

JAVA PROGRAMMING

Chapter 2

Classes, Inheritance, Visibility Control and Arrays

Study Notes for Engineering / Diploma Students

Includes: Detailed Notes • Tables • Code Examples • 15 MCQs with Answer Key • 10 Broad Questions

Contents

2.1 Classes, Objects and Methods

Defining a Class, Creating Objects, Accessing Class Members, Constructors, Method Overloading, Static Members

2.2 Inheritance: Extending a Class

Subclass Constructor, Multilevel and Hierarchical Inheritance, Overriding Methods, final Variables, Methods and Classes, Abstract Methods and Classes

2.3 Visibility Control

Public, Friendly (Default), Protected, Private and Private Protected Access

2.4 Arrays, Strings and Vectors

Arrays, One-Dimensional Arrays, Creating an Array, Two-Dimensional Arrays, Strings, Vectors, Wrapper Classes

Practice Section

15 Multiple Choice Questions, Answer Key, 10 Broad Questions

2.1 Classes, Objects and Methods

A **class** is the foundation of Java: every Java program is written inside a class, and a class defines a new **user-defined data type**. Once a class is defined, any number of **objects** (variables of that type) can be created from it. A class encapsulates *data* (fields/variables) and *code* (methods) that operate on the data, so the members of a class are collectively called its **fields** and **methods**.

2.1.1 Defining a Class

A class is declared using the **class** keyword followed by the class name and a body enclosed in braces. General form:

General form of a class

```
class ClassName {  
    // instance variables (fields)  
    type variable1;  
    type variable2;  
  
    // constructors  
    ClassName(parameters) { ... }  
  
    // methods  
    type methodName(parameters) {  
        // body  
    }  
}
```

Components of a class declaration

Table 2.1.1: Parts of a class declaration

Component	Description
Modifiers	public, abstract, final (optional) - control access and nature of the class
Class name	Must start with a capital letter by convention (e.g. Student)
Superclass	Optional; the parent class preceded by extends
Interfaces	Optional; list of interfaces preceded by implements
Body	Fields, constructors, methods, blocks enclosed in { }

Instance variables and methods

Variables declared inside a class but outside any method are **instance variables**; each object gets its own copy. A **method** is a block of code that performs a task. Its general form is: **modifier returnType methodName(parameter list) { body }**. If a method returns nothing, its return type is **void**; otherwise it must use the **return** statement to send a value back.

Example 2.1: Class definition with methods

```
class Rectangle {
    int length, width;                // instance variables

    void setData(int l, int w) {      // method with parameters, no return value
        length = l;
        width = w;
    }

    int area() {                      // method that returns a value
        return length * width;
    }
}
```

2.1.2 Creating Objects

Creating an object is a two-step process: **declaration** of a reference variable and **instantiation** using the **new** operator, which allocates memory on the heap and returns the reference (address) of the new object.

Example 2.2: Creating objects

```
Rectangle r1;                // step 1: declaration (r1 is null)
r1 = new Rectangle();        // step 2: instantiation (memory allocated)

Rectangle r2 = new Rectangle(); // both steps in one line
Rectangle r3 = r2;            // r3 refers to the SAME object as r2
```

Remember that assigning one reference to another ($r3 = r2$) does **not** create a copy of the object; both variables point to the same object, so a change made through one is visible through the other. Objects that are no longer referenced are removed automatically by the **garbage collector**.

2.1.3 Accessing Class Members

Members of an object are accessed using the **dot (.) operator**: **objectName.variableName** or **objectName.methodName(arguments)**. Static members are accessed with the class name: **ClassName.member**.

Example 2.3: Accessing members using the dot operator

```
public class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.setData(10, 5);                // calling a method
        System.out.println("Area = " + r1.area()); // Area = 50
        r1.length = 20;                  // direct access to variable
        System.out.println(r1.area());    // 100
    }
}
```

2.1.4 Constructors

A **constructor** is a special method that is automatically called when an object is created and is used to **initialise** the object. Rules:

- The constructor name must be **exactly the same as the class name**.
- It has **no return type**, not even void.

- It is invoked implicitly by the **new** operator.
- If the programmer writes no constructor, the compiler supplies a **default (no-argument) constructor**; once any constructor is written, the default one is no longer supplied.
- Constructors can be overloaded, but cannot be static, final or abstract, and are not inherited.

Table 2.1.2: Types of constructors

Type	Description	Example
Default constructor	Takes no arguments; sets default or fixed values	Student() { roll = 0; }
Parameterised constructor	Takes arguments to initialise fields with given values	Student(int r) { roll = r; }
Copy constructor	Takes an object of the same class and copies its data (user-written in Java)	Student(Student s) { roll = s.roll; }

Example 2.4: Constructors

```

class Student {
    int roll;
    String name;

    Student() {                // default constructor
        roll = 0; name = "Unknown";
    }
    Student(int roll, String name) { // parameterised constructor
        this.roll = roll;           // 'this' refers to the current object
        this.name = name;
    }
    Student(Student s) {         // copy constructor
        roll = s.roll; name = s.name;
    }
    void show() { System.out.println(roll + " " + name); }
}

Student s1 = new Student();      // 0 Unknown
Student s2 = new Student(101, "Amit"); // 101 Amit
Student s3 = new Student(s2);    // 101 Amit

```

The this keyword

this is a reference to the current object. It is used (1) to distinguish instance variables from parameters having the same name, (2) to call another constructor of the same class using **this(..)** (constructor chaining, which must be the first statement), and (3) to pass the current object as an argument.

Table 2.1.3: Constructor vs Method

Constructor	Method
Used to initialise an object	Used to perform an operation / expose behaviour
Name is same as class name	Any valid name
No return type	Must have a return type (or void)
Called automatically by new	Called explicitly using an object
Compiler provides a default one	Compiler provides none

2.1.5 Method Overloading

Method overloading means defining two or more methods with the **same name in the same class but with different parameter lists** (different number, type or order of parameters). It is an example of **compile-time polymorphism**: the compiler selects the method by matching the arguments in the call. The **return type alone cannot** distinguish overloaded methods.

Example 2.5: Method overloading

```
class Calc {
    int    add(int a, int b)           { return a + b; }
    int    add(int a, int b, int c)    { return a + b + c; }
    double add(double a, double b)     { return a + b; }
}
Calc c = new Calc();
c.add(2, 3);           // 5      (int, int)
c.add(1, 2, 3);        // 6      (three ints)
c.add(2.5, 1.5);       // 4.0    (double, double)
```

Constructor overloading is the same idea applied to constructors, as seen in Example 2.4. When no exact match is found, Java applies widening conversions (for example int to long or double) to find the closest method.

Table 2.1.4: Overloading vs Overriding

Overloading	Overriding
Same class	Subclass and superclass
Same name, different parameters	Same name and same parameters
Compile-time polymorphism	Run-time polymorphism
Return type may differ	Return type must be same (or covariant)

2.1.6 Static Members

A member declared with the **static** keyword belongs to the **class** rather than to any individual object. It is created once when the class is loaded, shared by all objects, and can be accessed without creating an object.

Table 2.1.5: Kinds of static members

Static member	Details
Static variable	Single copy shared by all objects; used for common data such as a counter or company name
Static method	Can be called using the class name; can access only static members directly; cannot use this or super
Static block	Runs once when the class is loaded, before main(); used to initialise static data

Example 2.6: Static variable, block and method

```
class Counter {
    static int count = 0;           // shared by all objects
    int id;

    static { System.out.println("Class loaded"); } // static block

    Counter() { count++; id = count; }

    static void showCount() {       // static method
        System.out.println("Objects created: " + count);
    }
}

public class Test {
    public static void main(String[] args) {
        new Counter(); new Counter(); new Counter();
        Counter.showCount();        // Objects created: 3
    }
}
```

Why is main() static? The JVM must call main() before any object exists, so it is declared *public static void main(String[] args)*.

Table 2.1.6: Static vs Instance members

Static member	Instance member
Belongs to the class	Belongs to an object
One copy for all objects	Separate copy for each object
Accessed by ClassName.member	Accessed by objectName.member
Memory allocated once at class loading	Memory allocated each time an object is created

2.2 Inheritance: Extending a Class

Inheritance allows a new class to be built from an existing class. The existing class is the **superclass** (base/parent class) and the new class is the **subclass** (derived/child class). The subclass inherits all non-private members of the superclass and may add its own members or redefine inherited methods. The keyword used is **extends**. Java allows a class to extend only one class (single inheritance of classes). All classes implicitly extend the root class **java.lang.Object**.

General form

```
class Superclass {
    // members
}
class Subclass extends Superclass {
    // additional members
}
```

2.2.1 Defining a Subclass

Example 2.7: Extending a class

```
class Employee {
    String name;
    double salary;
    void showSalary() { System.out.println(name + ": " + salary); }
}
class Manager extends Employee {
    int teamSize;                // new member of subclass
    void showTeam() { System.out.println("Team size: " + teamSize); }
}
public class Test {
    public static void main(String[] args) {
        Manager m = new Manager();
        m.name = "Ravi"; m.salary = 60000;    // inherited fields
        m.teamSize = 8;
        m.showSalary();                      // inherited method
        m.showTeam();
    }
}
```

Subclass constructor and the super keyword

When a subclass object is created, the **superclass constructor runs first**, then the subclass constructor. Constructors are **not inherited**. If the superclass has a parameterised constructor, the subclass must call it explicitly using **super(arguments)**, which must be the **first statement** in the subclass constructor. If **super()** is not written, the compiler inserts a call to the superclass's no-argument constructor automatically.

Table 2.2.1: Uses of the super keyword

Use of super	Purpose
super(args)	Calls the superclass constructor
super.variable	Accesses a superclass variable hidden by a subclass variable
super.method()	Calls a superclass method that has been overridden in the subclass

Example 2.8: super() in subclass constructor

```
class Person {
    String name;
    Person(String n) { name = n; System.out.println("Person constructor"); }
}
class Student extends Person {
    int roll;
    Student(String n, int r) {
        super(n);           // must be first statement
        roll = r;
        System.out.println("Student constructor");
    }
}
// new Student("Amit", 5) prints:
//   Person constructor
//   Student constructor
```

2.2.2 Multilevel Inheritance

In **multilevel inheritance** a class is derived from a class that is itself derived from another class, forming a chain: Grandparent to Parent to Child. The last class inherits members of all classes above it.

Example 2.9: Multilevel inheritance

```
class Animal { void eat() { System.out.println("Eating"); } }
class Dog extends Animal { void bark() { System.out.println("Barking"); } }
class Puppy extends Dog { void weep() { System.out.println("Weeping"); } }

Puppy p = new Puppy();
p.eat(); p.bark(); p.weep(); // all three methods available

//   Animal --> Dog --> Puppy
```

2.2.3 Hierarchical Inheritance

In **hierarchical inheritance** several subclasses are derived from one single superclass. They share the common features of the parent but each has its own specific features.

Example 2.10: Hierarchical inheritance

```
class Shape { void info() { System.out.println("I am a shape"); } }
class Circle extends Shape { void area() { System.out.println("pi*r*r"); } }
class Rectangle extends Shape { void area() { System.out.println("l*b"); } }
class Triangle extends Shape { void area() { System.out.println("0.5*b*h"); } }

//           Shape
//          /  |  \
//   Circle Rectangle Triangle
```

Table 2.2.2: Inheritance forms in Java

Form	Structure	Java support
Single	A to B	Yes (classes)
Multilevel	A to B to C	Yes (classes)
Hierarchical	A to B, A to C, A to D	Yes (classes)
Multiple	A and B to C	Not with classes (ambiguity / diamond problem); yes with interfaces

2.2.4 Overriding Methods

When a subclass defines a method with the **same name, same parameter list and same (or covariant) return type** as a method of its superclass, the subclass method **overrides** the superclass method. When called on a subclass object, the overriding version runs. This is **run-time polymorphism**, resolved by **dynamic method dispatch** using the actual object type.

Example 2.11: Method overriding

```
class Bank {
    double rate() { return 5.0; }
}
class SBI extends Bank {
    @Override
    double rate() { return 6.5; }    // overrides
}
class HDFC extends Bank {
    @Override
    double rate() { return 7.0; }
}
Bank b = new SBI();
System.out.println(b.rate());    // 6.5 (decided at run time)
b = new HDFC();
System.out.println(b.rate());    // 7.0
```

Rules for overriding

- Method name and parameters must be identical; the return type must be the same or a subtype.
- The overriding method cannot have a **weaker (more restrictive)** access than the original (a public method cannot be overridden as private).
- **final, static** and **private** methods cannot be overridden.
- The **@Override** annotation is optional but recommended: the compiler reports an error if the method does not really override.
- The overridden version can still be called with **super.methodName()**.

2.2.5 Final Variables, Methods and Classes

The keyword **final** is used to impose restrictions and has three uses.

Table 2.2.3: Uses of final

Use	Effect	Example
final variable	Value can be assigned only once; it becomes a constant	final double PI = 3.14159;
final method	Cannot be overridden by a subclass	final void show() { ... }
final class	Cannot be extended (no subclass possible)	final class Utility { ... }

Example 2.12: final method and final class

```
class Parent {
    final void display() { System.out.println("Cannot override me"); }
}
class Child extends Parent {
    // void display() { }    // ERROR: cannot override final method
}
final class Constants { }
// class Test extends Constants { }    // ERROR: cannot inherit from final class
```

Examples of final classes in the Java library are *String*, *Integer* and the other wrapper classes. A final reference variable cannot point to another object, but the contents of that object may still change.

2.2.6 Abstract Methods and Classes

An **abstract method** is a method that is declared without a body (ending with a semicolon) using the **abstract** keyword. A class containing one or more abstract methods must itself be declared **abstract**. Abstract classes represent general concepts and force subclasses to supply the missing details, which achieves **abstraction**.

- An abstract class **cannot be instantiated** (no objects using new), but reference variables of its type can be created.
- It can contain abstract methods, concrete (normal) methods, variables and constructors.
- A subclass must **override all abstract methods**, otherwise the subclass must also be declared abstract.
- abstract cannot be combined with final, private or static for methods (they would prevent overriding).
- An abstract class may have no abstract methods at all.

Example 2.13: Abstract class

```
abstract class Shape {
    abstract double area();           // no body
    void show() { System.out.println("Area = " + area()); } // concrete method
}
class Circle extends Shape {
    double r;
    Circle(double r) { this.r = r; }
    double area() { return 3.14159 * r * r; }
}
class Square extends Shape {
    double s;
    Square(double s) { this.s = s; }
    double area() { return s * s; }
}
public class Test {
    public static void main(String[] args) {
        // Shape x = new Shape(); // ERROR
        Shape a = new Circle(2);   a.show(); // Area = 12.56636
        Shape b = new Square(3);   b.show(); // Area = 9.0
    }
}
```

Table 2.2.4: Abstract vs Concrete class

Abstract class	Concrete class
Declared with the abstract keyword	Declared normally
Objects cannot be created	Objects can be created
May have abstract and non-abstract methods	All methods must have a body
Meant to be extended	Can be used directly

2.3 Visibility Control

Visibility control (access control) decides which parts of a program may access the members (variables, methods, constructors) of a class. It is applied using **access modifiers**. It supports **encapsulation** by hiding internal details and exposing only what is necessary. A class itself can only be *public* or *default* (no modifier); all four levels apply to members.

2.3.1 The Access Levels

Public access

Members declared **public** are accessible from **anywhere**: any class in any package. Used for the interface that other classes are meant to use, for example the *main()* method.

Friendly access (default / package-private)

Java has no keyword named *friend* (unlike C++). In Java, when **no modifier** is written, the member has **friendly** (default or package-private) access: it is visible to all classes in the **same package** but not to classes outside the package, even subclasses.

Protected access

A **protected** member is accessible within the **same package** and also in **subclasses in other packages** (through inheritance). It is used when members should be available to child classes but hidden from the rest of the world.

Private access

A **private** member is accessible **only within its own class**. It cannot be seen by subclasses or other classes, even in the same package. It gives the highest level of protection and is the basis of data hiding; private data is accessed through public getters and setters.

Private protected access

Early versions of Java (1.0 beta) had a **private protected** modifier, which made a member visible only to subclasses (in any package) and not to other classes of the same package. This combination **was removed in later versions** and is not valid in modern Java; a member declared as *private protected* causes a compile error. Its effect is not directly available now, so the nearest choices are protected or private.

2.3.2 Access Summary

Table 2.3.1: Access levels in Java

Modifier	Same class	Same package	Subclass (other package)	Other package (non-subclass)
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default (friendly)	Yes	Yes	No	No
private	Yes	No	No	No

Table 2.3.2: Where to use each modifier

Modifier	Typical use
public	Methods that form the public API; main() method; constants
protected	Members intended to be reused or overridden by subclasses
default	Helper classes and members used only inside one package
private	Instance variables (data hiding), internal helper methods, singleton constructors

Example 2.14: Effect of access modifiers

```
package pack1;
public class Base {
    public    int a = 1;
    protected int b = 2;
           int c = 3;      // default
    private  int d = 4;
    private void secret() { }
    public int getD() { return d; } // controlled access to private data
}

// In another class of the SAME package : a, b, c accessible; d is not.
// In a subclass of ANOTHER package      : a and b accessible only.
// In an unrelated class of ANOTHER package : only a is accessible.
```

2.4 Arrays, Strings and Vectors

2.4.1 Arrays

An **array** is a fixed-size, ordered collection of elements of the **same data type** stored in contiguous memory locations and referred to by a common name. Each element is accessed by an **index** that starts at **0**. In Java arrays are **objects**, are created dynamically with `new`, and have a built-in **length** property.

Advantages and limitations

- Many values can be stored under one name; elements are accessed quickly by index.
- Easy to traverse using loops and to sort or search.
- Size is **fixed** once created and cannot grow or shrink (Vector or ArrayList is used for that).
- Can hold only one data type.

2.4.2 One-Dimensional Array

A one-dimensional array is a list of elements stored in a single row.

Creating an array

Creating an array has three steps: **declaration**, **creation** (memory allocation) and **initialisation**.

Table 2.4.1: Steps of array creation

Step	Syntax	Example
1. Declaration	<code>type[] arrayName; or type arrayName[];</code>	<code>int[] marks;</code>
2. Creation	<code>arrayName = new type[size];</code>	<code>marks = new int[5];</code>
3. Initialisation	<code>arrayName[index] = value;</code>	<code>marks[0] = 78;</code>
Combined	<code>type[] a = new type[size];</code>	<code>int[] marks = new int[5];</code>
With values	<code>type[] a = {v1, v2, ...};</code>	<code>int[] marks = {78, 65, 90};</code>

When an array is created with `new`, all elements receive **default values** (0 for numeric types, false for boolean, '\u0000' for char and null for objects). The **length** (a property, not a method) gives the number of elements, so valid indices are **0 to length-1**; any other index throws *ArrayIndexOutOfBoundsException* at run time.

Example 2.15: Traversing a one-dimensional array

```
int[] a = {45, 12, 89, 33, 70};
int sum = 0, max = a[0];

for (int i = 0; i < a.length; i++) {
    sum += a[i];
    if (a[i] > max) max = a[i];
}
System.out.println("Sum = " + sum);           // 249
System.out.println("Max = " + max);           // 89

for (int x : a)                               // enhanced for loop
    System.out.print(x + " ");
```

Example 2.16: Sorting an array

```
int[] a = {50, 20, 40, 10, 30};
for (int i = 0; i < a.length - 1; i++) {           // bubble sort
    for (int j = 0; j < a.length - 1 - i; j++) {
        if (a[j] > a[j + 1]) {
            int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
        }
    }
}
// a is now {10, 20, 30, 40, 50}
```

2.4.3 Two-Dimensional Array

A **two-dimensional array** stores data in rows and columns (a table or matrix). It is actually an **array of arrays**. An element is accessed using two indices: **a[row][column]**.

Table 2.4.2: Two-dimensional array basics

Task	Syntax / Example
Declaration	int[][] m; or int m[][];
Creation	m = new int[3][4]; (3 rows, 4 columns)
Initialisation	int[][] m = { { 1, 2, 3 }, { 4, 5, 6 } };
Access element	m[1][2] = 60; // row 1, column 2
Rows / columns	m.length gives rows; m[0].length gives columns

Example 2.17: Matrix addition

```
int[][] a = { { 1, 2, 3 }, { 4, 5, 6 } };
int[][] b = { { 7, 8, 9 }, { 1, 2, 3 } };
int[][] c = new int[2][3];

for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        c[i][j] = a[i][j] + b[i][j];           // matrix addition
        System.out.print(c[i][j] + " ");
    }
    System.out.println();
}
// Output:
// 8 10 12
// 5 7 9
```

Jagged arrays: since each row is a separate array, rows may have different lengths, for example `int[][] j = new int[3][]; j[0] = new int[2]; j[1] = new int[4];`. Arrays can be passed to and returned from methods; they are passed by reference, so a method can modify the original array.

Table 2.4.3: 1-D vs 2-D arrays

One-dimensional	Two-dimensional
One index: a[i]	Two indices: a[i][j]
Linear list	Table of rows and columns
new int[5]	new int[3][4]
One loop to traverse	Nested loops to traverse

2.4.4 Strings

A **string** is a sequence of characters. Unlike C, Java strings are **objects** of the **String** class (package java.lang), not character arrays. String objects are **immutable**: once created, their content cannot be changed; operations such as concatenation produce a new String object. Strings may be created in two ways:

Creating strings

```
String s1 = "Java";           // string literal (stored in string pool)
String s2 = new String("Java"); // using new (separate object in heap)
char[] ch = {'J', 'a', 'v', 'a'};
String s3 = new String(ch);    // from a char array
```

Common methods of the String class (let s = "Hello World"):

Table 2.4.4: Important String methods

Method	Purpose	Example / Result
length()	Number of characters	s.length() gives 11
charAt(i)	Character at index i	s.charAt(1) gives 'e'
substring(b, e)	Part of string from b to e-1	s.substring(0, 5) gives "Hello"
toUpperCase() / toLowerCase()	Changes case	s.toUpperCase() gives "HELLO WORLD"
concat(str) or +	Joins strings	"Hi ".concat("All") gives "Hi All"
equals(str)	Compares content (case sensitive)	s.equals("hello world") gives false
equalsIgnoreCase(str)	Compares ignoring case	s.equalsIgnoreCase("hello world") gives true
compareTo(str)	Lexicographic comparison; 0 if equal	"a".compareTo("b") gives negative value
indexOf(ch)	First position of character / substring, -1 if absent	s.indexOf('o') gives 4
replace(a, b)	Replaces characters	s.replace('l', 'L') gives "HeLLo WorLd"
trim()	Removes leading and trailing spaces	" hi ".trim() gives "hi"
startsWith(str) / endsWith(str)	Tests the beginning / end	s.startsWith("He") gives true

== versus equals()

The == operator compares **references** (whether both variables refer to the same object), whereas **equals()** compares the **contents**. Thus `new String("Java") == "Java"` is false but `new String("Java").equals("Java")` is true.

StringBuffer and StringBuilder

Because String is immutable, repeated modification creates many objects. **StringBuffer** (thread-safe) and **StringBuilder** (faster, not thread-safe) are **mutable** classes that allow changes in the same object using methods such as `append()`, `insert()`, `reverse()`, `delete()` and `replace()`.

Table 2.4.5: String vs StringBuffer vs StringBuilder

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread safety	Safe (immutable)	Synchronised, safe	Not synchronised
Speed	Slow for many changes	Moderate	Fastest
Example	<code>s = s + "a";</code>	<code>sb.append("a");</code>	<code>sb.append("a");</code>

Example 2.18: StringBuffer

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");           // Hello World
sb.insert(5, ",");             // Hello, World
sb.reverse();                  // dlrow ,olleH
System.out.println(sb);
```

Example 2.19: Palindrome check

```
String s = "madam";
String rev = "";
for (int i = s.length() - 1; i >= 0; i--)
    rev = rev + s.charAt(i);
if (s.equals(rev)) System.out.println("Palindrome");
else System.out.println("Not a palindrome");
```

2.4.5 Vectors

An array has a fixed size. The **Vector** class (package **java.util**) implements a **dynamic (growable) array** that can increase or decrease its size automatically as elements are added or removed. A vector stores **objects** (not primitive types directly), can hold elements of different types, and its methods are **synchronised** (thread-safe). If the capacity is full, it grows automatically (by default it doubles).

Constructors

- **Vector()** - creates an empty vector with default capacity 10.
- **Vector(int capacity)** - creates a vector with the given initial capacity.
- **Vector(int capacity, int increment)** - also specifies how much it grows each time.

Table 2.4.6: Important Vector methods

Method	Purpose
addElement(obj) / add(obj)	Adds an element at the end
insertElementAt(obj, index)	Inserts an element at a given position
elementAt(index) / get(index)	Returns the element at the index
firstElement() / lastElement()	Returns first / last element
removeElement(obj)	Removes the first occurrence of the object
removeElementAt(index)	Removes the element at the index
removeAllElements() / clear()	Removes all elements
size()	Number of elements currently stored
capacity()	Current capacity of the vector
contains(obj)	Checks whether the object is present
indexOf(obj)	Position of the object, -1 if absent
copyInto(array)	Copies the elements into an array

Example 2.20: Using a Vector

```
import java.util.Vector;

public class VectorDemo {
    public static void main(String[] args) {
        Vector v = new Vector();
        v.addElement("Java");
        v.addElement(10);           // int is boxed to Integer
        v.addElement(3.5);
        v.insertElementAt("C++", 1);

        System.out.println("Size = " + v.size());    // 4
        for (int i = 0; i < v.size(); i++)
            System.out.println(v.elementAt(i));
        v.removeElement(10);
        System.out.println(v);           // [Java, C++, 3.5]
    }
}
```

Table 2.4.7: Array vs Vector

Array	Vector
Fixed size	Dynamic size
Can store primitives and objects	Stores only objects (primitives are converted using wrapper classes)
Elements of one type	Can store objects of different types
Faster, no synchronisation	Slightly slower, synchronised
Accessed using [] and length	Accessed using methods and size()
Part of the language	Part of java.util package

2.4.6 Wrapper Classes

Vectors and other collections can hold only objects, so primitive values must be converted into objects. Java provides a **wrapper class** for each primitive type in **java.lang**. A wrapper class *wraps* a primitive value inside an object. Wrapper objects are immutable.

Table 2.4.8: Primitive types and their wrapper classes

Primitive	Wrapper class	Primitive	Wrapper class
byte	Byte	float	Float
short	Short	double	Double
int	Integer	char	Character
long	Long	boolean	Boolean

Table 2.4.9: Common conversions using wrapper classes

Conversion	Method	Example
Primitive to object (boxing)	Constructor / valueOf()	Integer i = Integer.valueOf(25);
Object to primitive (unboxing)	intValue(), doubleValue(), etc.	int n = i.intValue();
String to primitive	parseInt(), parseDouble(), etc.	int x = Integer.parseInt("123");
Primitive to String	toString() / String.valueOf()	String s = Integer.toString(45);
Object to String	toString()	i.toString()

Autoboxing and unboxing: from Java 5 the compiler converts automatically between a primitive and its wrapper: *Integer a = 10;* (autoboxing) and *int b = a;* (unboxing). Useful constants and methods include *Integer.MAX_VALUE*, *Integer.MIN_VALUE*, *Character.isDigit(ch)*, *Character.isLetter(ch)* and *Boolean.parseBoolean(str)*.

Example 2.21: Wrapper classes

```
Integer a = 100;           // autoboxing
int b = a + 50;            // unboxing, b = 150
int n = Integer.parseInt("245"); // String to int
double d = Double.parseDouble("3.75"); // String to double
String s = Integer.toString(99); // int to String
Vector v = new Vector();
v.addElement(new Integer(5)); // storing primitive as object
v.addElement(7);             // autoboxing does it for us
```

Practice Section

A. Multiple Choice Questions (15 Marks)

Choose the correct option for each question.

Q1. Which keyword is used to create an object in Java?

- (a) create (b) new (c) object (d) alloc

Q2. A constructor in Java:

- (a) Has the return type void (b) Must be declared static (c) Has the same name as its class (d) Can have any name

Q3. Method overloading is possible when methods have:

- (a) Different return types only (b) The same name and different parameter lists (c) Different names (d) Different access modifiers only

Q4. Static members of a class belong to:

- (a) Each object separately (b) The method (c) The class (d) The package only

Q5. Which keyword is used to inherit a class in Java?

- (a) implements (b) inherits (c) super (d) extends

Q6. Java does not support multiple inheritance through:

- (a) Interfaces (b) Packages (c) Classes (d) Abstract methods

Q7. In a subclass constructor, super() must be:

- (a) The last statement (b) The first statement (c) Inside a loop (d) Written only in main()

Q8. A method declared final cannot be:

- (a) Called (b) Inherited (c) Overridden (d) Overloaded

Q9. Which statement about an abstract class is TRUE?

- (a) It must have only abstract methods (b) Objects can be created using new (c) It cannot be instantiated (d) It cannot have constructors

Q10. A member with no access modifier (default) is visible:

- (a) Everywhere (b) Only in its own class (c) Within the same package (d) In all subclasses everywhere

Q11. Which is the most restrictive access modifier?

- (a) public (b) protected (c) default (d) private

Q12. The default value of elements of an int array created with new int[3] is:

- (a) null (b) garbage (c) 0 (d) -1

Q13. If int[][] m = new int[3][4]; then m[0].length is:

- (a) 3 (b) 4 (c) 7 (d) 12

Q14. Which class represents a dynamic, growable array in java.util?

- (a) Array (b) String (c) Vector (d) Object

Q15. Which method converts the String "123" into the int value 123?

- (a) Integer.toString() (b) Integer.parseInt() (c) String.valueOf() (d) Integer.intValue()

B. Answer Key for MCQs

Table 2.4.10: Answer key

Q.No.	Answer	Explanation
1	(b)	new allocates memory and calls the constructor
2	(c)	A constructor has the class name and no return type
3	(b)	Parameter lists must differ; return type alone is not enough
4	(c)	One shared copy exists for the class
5	(d)	A subclass uses extends to inherit a class
6	(c)	Classes may extend only one class; interfaces can be multiple
7	(b)	super() or this() must be the first statement
8	(c)	final methods cannot be overridden
9	(c)	Abstract classes cannot be instantiated directly
10	(c)	Default (friendly) access is limited to the package
11	(d)	private allows access only inside the same class
12	(c)	Numeric array elements default to 0
13	(b)	m.length is rows (3); m[0].length is columns (4)
14	(c)	Vector grows automatically as elements are added
15	(b)	parseInt converts a String into an int

C. Broad Questions (10 Questions)

Answer the following questions in detail.

- 1 What is a class? Explain how a class is defined and how objects are created and members accessed, with a suitable Java program.
- 2 What is a constructor? Explain default, parameterised and copy constructors with examples. Differentiate between a constructor and a method.
- 3 Explain method overloading and constructor overloading with programs. Distinguish between overloading and overriding.
- 4 Explain static variables, static methods and static blocks with an example. Why is the main() method declared static?
- 5 What is inheritance? Explain single, multilevel and hierarchical inheritance with programs. Explain the use of the super keyword and subclass constructors.
- 6 What is method overriding? State its rules and explain run-time polymorphism with an example.
- 7 Explain the uses of the final keyword (variable, method and class) with examples.
- 8 What are abstract classes and abstract methods? Write a program to demonstrate an abstract class. Differentiate between abstract and concrete classes.
- 9 Explain visibility control in Java with the help of public, friendly (default), protected and private access. Draw the access summary table and comment on private protected access.

- 10 Explain one-dimensional and two-dimensional arrays with syntax and programs (sorting and matrix addition). Also explain Strings, Vectors and Wrapper classes with their important methods.

— End of Chapter 2 —

Poly Notes Hub