ALLAMA IQBAL OPEN UNIVERSITY

USMAN NOTES
www.usmannotes.com

Name:

Registration No.:

Course Code: 3582

Program: PGD

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Internet Programming Language (3582) • Level: PGD

Semester: Autumn 2026

Assignment No. 1

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

Note: This guide is prepared for study and reference purposes only. Students are advised to read the relevant parts of the course book, understand the concepts, compile and run every Java program given here on their own computer, and then write the assignment in their own words. Copying this guide word for word is not allowed and will not build the programming skill this course is designed to develop. No units are printed in the question booklet for this course, so none are shown on this title page.

USMAN NOTES www.usmannotes.com

Question 1: Class Variables, Instance Variables and Local Variables (20 Marks)

Question as printed: Explain the difference between class variables, instance variables, and local variables. Write a Java program demonstrating all three. (20)

Introduction

Every Java program stores its data in variables, but not all variables behave in the same way. Where a variable is declared decides three things that a programmer must understand before writing reliable code: how long the variable lives, how much memory it occupies and where

that memory is placed, and from which parts of the program it can be reached. Java recognises three principal kinds of variable according to the place of declaration. A class variable belongs to the class itself and is shared by every object. An instance variable belongs to one particular object and gives that object its own separate state. A local variable belongs to a single execution of a method or block and disappears as soon as that execution ends. Confusing these three kinds is one of the most common causes of beginner errors, such as expecting a value to be remembered when it has already been destroyed, or accidentally sharing a value among all objects when each object needed its own copy. This answer defines each kind, compares them point by point, and then demonstrates all three together in one complete, runnable program whose output is explained line by line.

Class Variables: One Copy for the Whole Class

A class variable is declared inside a class but outside every method, and it is marked with the keyword static. Because it belongs to the class rather than to any object, only one copy of it ever exists, however many objects are created — even if no object is created at all. It is loaded and initialised when the class itself is loaded into memory, before any object exists, and it survives for the whole life of the program. A class variable is reached through the class name, for example Student.college, although Java also permits access through an object reference, which is discouraged because it hides the shared nature of the variable. If any object changes a class variable, every other object immediately sees the new value, because all of them are looking at the same single storage location. Class variables are therefore the correct tool for facts that are genuinely common to all objects of a class: the name of the institution to which all students belong, a common interest rate shared by all bank accounts, a count of how many

objects have been created, or a constant-like configuration value. A class variable that is also marked final becomes a true constant whose value can never change after initialisation.

Instance Variables: One Copy for Each Object

An instance variable is also declared inside the class and outside every method, but it is declared without the keyword static. A fresh copy is created inside every object when that object is constructed with new, and each copy holds an independent value. One student object can store the name Ali and roll number 101 while a second stores Sara and 102, and changing one object never disturbs the other. Instance variables are created when the object is created and are destroyed, together with the object, when the object becomes unreachable and is collected by the garbage collector. If the programmer does not initialise an instance variable, Java assigns it a default value automatically: zero for numeric types, false for boolean, and null for references. Instance variables are reached through an object reference, for example s1.name and s2.name, and they are the normal way of representing the state or attributes of an object. Together, the instance variables of an object describe what that object is, while its methods describe what it does.

Local Variables: Temporary Working Storage

A local variable is declared inside a method, a constructor, or a smaller block such as the body of an if statement or a for loop. It exists only while that method or block is executing. It is created on entry, normally stored on the stack, and it ceases to exist the moment execution leaves the block, so it can never be used from another method. Its scope, the region of code in which its name is meaningful, is exactly the block in which it is declared. Unlike class and

instance variables, a local variable receives no default value at all. The Java compiler refuses to compile a program that reads a local variable before a value has definitely been assigned to it, which protects the programmer from using meaningless garbage data. Local variables are used for temporary working storage: loop counters, intermediate results of a calculation, values read from the user during one operation, and references that are needed only while a single task is performed. Because each call of a method receives its own fresh set of local variables, local variables are also what make recursion and multithreaded calls of the same method by different threads safe, since no thread can overwrite another thread's locals.

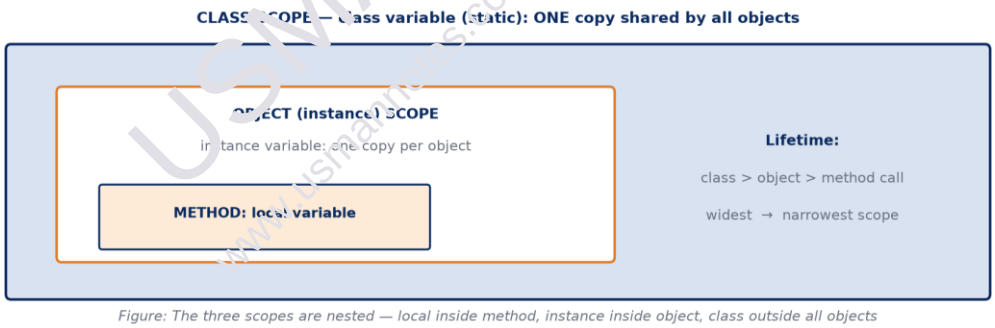


Figure: The three scopes are nested — local inside method, instance inside object, class outside all objects

Figure 1: The three scopes nested inside one another. The class variable is widest and longest-lived; the local variable is narrowest and shortest-lived.

Table 1: Class variables, instance variables and local variables compared

Feature	Class variable (static)	Instance variable	Local variable
Declared where	In the class, outside methods, with static	In the class, outside methods, without static	Inside a method, constructor or block
Belongs to	The class itself	One particular object	One execution of the method or block

Number of copies	Exactly one, shared by all objects	One separate copy per object	One copy per method call
Created when	When the class is loaded	When the object is created with new	When the method or block is entered
Destroyed when	When the class is unloaded / the program ends	When the object is garbage collected	When the method or block finishes
Default value	Yes — 0, false or null	Yes — 0, false or null	No — must be assigned before use
How accessed	ClassName.variable (preferred)	objectReference.variable	By name only, inside its block
Typical use	Shared data, counters, constants	State and attributes of an object	Temporary calculations, loop counters

USMAN NOTES www.usmannotes.com

Java Program Demonstrating All Three

The following program uses a Student class. The variable college is a class variable, shared by every student. The variables name and rollNo are instance variables, so each student keeps his or her own values. Inside the method addBonus a local variable called bonus is created, used for a short calculation, and destroyed when the method returns. The main method creates two students, changes the class variable once, and shows that both objects see the change, while their instance variables remain completely independent.

```
class Student {
```

```
    // CLASS VARIABLE: one copy, shared by all objects
    static String college = "Allama Iqbal Open University";

    // INSTANCE VARIABLES: separate copy in each object
    String name;
    int rollNo;

    Student(String name, int rollNo) {
        this.name = name;
        this.rollNo = rollNo;
    }

    void addBonus() {
        // LOCAL VARIABLE: exists only inside this method
        int bonus = 5;
        int total = rollNo + bonus;   // temporary working value
        System.out.println(name + ": roll " + rollNo
            + " + local bonus " + bonus + " = " + total);
        // 'bonus' and 'total' are destroyed here, on return
    }

    void display() {
        System.out.println(rollNo + "  " + name + "  " + college);
    }
}
```

USMAN NOTES www.usmannotes.com

```
public class VariableDemo {
    public static void main(String[] args) {
        Student s1 = new Student("Ali", 101);
        Student s2 = new Student("Sara", 102);

        s1.display();   // uses s1's own instance variables
        s2.display();   // uses s2's own instance variables
        s1.addBonus();  // local variable in action

        // Change the CLASS variable once, through the class name:
        Student.college = "AIOU, Islamabad";

        System.out.println("--- After changing the class variable ---
");
        s1.display();   // BOTH objects show the new value,
        s2.display();   // because there is only ONE copy
    }
}
```

Output of the program:

```
101  Ali  Allama Iqbal Open University
102  Sara  Allama Iqbal Open University
Ali: roll 101 + local bonus 5 = 106
--- After changing the class variable ---
101  Ali  AIOU, Islamabad
102  Sara  AIOU, Islamabad
```

Explanation of the Output

The output proves each claim made above. The first two lines show that s1 and s2 hold different names and roll numbers at the same time, which is possible only because instance variables are copied into each object separately. The third line shows the local variable bonus taking part in a calculation inside addBonus; if a second call of addBonus were made, a brand-new bonus would be created again with the value 5, because nothing of the previous call survives. The decisive demonstration comes after the single statement Student.college = "AIOU, Islamabad". No statement touches s1 or s2 individually, yet both display lines change together. That can happen only because college is not stored inside either object at all: both objects read the one class-level copy. A student should also note what the compiler forbids. Writing System.out.println(bonus) inside display or main would be a compile-time error, because a local variable has no existence outside its own method, and trying to print an uninitialised local variable is likewise rejected, whereas an uninitialised instance or class variable would simply print its default value.

Choosing the right kind is a design decision with practical consequences. Placing data in a class variable when each object needs its own value produces the classic bug in which every object appears to hold the last value assigned, because all assignments went to the same shared location. Placing shared data in instance variables wastes memory and invites inconsistency, because updating one copy leaves the others stale — if the college name changed, a hundred student objects would each need to be corrected. Local variables should be preferred whenever

a value is needed only during one operation, because they use memory briefly, cannot be corrupted by other objects or threads, and make a method easier to understand and test. A useful rule is to give every variable the narrowest scope and shortest lifetime that still allows the program to work correctly.

Conclusion

To conclude, class, instance and local variables differ in owner, number of copies, lifetime, default value and scope. A class variable is declared with static, exists once for the whole class, is shared by all objects and lives for the whole program. An instance variable exists once per object, gives each object its own independent state, receives a default value, and dies with its object. A local variable exists only inside one execution of a method or block, receives no default value, must be assigned before use, and is destroyed on return. The demonstration program shows all three working together: shared college data in a class variable, personal name and roll data in instance variables, and a short-lived bonus in a local variable. Understanding this distinction is foundational, because almost every later topic in Java — inheritance, static methods, multithreading and collections — assumes that the programmer knows exactly where a variable lives and who can change it.

USMAN NOTES www.usmannotes.com

Question 2: Single Inheritance — Child Class Adding New Methods (20 Marks)

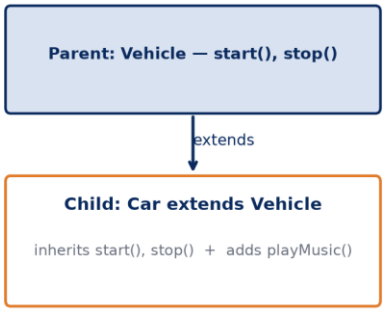
Question as printed: Write a Java program to demonstrate single inheritance where the child class adds new methods to the parent class. Explain how method access works in inheritance. (20)

Introduction

Inheritance is the object-oriented mechanism by which a new class is built from an existing class, acquiring its fields and methods and then adding or refining behaviour of its own. It expresses an is-a relationship: a car is a vehicle, a manager is an employee, a savings account is a bank account. Instead of copying the common code into every new class, the programmer writes it once in a parent class, also called a superclass or base class, and every child class, also called a subclass or derived class, inherits it automatically. This reuse shortens development, reduces duplication and, most importantly, means that a correction made in the parent is inherited by all children at once. Java supports several patterns of inheritance through interfaces, but for classes it permits only single inheritance: a class may extend exactly one direct parent. This answer explains single inheritance, presents a complete program in which a child class adds a genuinely new method that the parent does not have, and then explains carefully how method access works — which methods a child object can call, which members are not inherited, and how an overridden method is selected at runtime.

Single Inheritance and the extends Keyword

In single inheritance one child class extends one parent class using the keyword extends, as in class Car extends Vehicle. The child automatically receives every accessible field and method of the parent. It may then do three things: use the inherited members exactly as they are, override an inherited method by redefining it with the same signature to give more specialised behaviour, and add entirely new fields and methods that exist only in the child. The question specifically requires the third ability — adding new methods — because it shows that inheritance is not merely copying: the child is a genuine extension of the parent, more capable than the parent alone. Every Java class ultimately descends from the class Object, so even a class that extends nothing directly still inherits methods such as toString and equals from Object. Constructors, however, are never inherited, although the child's constructor must call a parent constructor, using super(), so that the parent part of the object is properly initialised before the child's own initialisation runs.



Single inheritance: one child, one direct parent — the child reuses and extends the parent

Figure 1: Single inheritance. Car extends Vehicle, inherits start() and stop(), and adds its own new method playMusic().

Java Program: Vehicle and Car

In the program below, the parent class Vehicle provides two general methods, start and stop, which any vehicle can perform. The child class Car inherits both of them unchanged and adds a new method, playMusic, which makes sense only for a car. The main method creates a Car object and calls all three methods through that one object, demonstrating inherited access and the new child-only behaviour side by side. It also shows the one-way nature of access: a Vehicle reference cannot call playMusic, because the parent knows nothing about methods invented by its child.

```
// PARENT CLASS (superclass)
class Vehicle {
    String brand = "Generic Vehicle";

    void start() {
        System.out.println("Vehicle starts with the key / button.");
    }

    void stop() {
        System.out.println("Vehicle stops when brakes are applied.");
    }
}

// CHILD CLASS (subclass): single inheritance with 'extends'
class Car extends Vehicle {
    int doors = 4;
```

```
    // NEW method added by the child — Vehicle has no such method
    void playMusic() {
        System.out.println("Car plays music on its audio system.");
    }

    // NEW method that uses an inherited field
    void showDetails() {
        System.out.println("Brand: " + brand + ", Doors: " + doors);
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        Car myCar = new Car();

        myCar.start();       // inherited from Vehicle
        myCar.playMusic();   // new method of Car itself
        myCar.showDetails(); // new method of Car
        myCar.stop();        // inherited from Vehicle

        // A parent reference can hold a child object,
        // but can call only methods known to the parent:
        Vehicle v = new Car();
        v.start();           // OK — start() is in Vehicle
```

USMAN NOTES www.usmannotes.com

```
        // v.playMusic();    // ERROR — Vehicle does not know
playMusic()
    }
}
```

Output of the program:

```
Vehicle starts with the key / button.
Car plays music on its audio system.
Brand: Generic Vehicle, Doors: 4
Vehicle stops when brakes are applied.
Vehicle starts with the key / button.
```

USMAN NOTES www.usmannotes.com

How Method Access Works in Inheritance

Method access in inheritance follows a small set of precise rules. First, a child object can call every public and protected method of its parent as if the method were its own, because those methods genuinely become members of the child. This is why myCar.start() compiles and runs even though the text of start appears only inside Vehicle. Second, access is controlled by the access modifiers. A private method of the parent is not inherited in any usable sense: it remains part of the parent object's internal working, and the child can reach that behaviour only indirectly through a public or protected parent method that calls it. A method with default, package-private access is inherited only if the child is in the same package. Third, the child may add new methods freely, as Car adds playMusic and showDetails, but those methods belong to the child alone. They can be called through a Car reference, never through a Vehicle reference,

and never on a plain Vehicle object, because method availability at compile time is decided by the type of the reference used to make the call.

Fourth, when the child redefines an inherited method with the same name and parameters, the new definition is said to override the old one. At runtime the Java Virtual Machine applies dynamic method dispatch: it looks at the actual object, not merely at the reference type, and runs the child's version. Thus if Car overrode start, then v.start() in the program above would run Car's version even though v is declared as a Vehicle, because v actually holds a Car object. This is the mechanism behind runtime polymorphism. Fifth, an inherited method that is not overridden simply runs the parent's code, but that code still operates on the child object, so any instance fields it reads are the child's fields. Sixth, a method marked final in the parent can be inherited and called but cannot be overridden, which lets a designer guarantee that a critical behaviour never changes. Seventh, static methods behave differently: they are inherited in the sense that they can be called through the child class name, but if the child defines a static method with the same signature it hides rather than overrides the parent method, and the version chosen depends on the reference type, a distinction examined in detail in the next question. Finally, inside a child method the keyword super can be used to call the parent's version of an overridden method or a parent constructor, which is useful when the child wants to extend, rather than completely replace, the parent's behaviour.

Table 1: What a child class can and cannot access from its parent

Parent member	Inherited / accessible in child?	Reason or condition
public method	Yes, freely	Public members are accessible

		everywhere
protected method	Yes	Protected is designed for inheritance: accessible in subclasses
default (package) method	Only in the same package	Package-private access stops at the package boundary
private method	No direct access	Private members belong to the parent alone; use them via a public parent method
Constructor	Not inherited	Each class defines its own; child calls parent constructor with super()
final method	Inherited but cannot be overridden	Final locks the behaviour
static method	Accessible, but hidden — not overridden — if redefined	Static methods belong to the class, not the object

A practical consequence follows from these rules: inheritance should be used only where a true is-a relationship exists and the child genuinely needs the parent's behaviour. Because the child is tightly coupled to the parent, a careless change in the parent can break every child — the so-called fragile base class problem. Good designers therefore keep parent classes stable, mark methods final where change would be dangerous, prefer protected over public for members intended mainly for children, and consider composition, holding a reference to another object,

when the relationship is has-a rather than is-a. In the demonstration program the relationship is sound: a car truly is a vehicle, it honestly reuses starting and stopping, and its new methods add car-specific capability without duplicating any vehicle code.

Conclusion

To conclude, single inheritance in Java lets one child class extend one parent class with the keyword extends, inheriting all accessible fields and methods while remaining free to add new methods of its own, as Car adds playMusic and showDetails to the inherited start and stop. Method access is governed by access modifiers — public and protected methods are inherited and callable, private methods are not directly accessible, and constructors are never inherited — and by the reference type at compile time, which decides which methods may be called at all. Where the child overrides a method, dynamic dispatch selects the child's version at runtime according to the actual object. Used with a genuine is-a relationship, inheritance removes duplication, centralises corrections in the parent, and lets programmers build specialised classes from reliable general ones, which is why it remains one of the central pillars of object-oriented programming in Java.

Question 3: Method Hiding in Java — Static Methods in Parent and Child (20 Marks)

Question as printed: What is method hiding in Java? Write a Java program to demonstrate the use of static methods in parent and child classes. (20)

Introduction

Java treats instance methods and static methods according to two entirely different rules, and method hiding is the name given to the static rule. When a child class defines an instance method with the same signature as a parent instance method, the child method overrides the parent method, and the version that runs is chosen at runtime from the actual object. When, however, a child class defines a static method with the same signature as a static method in the parent, no overriding takes place at all. Instead, the child's method hides the parent's method, and the version that runs is chosen at compile time from the type of the reference through which the call is made. The distinction sounds small but produces output that surprises many students: the same object called through a parent reference, runs the parent's static method, while called through a child reference it runs the child's static method. This answer defines method hiding precisely, contrasts it with overriding in a table, and demonstrates it with a complete program using static methods in a parent and a child class, explaining every line of the output.

What Is Method Hiding?

Method hiding occurs when a subclass defines a static method whose signature — name plus parameter types — is identical to a static method in its superclass. The subclass method does not replace the superclass method in any object; both methods continue to exist, each attached to its own class. The subclass method merely hides the superclass method for code that refers to the subclass, in exactly the same way that a field declared in a subclass hides a field of the same name in the superclass. The compiler resolves a static method call by looking at the

declared type of the reference, or at the class name written before the dot, and binds the call to that class's method once and for all. No runtime search of the object's real class takes place. For this reason static methods are said to be bound statically, or early, whereas overridden instance methods are bound dynamically, or late. It follows that calling a static method through an object reference is misleading style: the object contributes nothing to the choice, and good practice is always to call static methods through the class name, Parent.display() or Child.display(), so that the reader can see exactly which method will run.

Several related rules complete the definition. A static method cannot override an instance method, and an instance method cannot override a static method; attempting either produces a compile-time error, because the two kinds of method belong to different worlds — one to the class, one to the object. The @Override annotation, which the compiler checks, may be placed on a genuine instance-method override but must not be placed on a hiding static method, and the compiler will reject it there, which is a useful test of understanding. A static method also cannot use the keywords this or super, because both refer to an object and a static method runs without any particular object. Finally, hiding respects access rules in the same way as overriding: the hiding method cannot reduce the visibility of the method it hides, so a public static method in the parent cannot be hidden by a private or default-access static method in the child.

Table 1: Method overriding versus method hiding

Feature	Method overriding	Method hiding
Applies to	Instance methods	Static methods

Where defined	Child redefines a parent instance method	Child redefines a parent static method with the same signature
Binding time	Runtime (dynamic / late binding)	Compile time (static / early binding)
Chosen by	The actual class of the object	The declared type of the reference / the class name used
@Override allowed?	Yes — and recommended	No — compiler rejects it
Effect through a parent reference holding a child object	Child version runs	Parent version runs
Can use this / super?	Yes	No this; the method belongs to the class, not an object

USMAN NOTES www.usmannotes.com

Java Program: Static Methods in Parent and Child

The program below declares a static method display in the parent class Animal and a static method with exactly the same signature in the child class Dog. An instance method sound is also overridden, so that the output contrasts the two behaviours in a single run: the instance method follows the object, while the static method follows the reference type. Read the output carefully after the code, because the contrast is the whole point of the question.

```
class Animal {
    // STATIC method in the parent class
    static void display() {
```

```
        System.out.println("Display: I am an Animal (parent static
method).");
    }

    // INSTANCE method in the parent class
    void sound() {
        System.out.println("Sound: Animal makes a sound.");
    }
}

class Dog extends Animal {
    // STATIC method with the SAME signature:
    // this HIDES Animal.display() — it does NOT override it
    static void display() {
        System.out.println("Display: I am a Dog (child static
method).");
    }

    // INSTANCE method: this genuinely OVERRIDES Animal.sound()
    @Override
    void sound() {
        System.out.println("Sound: Dog barks.");
    }
}

public class MethodHidingDemo {
```

USMAN NOTES www.usmannotes.com

```
    public static void main(String[] args) {
        // 1. Called through the class names — the clearest form:
        Animal.display();   // parent static method
        Dog.display();      // child static method (hides the parent's)

        // 2. A parent reference holding a child object:
        Animal a = new Dog();
        a.display();        // PARENT version! chosen by reference type
        a.sound();          // CHILD version! chosen by actual object

        // 3. A child reference holding the same kind of object:
        Dog d = new Dog();
        d.display();        // child version — reference type is Dog
        d.sound();          // child version
    }
}
```

Output of the program:

```
Display: I am an Animal (parent static method).
Display: I am a Dog (child static method).
Display: I am an Animal (parent static method).
Sound: Dog barks.
Display: I am a Dog (child static method).
Sound: Dog barks.
```

Explanation of the Output

Lines one and two are straightforward: when a static method is called through a class name, that class's own method runs, so Animal.display() prints the parent message and Dog.display() prints the child message. The revealing lines are three and four, which are called on the very same kind of object — in both cases a Dog object — through a reference of type Animal. Line three prints the parent's display message, proving that the static call was bound to the reference type Animal at compile time and never consulted the object. Line four prints the child's sound message, proving that the instance call was bound at runtime to the actual object, a Dog. One object, two opposite selection rules: that single contrast is the complete difference between hiding and overriding. Lines five and six confirm that when the reference type and the object type agree, as they do for the Dog reference d, both rules happen to select the child's methods, so no surprise appears. A student who can predict all six lines before running the program has understood method hiding; a student who expects line three to print the Dog message has confused hiding with overriding, which is precisely the examination trap this question is designed to expose.

In real design, method hiding should be used sparingly and deliberately. Because the executed method depends on the reference type, code that hides static methods can behave differently after an apparently harmless refactoring that changes a declared type, and readers routinely misread such calls as polymorphic when they are not. Static methods are best reserved for utility behaviour that truly belongs to one class — factory helpers, mathematical functions, constants access — and given distinct names in a subclass unless there is a clear reason to mirror the parent's API, as some framework and testing patterns do. Where genuine runtime

choice between parent and child behaviour is wanted, the correct tool is always an instance method and overriding, never a static method. It is also worth remembering that the same hiding idea applies to fields: a field declared in the child with the same name as a parent field hides the parent field, and field access, like static method access, is decided by the reference type.

Conclusion

To conclude, method hiding in Java occurs when a child class defines a static method with the same signature as a static method in its parent. The child method hides, but does not override, the parent method: both methods continue to exist in their own classes, the call is bound at compile time, and the version executed is selected by the declared type of the reference or the class name used in the call. The demonstration program proves this by calling display through an Animal reference holding a Dog object and obtaining the parent's version, while the overridden instance method sound on the same object produces the child's version. Remembering the slogan static follows the reference, instance follows the object will prevent the most common error in this topic and explains why static methods should normally be called through class names and should rarely be hidden at all.

Question 4: Sorting an Array in Ascending Order — Bubble Sort (20 Marks)

Question as printed: Write a Java program to sort an array of integers in ascending order using any sorting method (e.g., bubble sort). Explain the algorithm used. (20)

Introduction

Sorting — arranging data in a definite order, ascending or descending — is one of the most fundamental operations in computing. Sorted data can be searched in a fraction of the time by binary search, presented readably in reports and mark sheets, merged with other sorted data, and analysed for smallest, largest, median and duplicate values. Java's library provides a ready-made Arrays.sort method, but a student of programming must also be able to write a sorting algorithm from first principles, because the algorithm teaches loops, comparisons, swapping and the idea of time complexity in one compact exercise. This answer uses bubble sort, the method suggested in the question. It first explains the algorithm in plain language, step by step and pass by pass, then gives a complete Java program that sorts an array of integers in ascending order, traces the program on a concrete set of numbers, and finally discusses the efficiency of the method and the small optimisation that stops the algorithm early when the array becomes sorted before all passes are complete.

The Bubble Sort Algorithm Explained

Bubble sort works by repeatedly stepping through the array, comparing each pair of neighbouring elements, and swapping them whenever they are in the wrong order. For ascending order, the wrong order means the left element is greater than the right element. One complete step through the array is called a pass. The reason for the name is visible in the very first pass: the largest value keeps winning every comparison and being swapped rightwards, so it bubbles up to the last position, exactly as a bubble rises to the surface. After the first pass the largest element is guaranteed to be in its final place, so the second pass needs to examine only

the remaining unsorted part, and it places the second-largest element in its final place. Pass after pass, the sorted region grows from the right-hand end while the unsorted region shrinks from the left, until after at most n minus 1 passes for n elements no unsorted region remains and the array is fully sorted in ascending order.

The algorithm can be written as clear steps. Step 1: set the boundary of the unsorted region to the whole array. Step 2: for each adjacent pair within the unsorted region, from the first pair to the last, compare the two values. Step 3: if the left value is greater than the right value, swap them using a temporary variable, so that the larger value moves one step right. Step 4: at the end of the pass, reduce the unsorted region by one, because its largest value has reached its final position. Step 5: repeat the passes until the unsorted region contains fewer than two elements. An important refinement keeps a boolean flag, often called swapped: the flag is set to false at the start of each pass and to true whenever a swap occurs. If a whole pass finishes with no swap at all, the array must already be in order, and the algorithm can stop immediately instead of wasting the remaining passes. This single check makes bubble sort surprisingly fast on nearly sorted data, a common real-world case.

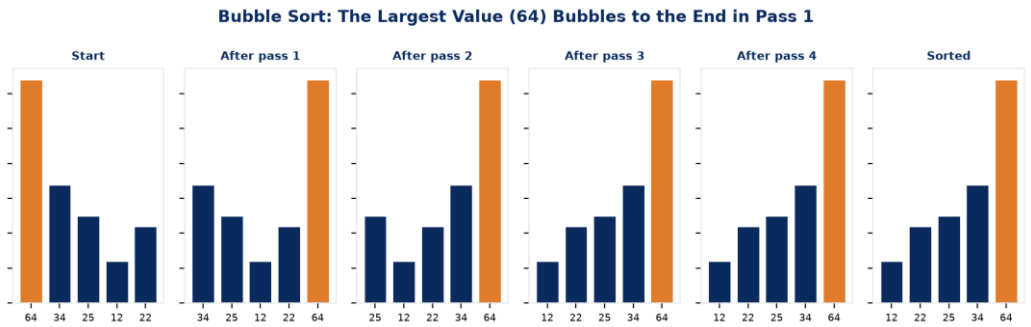


Figure 1: Bubble sort on the array 64, 34, 25, 12, 22. In pass 1 the largest value, 64, bubbles to the end; later passes settle 34 and 25 in the same way.

Worked Trace of One Example

Consider the array 64, 34, 25, 12, 22 used in the figure and in the program. In pass 1 the comparisons run as follows: 64 is compared with 34 and swapped, giving 34, 64, 25, 12, 22; then 64 is compared with 25 and swapped, giving 34, 25, 64, 12, 22; then 64 is compared with 12 and swapped, giving 34, 25, 12, 64, 22; finally 64 is compared with 22 and swapped, giving 34, 25, 12, 22, 64. The value 64 has reached the end and never needs to be examined again. In pass 2 the same process on the first four elements produces 25, 12, 22, 34, 64, placing 34 in its final position. In pass 3 the result is 12, 22, 25, 34, 64, and in pass 4 no swap occurs at all, so the early-exit flag stops the algorithm with the array sorted. Notice how each pass performs one fewer comparison than the previous pass, because the sorted tail is excluded, which is why the inner loop in the program runs to n minus 1 minus i.

Java Program: Bubble Sort in Ascending Order

```
public class BubbleSortDemo {


    // Sorts the array in ascending order using bubble sort
    static void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {          // passes
            boolean swapped = false;               // early-exit flag
            for (int j = 0; j < n - 1 - i; j++) {  // adjacent pairs
                if (arr[j] > arr[j + 1]) {         // wrong order?
                    // swap arr[j] and arr[j+1] using a temp variable
                    int temp = arr[j];
```

```
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) break;  // a full pass with no swap = sorted
        }
    }

    static void printArray(String label, int[] arr) {
        System.out.print(label);
        for (int value : arr) System.out.print(value + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        int[] numbers = {64, 34, 25, 12, 22};
        printArray("Before sorting: ", numbers);
        bubbleSort(numbers);
        printArray("After sorting:  ", numbers);
    }
}
```

Output of the program:

```
Before sorting: 64 34 25 12 22
After sorting:  12 22 25 34 64
```

Explanation of the Program

The program is built from three small methods so that each idea stands alone. The method bubbleSort contains the algorithm itself: the outer loop controls the passes, the inner loop walks through the still-unsorted region comparing neighbouring pairs, and the three-line swap, performed through the temporary variable temp, exchanges a pair that is in the wrong order. The flag swapped is reset at the start of every pass and set whenever a swap happens; if a pass ends with the flag still false, the break statement ends the sorting at once. The helper method printArray simply displays the array with a label, using an enhanced for loop, so that the before and after states can be compared on the screen. The main method creates the sample array, prints it, calls bubbleSort, and prints it again. Because arrays in Java are passed by reference to the array object, the changes made inside bubbleSort are visible in main after the call returns — no return value is needed. Reversing the single comparison, from arr[j] greater than arr[j+1] to arr[j] less than arr[j+1], would sort the same array in descending order, which shows how completely the ordering depends on that one test.

Efficiency and Comparison with Other Methods

Bubble sort performs, in the worst and average cases, roughly n squared comparisons for n elements, written as O(n squared) in Big-O notation, because each of the nearly n passes scans nearly n elements. For the small arrays used in teaching, and for the few hundred elements of a class mark sheet, this is perfectly acceptable — five elements need at most ten comparisons. For very large data, however, the quadratic growth becomes expensive: doubling the data

roughly quadruples the time, so production systems use faster algorithms such as quick sort, merge sort and the tuned dual-pivot quick sort behind Java's own Arrays.sort, which grow as O(n log n). With the early-exit flag, bubble sort's best case is much better: on an already sorted array a single pass of n minus 1 comparisons proves the order and the algorithm stops, giving O(n) best-case behaviour. Bubble sort is also stable, meaning equal elements keep their original relative order, and it sorts in place, needing only one temporary variable of extra memory. Its real value in a course such as this one is educational: no other sorting algorithm shows comparison, swapping, passes and complexity quite so visibly.

Table 1: Bubble sort at a glance

Aspect	Bubble sort behaviour
Basic idea	Compare neighbours, swap if out of order; largest value bubbles to the end of each pass
Passes needed	At most n - 1 for n elements; fewer with the early-exit flag
Worst / average time	O(n^2) comparisons and swaps
Best time (already sorted, with flag)	O(n) — one confirming pass
Extra memory	O(1) — one temporary variable; sorts in place
Stable?	Yes — equal elements keep their original order
Best used for	Small or nearly sorted data; teaching loops, swapping and complexity

USMAN NOTES www.usmannotes.com

Conclusion

To conclude, bubble sort arranges an array in ascending order by repeatedly comparing adjacent elements and swapping any pair that is in the wrong order, so that after each pass the largest remaining value settles in its final position at the end of the unsorted region. The Java program given above implements the algorithm with an outer pass loop, an inner comparison loop, a temporary variable for swapping, and an early-exit flag that stops the work as soon as a pass makes no swap. Its output, 12, 22, 25, 34, 64, is correctly sorted in ascending order. Although faster O(n log n) algorithms are preferred for very large data sets, bubble sort remains the clearest introduction to sorting, and a student who can trace it pass by pass, as done in this answer, understands the logic on which all comparison-based sorting methods are built.

Question 5: Nested Classes and Inner Classes in Java (20 Marks)

Question as printed: Explain the concept of nested classes in Java. Write a Java program to demonstrate the use of an inner class. (20)

Introduction

Most Java classes are written as separate top-level units, one class per file, but Java also allows a class to be declared inside another class. Such a class is called a nested class, and the class that contains it is called the outer or enclosing class. Nesting is not a trick for saving files; it is a design tool for grouping classes that logically belong together, for keeping a helper class hidden inside the only class that uses it, and for giving a class privileged access to the private details of its companion. The best-known example in everyday Java is the relationship between a data structure and its iterator: the iterator class is naturally nested inside the collection it traverses.

Nested classes divide into two families with quite different behaviour — static nested classes, which belong to the outer class itself, and inner classes, which belong to an object of the outer class. This answer explains the concept and the types of nested class, compares them in a table, and then demonstrates a non-static inner class with a complete program in which the inner class reads and modifies the private data of its outer object.

Types of Nested Class

A static nested class is declared with the keyword static inside the outer class. Despite being written inside, it behaves almost like an ordinary top-level class: it can be created without any outer object, as in new Outer.Inner(), written with the outer class name, and it can access only the static members of the outer class directly, because it has no link to any particular outer object. Programmers use static nested classes to group a closely related helper with its main class — for example a Node class nested inside a custom linked list, or a Builder nested inside the class it builds — without polluting the top-level namespace. A non-static nested class, by contrast, is a true inner class, and it is the kind demonstrated by the program in this answer. Every inner class object is silently attached to one specific outer class object, holds a hidden reference to it, and can therefore access all of the outer object's members, including private fields and private methods, without any qualification. In return, an inner object cannot exist on its own: it must be created through an outer object, using the syntax outerObject.new InnerClass().

Two further, more specialised forms complete the family. A local class is a class declared inside a method or block; it is visible only within that block, rather like a local variable, and is used

when a small class is needed for one short task only, most often in older event-handling and thread code. An anonymous class is a class with no name, declared and instantiated in a single expression, traditionally used to provide a one-off implementation of an interface — for example a button listener or a Runnable — although modern Java often prefers a lambda expression for the same purpose when the interface has a single abstract method. All four forms share the central idea: scope. Just as a variable declared in a small scope is protected from the rest of the program, a class declared inside another class is protected, organised and granted a carefully controlled relationship with its enclosing code. Nested classes also have sensible restrictions that follow from their nature: before Java 16 an inner class could not declare static members of its own, because it was tied to an object rather than to a class, and an inner class can never be a top-level entry point with a main method of its own in the usual style.

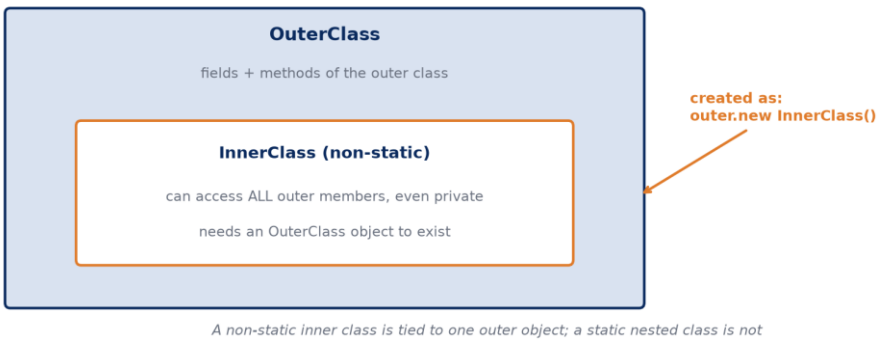


Figure 1: A non-static inner class lives inside an outer object and can reach all of that object's members, including private ones.

Table 1: The four kinds of nested class compared

Type	Declared where /	Needs an outer	Access to outer	Typical use

	how	object?	members	
Static nested class	Inside outer class, with static	No	Static members only (directly)	Helper grouped with its class, e.g. Node, Builder
Inner class (non-static)	Inside outer class, without static	Yes — created as outer.new Inner()	All members, including private	Helper that must work on one outer object's data, e.g. iterator, engine of a car
Local class	Inside a method or block	Tied to the enclosing execution	Outer members plus effectively final local variables	One short task inside a single method
Anonymous class	In one expression, no name	Depends on context	As for its enclosing context	One-off interface implementation, listener, Runnable

Why Use Nested Classes?

Four practical advantages justify the feature. First, logical grouping: when a class is useful to only one other class, keeping it inside that class places related code together and keeps the package tidy. Second, encapsulation and hiding: a nested class can be declared private, in which case no code outside the outer class can even name it, let alone misuse it — a stronger hiding than any package arrangement can give a top-level class. Third, privileged access: an inner class

can read and write the private members of its outer object directly, which removes the temptation to weaken encapsulation by making fields public or adding unnecessary getter and setter methods merely so that a helper can do its job. The outer class can likewise access the private members of its inner class. Fourth, readability and maintenance: code that belongs together can be read together, and in graphical and event driven programming inner and anonymous classes keep the handler beside the component that uses it. Set against these advantages is one cost that must be taught honestly: an inner class object keeps its outer object alive as long as the inner object exists because of the hidden reference, so careless use can retain memory longer than intended — another reason to prefer a static nested class whenever the link to an outer object is not genuinely needed.

Java Program Demonstrating an Inner Class

The program models a Car as the outer class and its Engine as a non-static inner class. The engine genuinely belongs to one particular car: it reads the car's private fuel level, reduces that private value as it runs, and reports the car's private model — all without any getter or setter. The main method shows the defining syntax of the topic: first an outer object is created, and only then is the inner object created from it with car.new Engine().

```
// OUTER CLASS
class Car {
    private String model = "Toyota Corolla";  // private outer data
    private int fuel = 10;                    // litres, private

    // INNER CLASS (non-static nested class)
```

```
    class Engine {
        void start() {
            // The inner class reads the OUTER object's
            // private members directly — no getter needed:
            System.out.println("Engine of " + model + " started.");
            System.out.println("Fuel before drive: " + fuel + "
litres");
            fuel = fuel - 3;   // it can also MODIFY private outer data
            System.out.println("Fuel after drive:  " + fuel + "
litres");
        }
    }

    void showCar() {
        System.out.println("Car model: " + model);
    }
}

public class InnerClassDemo {
    public static void main(String[] args) {
        // Step 1: create the OUTER object first
        Car car = new Car();
        car.showCar();

        // Step 2: create the INNER object THROUGH the outer object
        Car.Engine engine = car.new Engine();
        engine.start();
```

USMAN NOTES www.usmannotes.com

```
        }
    }
```

Output of the program:

```
Car model: Toyota Corolla
Engine of Toyota Corolla started.
Fuel before drive: 10 litres
Fuel after drive:  7 litres
```

Explanation of the Program

Three points in the output deserve attention. First, the type of the engine is written Car.Engine, showing that the inner class is a member of Car and is named through it, just as a field or method would be. Second, the statement car.new Engine() makes the attachment explicit: this engine belongs to this car, and if a second Car object were created, its engine would see that car's own model and fuel, completely separate from the first car's. Third, and most important, the inner method start reads and changes the private fields model and fuel with no qualification and no accessor method. If Engine were an ordinary separate top-level class, both accesses would be compile-time errors, because private means private to every other class. The inner class is the deliberate exception that Java grants to classes nested inside the owner of the data. If the outer class needs to refer to its own members unambiguously from inside the inner class, especially where names clash, it uses the qualified form Car.this.model; and where the inner and outer classes declare members with the same name, the inner member shadows the

outer one for unqualified references, again resolved with the qualified this when the outer member is intended.

Students often ask when to choose an inner class over a static nested class, and the program supplies the test. Ask: does the nested object need to work on the data of one particular outer object? The engine does — it burns this car's fuel — so a non-static inner class is correct. If the answer is no, as with a Node that merely holds data and links, a static nested class is cleaner, uses slightly less memory, and avoids the hidden outer reference entirely. The same judgement explains the design of the Java library itself: the entry and node types inside collection classes are typically static nested classes, while iterators, which must walk the current contents of one specific collection object, have traditionally been implemented as inner classes. Learning nested classes therefore does more than add a syntax feature — it trains the judgement of placing each class in the smallest scope that honestly expresses its relationship to the rest of the program.

Conclusion

To conclude, a nested class is a class declared inside another class, used to group logically related classes together, to hide a helper class inside its only user, and to give the nested class controlled access to the outer class's private members. Static nested classes belong to the outer class and need no outer object, while true inner classes belong to one outer object, are created with outer.new Inner(), and can read and modify that object's private data directly, as the Engine inner class does with the Car's private fuel and model in the demonstration program. Local and anonymous classes extend the same idea to method scope and to single-expression

use. Used with judgement — inner only when a real link to an outer object exists, static nested otherwise — nested classes produce code that is better encapsulated, better organised and closer to the real relationships between the objects being modelled.

USMAN NOTES
www.usmannotes.com