

1. **Inheritance:** A mechanism where one class acquires properties (fields) and behaviors (methods) of another class. Types: Single level, Multilevel. Concepts: Method Overriding, Dynamic Method Dispatch, Abstract Classes.
2. **Interfaces:** A contract in Java where a class agrees to implement the abstract methods declared in the interface.
3. **Packages:** Grouping related classes and interfaces into a single unit. Packages help avoid naming conflicts and manage access protection.

4. **Extending Interfaces and Packages:** Mechanism to create new interfaces or packages by building on existing ones.
5. **Package and Class Visibility:** Controls which classes or members can be accessed from other classes or packages (public, private, protected, default).
6. **Standard Java Packages (util, lang, io, net):** Java's built-in packages that provide commonly used classes.
7. **Wrapper Classes:** Convert primitive data types into objects (e.g., `int` to `Integer`).
8. **Autoboxing/Unboxing:** Automatic conversion between primitive types and their corresponding wrapper classes.
9. **Enumerations (Enums):** Special classes used to define collections of constants.
10. **Metadata (Annotations):** Provide data about the program that is not part of the program itself. Used for compiler instructions, runtime processing, etc.

◆ 2. Interfaces

Definition:

An interface is a blueprint of a class. It contains abstract methods only (Java 8+ allows default/static methods too). A class implements an interface and defines its methods.

There is no concept of multiple-inheritance in Java, but, Interfaces in Java are, for the most part, unique to the language, play a role similar to that of multiple-inheritance. Another unique feature in Java is Packages. Packages are containers for classes that are used to keep the class name space compartmentalized.

```
interface Animal {  
    void sound();  
}  
  
class Dog implements Animal {  
    public void sound() {  
        System.out.println("Dog barks");  
    }  
}
```

◆ 6. Standard Java Packages

Package	Purpose	Example Classes
<code>java.lang</code>	Basic classes (auto-imported)	<code>String</code> , <code>Math</code>
<code>java.util</code>	Data structures and utilities	<code>ArrayList</code> , <code>HashMap</code>
<code>java.io</code>	Input/Output operations	<code>File</code> , <code>BufferedReader</code>
<code>java.net</code>	Networking	<code>URL</code> , <code>Socket</code>

◆ 7. Wrapper Classes

Used to convert primitive data types into objects.

Primitive	Wrapper
<code>int</code>	<code>Integer</code>
<code>char</code>	<code>Character</code>
<code>double</code>	<code>Double</code>

```
java

int a = 10;
Integer obj = Integer.valueOf(a); // Boxing
```

◆ 8. Autoboxing/Unboxing

Autoboxing: Primitive → Object

Unboxing: Object → Primitive

```
java

// Autoboxing
int a = 5;
Integer obj = a; // Automatically boxed

// Unboxing
int b = obj; // Automatically unboxed
```

Exception Handling, Threading, Networking and Database Connectivity

1. Exception Handling

Java handles runtime errors using *exceptions*. This helps in writing robust and error-free programs.

- **Exception Types:** Two main categories — **Checked exceptions** (must be handled at compile time) and **Unchecked exceptions** (occur at runtime).
- **Uncaught Exceptions:** Exceptions not caught using `try-catch` blocks lead to program termination.
- **throw:** Used to explicitly throw an exception object.
- **Built-in Exceptions:** Java provides many predefined exceptions like `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`, etc.
- **Creating Your Own Exceptions:** You can create custom exception classes by extending the `Exception` class.

❖ What is an Exception?

An **exception** is an event that disrupts the normal flow of the program. It is an object that represents an error.

❖ Exception Types:

- **Checked Exceptions:** Must be handled during compilation (e.g., `IOException`, `SQLException`).
- **Unchecked Exceptions:** Occur during runtime (e.g., `ArithmeticException`, `NullPointerException`).

❖ Uncaught Exception:

If not handled using `try-catch`, the program will terminate abnormally.

❖ throw Keyword:

Used to manually throw an exception

```
public class ThrowExample {
    public static void main(String[] args) {
        throw new ArithmeticException("Manually thrown exception");
    }
}
```

❖ Built-in Exceptions:

Some common ones:

- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`
- `NullPointerException`
- `ClassNotFoundException`

❖ Creating Your Own Exception:

You can create a custom exception by extending the `Exception` class.

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}

public class CustomExceptionTest {
    public static void main(String[] args) {
        try {
            throw new MyException("This is a custom exception");
        } catch (MyException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

2. Multi-threading

Multithreading allows concurrent execution of two or more parts of a program for maximum CPU utilization.

- **Thread Class and Runnable Interface:** Two ways to create threads — extend the `Thread` class or implement the `Runnable` interface.
- **Creating Single and Multiple Threads:** Multiple threads can run simultaneously, performing different tasks. You can create multiple threads and run them in parallel.
- **Thread Prioritization:** Threads can be assigned priorities to control execution order using constants like `Thread.MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY`.
- **Synchronization and Communication:** Prevents thread interference and ensures consistent data. Inter-thread communication uses methods like `wait()`, `notify()`, and `notifyAll()`.
- **Suspending/Resuming Threads:** Temporarily pause or resume threads (though `suspend()` and `resume()` are deprecated, alternatives using flags are used).

❖ What is Multi-threading?

It is the ability to run multiple threads (lightweight subprocesses) simultaneously.

❖ Thread Creation:

(a) Extending the Thread class:


```

class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running.");
    }
}

public class Test {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}

```

(b) Implementing Runnable interface:

```

class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread running.");
    }
}

public class Test {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}

```

❖ Thread Prioritization:

Each thread has a priority from 1 (MIN) to 10 (MAX).

```
t1.setPriority(Thread.MIN_PRIORITY);  
t2.setPriority(Thread.MAX_PRIORITY);
```

3. Networking (Using **java.net** package)

Networking in Java allows programs to communicate over a network.

- **java.net Package:** Provides classes like **Socket**, **ServerSocket**, **InetAddress**, and **URL** to perform networking operations.
- **Overview of TCP/IP and Datagram Programming:**
 - **TCP/IP** (Transmission Control Protocol): Provides reliable communication (using **Socket** and **ServerSocket** classes).
 - **Datagram Programming:** Based on UDP (User Datagram Protocol), used for sending and receiving packets using **DatagramSocket** and **DatagramPacket**.

1. IP (Internet Protocol)

❖ What is IP?

IP (Internet Protocol) is a set of rules that govern how data is sent and received over the Internet. It is responsible for **addressing** and **routing** packets of data from the sender to the receiver.

❖ Key Points:

- Every device on the internet has a unique **IP address** (e.g., **192.168.1.1**).
- IP ensures data is delivered to the correct destination.
- IP is a **connectionless protocol**: It does not guarantee delivery, order, or error checking.
- Works with both **TCP** and **UDP**.

2. TCP (Transmission Control Protocol)

❖ What is TCP?

TCP is a **connection-oriented** protocol used for reliable communication between computers over a network.

❖ Features:

- Establishes a **connection** before data is transferred.
- Ensures **reliable** and **ordered** delivery of data.
- Performs **error-checking** and **retransmission** if data is lost.
- Slower than UDP but highly **reliable**.

❖ Common Uses:

- Web browsing (HTTP/HTTPS)
- File transfers (FTP)
- Email (SMTP)



3. UDP (User Datagram Protocol)

❖ What is UDP?

UDP is a **connectionless** protocol used for **fast, lightweight** data transmission. It doesn't guarantee delivery, order, or error-checking.

❖ Features:

- No need to establish a connection.
- **Faster** than TCP.
- May lose or duplicate packets.
- No acknowledgment mechanism.

❖ Common Uses:

- Online gaming
- Video streaming
- Voice over IP (VoIP)

Summary Table

Feature	TCP	UDP
Type	Connection-oriented	Connectionless
Reliability	High (guarantees delivery)	Low (no guarantee)
Speed	Slower	Faster
Error Checking	Yes	Limited
Use Cases	Web, email, FTP	Games, streaming, voice calls
Java Classes	<code>Socket</code> , <code>ServerSocket</code>	<code>DatagramSocket</code> , <code>DatagramPacket</code>

4. Database Connectivity (Using JDBC)

JDBC (Java Database Connectivity) is an API for connecting and executing queries with databases.

- **Accessing and Manipulating Databases using JDBC:**
 - Load the JDBC driver.
 - Establish a connection with the database.
 - Execute SQL queries using `Statement` or `PreparedStatement`.
 - Process the results using `ResultSet`.
 - Close the connection after operations.

❖ What is JDBC?

JDBC (Java Database Connectivity) is an API for connecting Java applications to a database.

❖ Basic Steps to use JDBC:

1. Load the JDBC Driver

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

2.Establish Connection

```
Connection con =  
DriverManager.getConnection("jdbc:mysql://localhost:3306/dbname", "user",  
"password");
```

3.Create Statement and Execute Query

```
Statement stmt = con.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT * FROM students");
```

4.Process Results

```
while (rs.next()) {  
    System.out.println(rs.getString("name"));  
}
```

5.Close the Connection

```
con.close();
```

4. Applets and Event Handling

1. Java Applets

- **Introduction to Applets:**

Applets are small Java programs that run inside a web browser or applet viewer. They are different from standalone Java applications and are typically used for interactive features on web pages.

- **Writing Java Applets:**

Applets are created by extending the `Applet` class or `JApplet` (for Swing-based applets). You override methods like `init()`, `start()`, `paint()`, and `stop()`.

- **Working with Graphics:**

Applets use the `paint(Graphics g)` method to draw shapes, lines, text, etc. You can use the `Graphics` class to create visual content.

- **Incorporating Images & Sounds:**

Applets can display images using `getImage()` and play audio using `AudioClip`. These add multimedia capabilities to applets.

1. Introduction to Applets

What is an Applet?

An **applet** is a **small Java program** designed to be embedded in an HTML page and run inside a **Java-enabled web browser** or an **applet viewer**.

Characteristics:

- No `main()` method
- Runs within a browser using Java Plugin
- Used for **interactive** web components (e.g., animations, forms)
- Limited access to system resources due to security (sandbox model)

2. Writing Java Applets

How to Create an Applet

To create an applet, you need to:

1. Import `java.applet.Applet` and `java.awt.*`
2. Extend the `Applet` or `JApplet` class
3. Override its **lifecycle methods**:
 - `init()` – for initialization
 - `start()` – when applet becomes active
 - `paint(Graphics g)` – to draw content
 - `stop()` – when the applet is inactive
 - `destroy()` – when applet is removed

Example: Simple Applet

```
import java.applet.Applet;
import java.awt.Graphics;

/* <applet code="HelloApplet.class" width="300" height="100"></applet> */

public class HelloApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello from Applet!", 50, 50);
    }
}
```

Note:

To run this, you need to place the `<applet>` tag in an HTML file and open it using an applet viewer or compatible browser.

3. Working with Graphics in Applets

Graphics Class

Applets use the `Graphics` class to draw shapes and text. This is done inside the `paint(Graphics g)` method.

Common Methods:

- `drawString(String, x, y)` – Draws text
- `drawLine(x1, y1, x2, y2)` – Draws a line
- `drawRect(x, y, width, height)` – Draws rectangle
- `drawOval(x, y, width, height)` – Draws oval
- `setColor(Color c)` – Sets drawing color
- `setFont(Font f)` – Sets font for text


```

import java.applet.Applet;
import java.awt.*;

/* <applet code="GraphicsApplet.class" width="300" height="200"></applet> */

public class GraphicsApplet extends Applet {
    public void paint(Graphics g) {
        g.setColor(Color.BLUE);
        g.drawString("Drawing in Applet", 50, 20);

        g.setColor(Color.RED);
        g.drawLine(30, 40, 150, 40);

        g.setColor(Color.GREEN);
        g.drawRect(30, 60, 100, 40);

        g.setColor(Color.MAGENTA);
        g.drawOval(30, 120, 100, 40);
    }
}

```

4. Incorporating Images & Sounds in Applets

📌 Displaying Images

Use the `getImage()` method to load images.

```

Image img;

public void init() {
    img = getImage(getDocumentBase(), "image.jpg");
}

public void paint(Graphics g) {
    g.drawImage(img, 10, 10, this);
}

```

Playing Sounds

Use the `AudioClip` interface to play sounds.

```
import java.applet.*;
import java.awt.*;
import java.net.URL;

public class SoundApplet extends Applet {
    AudioClip clip;

    public void init() {
        clip = getAudioClip(getDocumentBase(), "sound.wav");
    }

    public void start() {
        clip.play(); // You can also use clip.loop() or clip.stop()
    }
}
```

Important:

- `getDocumentBase()` returns the URL of the HTML page that contains the applet.
- Images and sound files must be in the same directory or accessible via URL.

✓ Summary (Key Points for Exam):

Feature	Description
Applet Definition	A mini Java program that runs in a web browser or applet viewer
Lifecycle Methods	<code>init()</code> , <code>start()</code> , <code>paint()</code> , <code>stop()</code> , <code>destroy()</code>
Drawing Graphics	Use <code>Graphics</code> class with <code>paint()</code> method
Adding Multimedia	Use <code>getImage()</code> for images and <code>AudioClip</code> for sounds
Code Placement	Applet code must be embedded in an HTML file using <code><applet></code> tag

2. Event Handling Mechanisms

- Java follows an **event-driven** programming model. Events are actions like button clicks, key presses, mouse movements, etc.

📌 What is Event Handling?

Event handling is the mechanism that controls the event and decides what should happen if an event occurs. Java uses an **event-driven programming model** where user actions like:

- Button clicks
 - Key presses
 - Mouse movements
- generate **events** that the program responds to

3. Listener Interfaces

- Java uses **listeners** to handle events. For each type of event, there's a corresponding **listener interface** (e.g., `ActionListener`, `MouseListener`, `KeyListener`).
- You implement these interfaces and override their methods to respond to events.

Common listener interfaces:

- `ActionListener` – for button clicks
- `MouseListener` – for mouse events
- `KeyListener` – for keyboard events

4. Adapter and Inner Classes

- **Adapter Classes:**
Provide default implementations for listener interfaces with multiple methods. You can extend an adapter class instead of implementing all methods.
Example: `MouseAdapter`, `KeyAdapter`.
- **Inner Classes:**
Classes defined within another class. Useful for handling events because they can access members of the outer class directly.

Adapter Classes

- Used to avoid implementing all methods of a listener interface.
- You can extend an adapter and override only required methods.
- Examples: `MouseAdapter`, `KeyAdapter`

```
addMouseListener(new MouseAdapter() {  
    public void mouseClicked(MouseEvent e) {  
        System.out.println("Mouse Clicked");  
    }  
});
```

Inner Classes

- A class defined within another class.
- Useful for event handling to access outer class variables directly.

```
class Outer {  
    void createButton() {  
        Button b = new Button("Click");  
        b.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) {  
                System.out.println("Button clicked");  
            }  
        });  
    }  
}
```

5. GUI Design and Implementation

- **AWT (Abstract Window Toolkit):**
Older GUI toolkit in Java. Includes basic components like `Label`, `Button`, `TextField`, `Checkbox`, etc.
- **Swing (Java Foundation Classes):**
More advanced GUI toolkit. Components include `JLabel`, `JButton`, `JTextField`, etc. Swing components are more powerful and flexible than AWT.
- **Layout Managers:**
Help in arranging GUI components in containers. Examples: `FlowLayout`, `BorderLayout`, `GridLayout`.
- **Menus:**
Created using `JMenuBar`, `JMenu`, and `JMenuItem` to add menu functionality in GUI applications.
- **Events and Listeners:**
GUI components generate events (like button click), and listeners handle those events.

AWT (Abstract Window Toolkit)

- Older toolkit with components like `Button`, `Label`, `TextField`, `Checkbox`, etc.
- **Example:**

```
Frame f = new Frame("AWT Example");
Button b = new Button("Click");
f.add(b);
f.setSize(200, 200);
f.setLayout(new FlowLayout());
f.setVisible(true);
```

Swing (Java Foundation Classes)

- Newer and more powerful GUI toolkit.
- Components start with **J** (e.g., **JButton**, **JLabel**, **TextField**)

Example: Swing Form

```
import javax.swing.*;

public class SwingForm {
    public static void main(String[] args) {
        JFrame f = new JFrame("Swing Form");
        JButton b = new JButton("Click");
        JTextField tf = new JTextField("Enter text");
        tf.setBounds(50, 50, 150, 20);
        b.setBounds(50, 100, 95, 30);
        f.add(tf);
        f.add(b);
        f.setSize(300, 300);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

Layout Managers

Used to arrange GUI components:

- **FlowLayout**: Left to right, wraps around.
- **BorderLayout**: North, South, East, West, Center.
- **GridLayout**: Grid format.

Menus in Java

- Created using:
 - `JMenuBar`, `JMenu`, and `JMenuItem`.

```
JMenuBar mb = new JMenuBar();
JMenu menu = new JMenu("File");
JMenuItem item = new JMenuItem("Open");
menu.add(item);
mb.add(menu);
frame.setJMenuBar(mb);
```

6. Drawing Graphics

- Use the `Graphics` class to draw shapes like:
 - `drawLine()`, `drawRect()`, `drawOval()` for lines, rectangles, and ovals.
 - `drawString()` for text.
- You can also set different **fonts** and **colors** using methods like `setFont()` and `setColor()`.

7. Overview of Servlets

- **Servlets** are Java programs that run on a server and handle requests from web clients (like browsers).
- They are used to create dynamic web pages and web applications.
- Servlets are part of **Java EE** and work with **HTTP** protocol to respond to web requests.

