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. 2

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: Java Annotations — Purpose and Built-In Annotations (20 Marks)

Question as printed: What are Java annotations? Explain their purpose and write a Java program using at least two built-in annotations. (20)

Introduction

An annotation is a form of metadata — data about the program itself — that a programmer attaches to a class, method, field, parameter or other declaration in Java source code. An annotation begins with the at sign, as in @Override, and it changes nothing about the direct

logic of the statement it marks. Instead, it gives information to three possible audiences: the compiler, which can check rules and issue warnings; development tools and frameworks, which can generate code, configuration or documentation from the marks; and, for annotations retained at runtime, the running program itself, which can read them through reflection and act on them. Annotations were introduced in Java 5 to replace fragile conventions such as long marker names in comments and separate XML configuration files with a standard, type-checked mechanism written inside the code where it belongs. This answer explains what annotations are, sets out their purposes, describes the principal built-in annotations supplied with the Java language, and then demonstrates two of them — in fact three — in one complete program whose compiler behaviour is explained.

What Are Annotations? Purpose Explained

The purposes of annotations fall into three groups. The first group is instructions and checks for the compiler. An annotation can order the compiler to verify something that the programmer claims is true, to suppress a specific warning that the programmer has examined and accepted, or to mark code as obsolete so that anyone who still uses it is warned. These annotations cost nothing at runtime and exist purely to catch mistakes earlier and to make intentions explicit to human readers. The second group is processing at compile time or deployment time. Frameworks and tools read annotations to generate extra source code, database mappings, web-service endpoints or configuration, so that the programmer declares what is wanted — this method answers this web address, this field maps to that database column — instead of writing repetitive boilerplate and separate configuration files by hand. Modern Java development with frameworks for web services, persistence and testing is built overwhelmingly

on this use. The third group is runtime processing through reflection: an annotation whose retention is RUNTIME remains stored in the class file and can be discovered while the program runs, allowing a framework to decide behaviour dynamically — for example which methods are test cases, which methods may be called by which users, or in what order steps should execute.

Two technical ideas govern how an annotation works. The retention policy states how long the annotation is kept: SOURCE annotations are discarded by the compiler after checking, CLASS annotations are stored in the class file but not normally available at runtime, and RUNTIME annotations are stored and can be read by reflection. The target states where the annotation may legally be written: on a type, a method, a field, a parameter, a constructor and several other places, and the compiler rejects an annotation used in a place its definition forbids. Programmers may also define their own annotations with the @interface syntax and attach information to them with meta-annotations such as @Retention and @Target, although this question asks only about the built-in annotations supplied by Java itself. A final point of style should be learned early: an annotation is information about code, not a substitute for code. It never runs by itself; something — compiler, tool or framework — must read it and act, and an annotation that nothing reads has no effect at all.

Table 1: The principal built-in annotations of Java

Annotation	Written on	Purpose	What happens if ignored / misused
@Override	A method	Declares that the method overrides a method of the	Compiler error if the signature does not actually match an

		superclass or implements an interface method	inherited method — catches spelling and parameter mistakes
@Deprecated	Class, method or field	Marks the element as obsolete and discouraged	Compiler warning wherever the element is still used
@SuppressWarnings	Class, method, field, variable	Suppresses named compiler warnings, e.g. "unchecked", "deprecation"	Warnings are hidden for that element only; misuse can hide real problems
@FunctionalInterface	An interface	Declares that the interface is intended to have exactly one abstract method	Compiler error if a second abstract method is added
@SafeVarargs	A method with varargs of a generic type	Declares that the method safely uses its variable arguments	Suppresses the heap-pollution warning at the declaration and call sites

USMAN NOTES www.usmannotes.com

Java Program Using Built-In Annotations

The program below uses three built-in annotations so that the requirement of at least two is comfortably exceeded. @Deprecated marks an old method of a parent class as obsolete. @Override guarantees that the child's greeting method genuinely overrides the parent's.

@SuppressWarnings is used in the main method for a small, deliberate demonstration involving the deprecated method. The comments in the code explain the compiler message each annotation produces.

```
class Greeter {
    // @Deprecated: this old method should no longer be used.
    @Deprecated
    void oldGreeting() {
        System.out.println("Hello (old style).");
    }

    void greeting() {
        System.out.println("Hello from the parent Greeter.");
    }
}

class FriendlyGreeter extends Greeter {
    // @Override: the compiler CHECKS that this method really
    // overrides a parent method. A typing mistake in the name
    // or parameters would be a compile-time ERROR, not a
    // silent new method.
    @Override
    void greeting() {
        System.out.println("Hello from the child FriendlyGreeter!");
    }
}
```

USMAN NOTES www.usmannotes.com

```
public class AnnotationDemo {
    // @SuppressWarnings: we KNOW oldGreeting() is deprecated;
    // we call it once, deliberately, to show its output, so the
    // deprecation warning is suppressed for this method only.
    @SuppressWarnings("deprecation")
    public static void main(String[] args) {
        FriendlyGreeter g = new FriendlyGreeter();
        g.greeting();       // overridden method — child version runs
        g.oldGreeting();    // deprecated method — still works,
                            // but the annotation warns against using it
        System.out.println("Program finished.");
    }
}
```

USMAN NOTES www.usmannotes.com

Output of the program:

```
Hello from the child FriendlyGreeter!
Hello (old style).
Program finished.
```

Explanation of the Program and of Compiler Behaviour

Running the program is only half of the demonstration; the other half happens at compile time, and a student should deliberately experiment with it. First, change the name greeting in FriendlyGreeter to greetings, keeping the @Override annotation. With ordinary overriding this mistake would compile silently and produce a brand-new method that nothing ever calls — one

of the most confusing bugs in inheritance — but with @Override the compiler stops with an error stating that the method does not override or implement a method from a supertype. The annotation has converted a silent logic error into an immediate, clearly located compile error. Second, remove the @SuppressWarnings line from main and compile again: the program still compiles, because using deprecated code is legal, but the compiler now issues a warning that oldGreeting has been deprecated, exactly as the @Deprecated annotation on the method intends. The warning travels with the method wherever it is used, in any project, without the author of Greeter writing a word of checking code. Third, notice what did not happen: none of the annotations changed the printed output in any way. greeting still prints the child's message by ordinary overriding, and oldGreeting still prints its old message. Annotations informed the compiler and the reader; they did not execute.

In professional Java development the same three ideas scale up enormously. Persistence frameworks map classes to database tables with annotations such as @Entity and @Column placed on the class and its fields, so the mapping lives beside the code it describes instead of in a separate file that can drift out of date. Web frameworks route requests with annotations placed on handler methods. Testing frameworks discover test methods by a runtime annotation, executing each marked method and reporting the results, while the test class itself contains no registration code at all. In each case the pattern is the one this small program illustrates: the programmer declares intent with a standard mark, and a tool that understands the mark does the repetitive work. For this reason annotation literacy is no longer optional for a Java programmer — reading modern framework code is impossible without it — yet the built-in

annotations remain the right starting point, because @Override alone, used on every overriding method without exception, prevents an entire category of inheritance mistakes.

Conclusion

To conclude, Java annotations are standardised metadata attached to declarations with the at sign. They serve the compiler, which checks claims such as @Override and warns about @Deprecated code; they serve tools and frameworks, which generate code and configuration from declared marks; and, when retained at runtime, they serve the running program through reflection. The demonstration program uses @Override to guarantee a genuine override, @Deprecated to mark an obsolete method, and @SuppressWarnings to accept one known warning deliberately, and its compile-time experiments show the real value of annotations: mistakes caught early, intentions made visible, and repetitive work handed to tools — all without changing what the program itself computes.

Question 2: LinkedList in Java — Adding, Removing and Displaying Elements (20 Marks)

Question as printed: Write a Java program to demonstrate the use of LinkedList. Add elements, remove an element, and display the updated list. Also, describe how a LinkedList stores data. (20)

Introduction

Java's Collections Framework offers several list implementations, and choosing between them requires knowing how each one stores its data internally. The two most used are ArrayList,

which stores elements in a resizable array, and LinkedList, the subject of this question, which stores elements in a chain of separate node objects linked to one another by references. LinkedList, in the package java.util, implements the List interface and also the Deque interface, so the same class can be used as an ordinary list, as a queue, or as a stack. Its great strength is cheap insertion and deletion, especially at the two ends and wherever an iterator already stands, because elements are never shifted in memory; its corresponding weakness is slow access by index, because reaching the hundredth element means walking through the ninety-nine nodes before it. This answer first describes exactly how a LinkedList stores data, node by node, then gives a complete program that adds elements, removes an element, and displays the list before and after the removal, and finally compares LinkedList with ArrayList so that the choice between them can be made intelligently.

How a LinkedList Stores Data

A LinkedList does not store its elements side by side in one continuous block of memory, as an array does. Instead, every element is wrapped in its own separate object called a node. In Java's implementation, which is a doubly linked list, each node contains three things: the element itself, a reference usually called next pointing to the following node, and a reference usually called prev pointing to the previous node. The list object itself remembers only two nodes directly: the first node, called the head, and the last node, called the tail. The head's prev reference is null, because nothing stands before it, and the tail's next reference is null, because nothing stands after it. The nodes may be scattered anywhere in memory; their logical order exists only through the chain of references, exactly as a treasure hunt moves from clue to clue rather than by positions on a shelf. To reach an element the program must follow the chain

from an end — and because the list is doubly linked, Java starts from whichever end, head or tail, is closer to the requested index, which halves the walking distance without removing the fundamental cost.

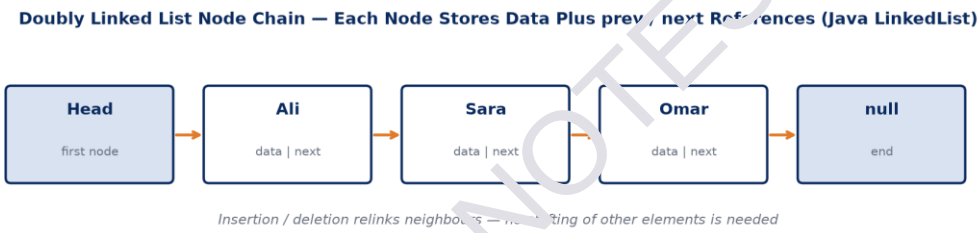


Figure 1: A doubly linked list. Each node holds one element plus references to its neighbours; the list itself holds only the head and the tail.

This storage design explains every performance characteristic of the class. Adding an element at the head or tail is constant-time work: one new node is created and two or three references are rewired, and no other element moves. Removing an element, once its node is located, is equally cheap: the node's predecessor is linked directly to its successor, the removed node is abandoned to the garbage collector, and again nothing shifts — in sharp contrast to an array, where removing the first element forces every remaining element to move one place left. The price is paid in access and memory. Asking for the element at index 500 requires following 500 references (or fewer, starting from the nearer end), so indexed access is proportional to the index, written O(n), while an array answers the same request instantly in O(1). Each element also costs more memory, because every node carries two references in addition to the element itself, whereas an array stores references only. A LinkedList is therefore not better or worse than an ArrayList in general; it is better exactly where a program inserts and removes

frequently, especially at the ends, and worse where a program reads elements by position frequently.

Java Program: Add, Remove and Display

The program below creates a LinkedList of student names, adds elements at the end and at the beginning, displays the list, removes one named element and also removes the element at a stated index, and then displays the updated list exactly as the question requires. It also shows three small extras that reveal the Deque nature of the class: reading the first and last elements without removing them, and reporting the size of the list.

```
import java.util.LinkedList;


public class LinkedListDemo {
    public static void main(String[] args) {
        // Create an empty LinkedList of Strings
        LinkedList<String> students = new LinkedList<>();


        // ADD elements at the end
        students.add("Ali");
        students.add("Sara");
        students.add("Omar");
        students.add("Fatima");


        // ADD elements at the two ends (Deque methods)
        students.addFirst("Zainab");  // becomes the head
        students.addLast("Bilal");    // becomes the tail
```

```
        System.out.println("List after adding elements:");
        System.out.println(students);
        System.out.println("Size: " + students.size());
        System.out.println("First: " + students.getFirst()
                           + "  Last: " + students.getLast());

        // REMOVE an element by value
        students.remove("Omar");

        // REMOVE an element by index (index 1 at this moment)
        students.remove(1);

        System.out.println();
        System.out.println("Updated list after removals:");
        System.out.println(students);
        System.out.println("Size: " + students.size());

        // DISPLAY the updated list element by element
        System.out.println("Elements one by one:");
        for (String name : students) {
            System.out.println(" - " + name);
        }
    }
}
```

USMAN NOTES www.usmannotes.com

Output of the program:

```
List after adding elements:
[Zainab, Ali, Sara, Omar, Fatima, Bilal]
Size: 6
First: Zainab  Last: Bilal


Updated list after removals:
[Zainab, Sara, Fatima, Bilal]
Size: 4
Elements one by one:
 - Zainab
 - Sara
 - Fatima
 - Bilal
```

USMAN NOTES www.usmannotes.com

A careful reading of the output repays attention, because the second removal surprises many students. After the six additions the list is Zainab, Ali, Sara, Omar, Fatima, Bilal. Removing the value Omar deletes that node and relinks its neighbours directly to each other — no other name moves in memory, only references change. The list is then Zainab, Ali, Sara, Fatima, Bilal, so index 1 at that moment holds Ali. Removing index 1 therefore deletes Ali, leaving the final updated list Zainab, Sara, Fatima, Bilal, of size 4, exactly as printed. Students should trace this for themselves, because it teaches the essential lesson that indexes are positions counted afresh at the moment of each call, not permanent labels attached to elements: the same index number names a different element after an earlier removal has taken place.

LinkedList Versus ArrayList

Table 1: LinkedList and ArrayList compared

Operation / feature	LinkedList	ArrayList
Storage	Chain of separate node objects with prev / next references	One resizable array
Access by index get(i)	Slow — O(n), walk the chain from the nearer end	Fast — O(1), direct array access
Add / remove at the ends	Fast — O(1), relink a few references	Fast at the end (amortised); slow at the start — shifts elements
Add / remove in the middle	Fast once the position is reached; no shifting	Shifts all following elements — O(n)
Memory per element	Higher — element plus two references per node	Lower — one reference per array slot
Extra roles	Also a Deque: queue and stack methods (addFirst, poll, push, pop)	List only
Choose it when	Frequent insertions / deletions, queue-like use	Frequent reading by index, general default choice

The practical guidance that follows from the table is simple. Use a LinkedList when the program behaves like a queue or a waiting line, when elements are constantly added and removed at the ends, or when an iterator is already moving through the list and inserting or deleting as it

goes. Use an ArrayList — the normal default — when elements are mainly read, searched by position, or added at the end. Declaring the variable with the interface type, for example List of String, where only list behaviour is needed, keeps the code flexible enough to change the implementation later with a one-line edit, which is good professional practice in either case.

Conclusion

To conclude, a LinkedList stores its data as a doubly linked chain of independent node objects, each holding one element plus references to its predecessor and successor, with the list remembering only the head and the tail. Because order lives in the references rather than in memory positions, insertion and removal need only a few references to be rewired and no element ever shifts, while access by index must walk the chain and is correspondingly slow. The demonstration program adds six names with add, addFirst and addLast, displays the list, removes one element by value and one by index, and displays the updated list, exactly as the question requires. Choosing LinkedList for insertion-heavy and queue-like work, and ArrayList for index-heavy reading, is the professional judgement this topic is meant to build.

Question 3: Synchronization in Multithreading (20 Marks)

Question as printed: Explain the concept of synchronization in multithreading. Write a Java program using a synchronized method to prevent data inconsistency. (20)

Introduction

A multithreaded program runs several threads — independent paths of execution — at the same time inside one process, sharing the same memory. Multithreading makes programs

responsive and fast: a server can serve many clients at once, and a long calculation need not freeze the screen. The same sharing that creates the power, however, creates the danger. When two threads read and modify the same data at the same time without coordination, their operations can interleave in ways that lose updates, read half-written values, and leave the data inconsistent — correct by neither thread's intention. Synchronization is Java's principal mechanism for preventing exactly this: it ensures that a sensitive block of code is executed by only one thread at a time, in an orderly, atomic fashion. This answer explains the race condition that synchronization prevents, explains how the synchronized keyword and the monitor lock behind it work, and then presents a complete program in which a synchronized method protects a shared bank-like counter, contrasting its reliable output with the unreliable output of the same program without synchronization.

The Problem: Race Conditions and Data Inconsistency

The classic illustration is a shared counter incremented by two threads. Incrementing looks like one action but is really three: read the current value, add one, and write the result back. Suppose the counter stands at 100 and both threads read it at nearly the same moment. Each adds one in its own registers, obtaining 101, and each writes 101 back. Two increments have been performed, yet the counter has advanced only once — one update has been silently lost. Which update is lost, and how often it happens, depends on the exact timing of the threads, which the operating system scheduler changes from run to run. A section of code whose correctness depends on the relative timing of threads in this way is called a critical section, and the defect is called a race condition, because the threads are effectively racing and the result depends on who happens to win each step. Data inconsistency is the visible consequence: a

bank balance credited twice but increased once, seats sold twice for the same booking, or, in the demonstration below, a counter that should reach exactly 2,000 but stops at an unpredictable smaller number that differs on every run. Non-determinism — a program that gives different answers on identical input — is what makes race conditions so dangerous and so hard to test away.

Synchronization and the Monitor Lock

Synchronization solves the problem by attaching a lock, called a monitor, to every Java object. When a method is declared with the keyword synchronized, a thread entering that method must first acquire the monitor of the object on which the method was called. If another thread already holds that monitor, the entering thread is blocked — made to wait — until the holder finishes the method and releases the lock. Only then does one waiting thread acquire the lock and proceed. The effect is that the whole synchronized method becomes effectively atomic with respect to other synchronized methods on the same object: it runs from start to finish without any other synchronized method on that object interleaving with it. In the counter example, the read, add and write are forced to happen as one uninterruptible unit, so no update can ever be lost. Java offers the same protection at a finer grain with a synchronized block, synchronized (object) { ... }, which locks only the statements inside the braces and is preferred when only a small part of a method is critical, because it keeps the locked region — and therefore the waiting time of other threads — as short as possible. A synchronized static method locks the monitor of the Class object instead of an instance. Two supporting guarantees complete the picture: releasing a monitor makes all the holder's writes visible to the next thread that acquires the same monitor, so synchronized code also fixes visibility problems, and

Java monitors are reentrant, meaning a thread that already holds a lock may acquire it again when synchronized methods call one another, without deadlocking against itself.

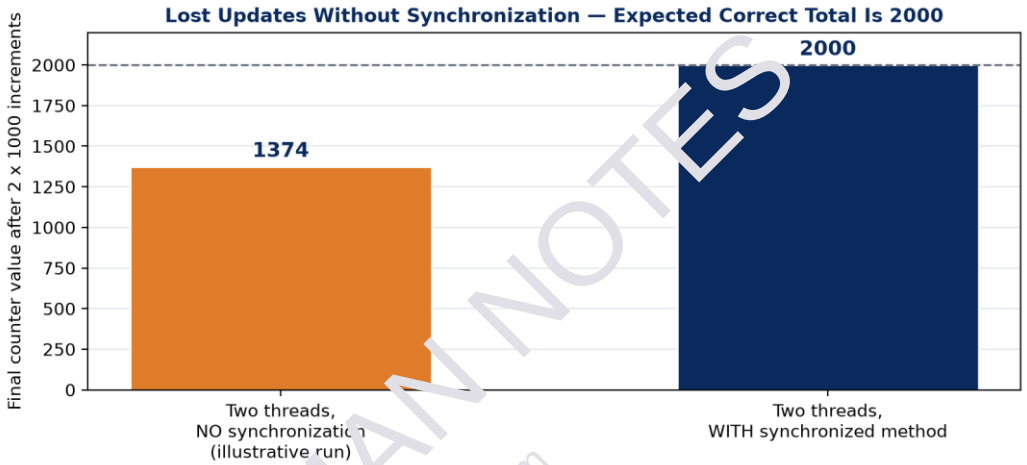


Figure 1: Without synchronization, two threads incrementing a shared counter 1,000 times each lose updates and finish below 2,000 (illustrative run). With a synchronized method the total is exactly 2,000 on every run.

Java Program: A Synchronized Method Protecting a Shared Counter

The program models the simplest possible shared resource: a counter that two threads each increment one thousand times. The increment method is declared synchronized, so the two threads are forced to take turns inside it, one complete increment at a time. The main method starts both threads, waits for both to finish with join, and prints the final value, which is exactly 2,000 every time the program runs. Removing the single word synchronized — an experiment every student should perform — makes the final value fall short of 2,000 by an unpredictable amount that changes on each run, which is the data inconsistency the question asks about, made visible.

```
class Counter {
```

```
    private int count = 0;

    // SYNCHRONIZED METHOD: only ONE thread may execute
    // this method on this object at a time. The read-add-write
    // inside it can therefore never be interrupted halfway.
    synchronized void increment() {
        count++;
    }

    int getCount() {
        return count;
    }
}

class Worker extends Thread {
    private Counter counter;

    Worker(Counter counter, String name) {
        super(name);
        this.counter = counter;   // SHARED counter object
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
```

USMAN NOTES www.usmannotes.com

```
            System.out.println(getName() + " finished.");
        }
    }

    public class SynchronizationDemo {
        public static void main(String[] args) throws InterruptedException
    {
            Counter shared = new Counter();           // ONE shared object
            Worker t1 = new Worker(shared, "Thread-1");
            Worker t2 = new Worker(shared, "Thread-2");

            t1.start();
            t2.start();

            t1.join();   // wait until both threads have finished
            t2.join();

            System.out.println("Expected value: 2000");
            System.out.println("Actual value:   " + shared.getCount());
        }
    }
```

Output of the program (with synchronized, on every run):

```
Thread-1 finished.
Thread-2 finished.
```

```
Expected value: 2000
Actual value:   2000
```

Typical output of the same program with the word synchronized removed (different on every run):

```
Thread-1 finished.
Thread-2 finished.
Expected value: 2000
Actual value:   1374
```

USMAN NOTES www.usmannotes.com

Explanation, Costs and Good Practice

The correctness of the synchronized version rests on one fact that students must state precisely: both worker threads call increment on the same Counter object, so both contend for the same monitor. Had each thread been given its own Counter object, synchronization would have been useless decoration, because each object has its own lock and its own count, and each thread would simply reach 1,000 in private. Synchronization only ever coordinates threads that share both the data and the lock that guards it. It is equally important to see what synchronization does not do. It does not make the program single-threaded overall: while one thread holds the counter's monitor, the other threads remain free to do any work that does not need that monitor, such as preparing data or updating the display. Only the small critical section is serialised. Nor does synchronization automatically prevent every threading defect: if two locks are acquired in different orders by different threads, the threads can deadlock, each

waiting forever for the other's lock, and synchronization must therefore be designed with a consistent lock order and with critical sections kept as short as reasonably possible.

There is also a real cost to acknowledge. A thread that finds the monitor held must wait, and if a synchronized method is long or very frequently called, threads spend their time queuing instead of working, and the program can become slower than a single-threaded version. Good practice follows directly: synchronize only the code that touches shared mutable data, prefer a synchronized block around the few critical statements over synchronizing an entire long method, never hold a lock while performing slow input or output, and consider the ready-made tools Java provides for common cases — atomic classes such as AtomicInteger for simple counters, which achieve the same correctness as the demonstration program without an explicit lock, and the concurrent collections, which are internally synchronized by design. The volatile keyword deserves a careful mention too: it guarantees that reads and writes of a single variable are visible and atomic, which is enough for a simple flag, but it cannot protect a compound action such as increment, because read-add-write can still interleave — which is why the counter in this answer needs synchronized and not merely volatile.

Table 1: Synchronization at a glance

Aspect	Explanation
Problem prevented	Race conditions: interleaved read-modify-write operations losing updates and leaving shared data inconsistent
Mechanism	Every object has a monitor (lock); a synchronized method or block must acquire it, and only one

	thread holds it at a time
Guarantee	Critical section runs atomically with respect to other synchronized code on the same object; writes become visible to the next lock holder
Forms	synchronized method, synchronized block on an object, synchronized static method (locks the Class object)
Cost / risk	Waiting threads and reduced parallelism if sections are long; deadlock if locks are taken in inconsistent orders
Alternatives for simple cases	AtomicInteger and other java.util.concurrent.atomic classes; concurrent collections; volatile for a single flag only

USMAN NOTES www.usmannotes.com

Conclusion

To conclude, synchronization in multithreading is the disciplined use of an object's monitor lock to ensure that a critical section — code that reads and modifies shared data — is executed by only one thread at a time. Without it, interleaved operations produce race conditions and data inconsistency, as the unsynchronized version of the demonstration shows by finishing below the expected 2,000 by an unpredictable amount. With a single synchronized keyword on the increment method, the same program finishes at exactly 2,000 on every run, because each read-add-write is performed atomically and every update is preserved. Used on the smallest region that genuinely needs it, on a lock that is truly shared, synchronization converts an

unreliable concurrent program into a correct one — which is why it is the foundational thread-safety technique every Java programmer must master before using threads on shared data.

Question 4: throw Versus throws in Java Exception Handling (20 Marks)

Question as printed: What is the difference between throw and throws? Write a Java program that uses both to handle exceptions. (20)

Introduction

Exception handling is Java's structured way of dealing with abnormal situations — invalid input, a missing file, division by zero, a failed network connection — without allowing them to crash the program or corrupt its data. Two keywords of this system are confused by students more often than any others, because their names differ by a single letter: throw and throws. They are not two spellings of one idea; they do opposite kinds of work at opposite places in the code. The keyword throw is an action performed inside a method body: it actually creates and launches an exception object at the moment a problem is detected. The keyword throws is a declaration placed in a method header: it warns callers, and instructs the compiler, that the method may pass a particular checked exception outward instead of handling it itself. This answer defines each keyword precisely, compares them point by point in a table, and then gives a complete program that uses both together — a method that throws an exception with throw, declares it with throws, and a caller that handles it with try and catch — explaining how control flows through the program in both the success and the failure cases.

throw: Actually Throwing an Exception

The throw keyword is an executable statement. It is written inside a method or block, it is followed by a single exception object — normally created on the spot with new — and when it executes, the normal flow of the method stops immediately at that line and the Java runtime begins searching for a matching catch block to handle the exception. No statement after the throw in that path is executed unless the exception is caught inside the same method. Programmers use throw to enforce the rules of their own code: a method that receives an age of minus five, a mark of 140 out of 100, or a withdrawal larger than the balance cannot proceed sensibly, so it throws an exception describing the violation, often an IllegalArgumentException or a custom exception class written for the application. Any object whose class extends Throwable can be thrown, which includes both checked exceptions, such as IOException, and unchecked exceptions, such as ArithmeticException and the RuntimeException family. After the keyword throw exactly one object is named — throw is singular in grammar and in behaviour, one statement throws one exception object — and code written after a throw that can never be reached is rejected by the compiler as unreachable, which is further proof that throw is a real control-flow action and not a comment or declaration.

throws: Declaring That a Method May Pass an Exception On

The throws keyword, by contrast, executes nothing. It appears once, in the header of a method or constructor, after the parameter list and before the body, followed by a list of one or more exception class names separated by commas, as in void readFile() throws IOException, FileNotFoundException. Its job is contractual and informational. For checked exceptions, Java

enforces the handle-or-declare rule: a method that may cause a checked exception must either handle it internally with try and catch, or declare it with throws so that the responsibility passes visibly to its caller. The caller is then placed under the same obligation, and in this way the declaration climbs the call chain until some method finally handles the exception, or the main method declares it and the runtime terminates the program with a stack trace if it occurs. Declaring an exception does not mean the exception will occur; it means it might, and that every caller is entitled — and for checked exceptions, compelled — to plan for that possibility. Unchecked exceptions may be listed in a throws clause for documentation, but the compiler does not require it, because unchecked exceptions represent programming errors and conditions that may arise almost anywhere. It is also a rule of overriding that a child method may not declare broader checked exceptions than the parent method it overrides, so a throws clause is genuinely part of a method's public contract, visible in documentation and enforced by the compiler on both sides of the call.

Table 1: throw versus throws compared

Feature	throw	throws
Nature	An executable statement — an action	A declaration — part of a method's contract
Written where	Inside a method body or block	In the method or constructor header, after the parameters
Followed by	One exception object, e.g. new ArithmeticException("...")	One or more exception class names, comma-separated
How many exceptions	Exactly one object per throw	Several classes may be declared

	statement	in one clause
Effect when reached	Stops the current flow and starts the search for a catch block	Nothing executes; it informs callers and the compiler
Main purpose	To raise / create an exception when a problem is detected	To declare that checked exceptions may propagate to the caller
Example	throw new IllegalArgumentException("Age cannot be negative");	void checkAge(int age) throws IllegalArgumentException
Compiler's role	Checks reachability after the statement	Enforces handle-or-declare for checked exceptions on every caller

USMAN NOTES www.usmannotes.com

Java Program Using Both throw and throws

The program models a simple eligibility check. The method checkAge validates an age: if the age is below 18 it uses throw to raise an ArithmeticException with a clear message, and its header uses throws to declare that possibility to callers. The main method calls checkAge inside a try block and handles the possible exception in a catch block, so the program reports the problem gracefully instead of crashing. The method is called twice — once with a valid age and once with an invalid age — so both paths of the logic appear in the output.

```
public class ThrowThrowsDemo {
```

```
    // 'throws' in the HEADER declares that this method may
    // pass an ArithmeticException to its caller.
    static void checkAge(int age) throws ArithmeticException {
        if (age < 18) {
            // 'throw' in the BODY actually creates and throws
            // ONE exception object at this moment:
            throw new ArithmeticException(
                "Access denied: age " + age + " is below 18.");
        }
        System.out.println("Access granted: age " + age + " is
valid.");
    }

    public static void main(String[] args) {
        // Case 1: a valid age — no exception occurs
        try {
            checkAge(21);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }

        // Case 2: an invalid age — throw fires inside checkAge,
        // the exception travels to this caller (as 'throws'
        // declared), and the catch block handles it:
        try {
            checkAge(15);
        } catch (ArithmeticException e) {
```

```
            System.out.println("Exception caught: " + e.getMessage());
        }

        System.out.println("Program ended normally — it did not
crash.");
    }
}
```

Output of the program:

```
Access granted: age 21 is valid.
Exception caught: Access denied: age 15 is below 18.
Program ended normally — it did not crash.
```

USMAN NOTES www.usmannotes.com

Explanation of the Flow of Control

Trace the second call carefully, because it exercises every element of the question in sequence. The main method enters its try block and calls checkAge(15). Inside checkAge the test age less than 18 succeeds, the throw statement executes, and an ArithmeticException object carrying the message is created. At that instant the remaining flow of checkAge is abandoned — no further statement in that call runs — and the runtime looks for a handler. Because checkAge itself contains no try and catch, and because its header honestly declared the exception with throws, the exception propagates outward to the caller, precisely as the contract announced. The caller's catch block matches the exception type, so it receives the object, prints its message with getMessage, and control continues after the try-catch structure, reaching the final line

that proves the program survived. The first call shows the complementary path: the test fails, throw never executes, the success message prints, and the catch block is simply skipped. Two further details complete understanding. If the catch block were removed and main did not declare the exception either, an unchecked exception such as this one would still compile — unchecked exceptions are exempt from the handle-or-declare rule — but it would terminate the thread with a stack trace at runtime; replacing ArithmeticException with a checked exception such as IOException would make the compiler itself refuse to compile until the exception is caught or declared, which is the sharpest practical demonstration of what throws is for. Finally, throw is frequently used with custom exception classes — a class extending Exception, written by the programmer with a descriptive name such as InsufficientBalanceException — so that an application's own rules can be reported through the same standard mechanism as the library's exceptions.

Good practice ties the two keywords together in a single discipline: detect problems early and throw specific exceptions with clear messages at the point of detection; declare checked exceptions honestly with throws rather than swallowing them in an empty catch block; and handle exceptions at the level that can actually do something useful — inform the user, retry, substitute a default, or record the failure — instead of catching them merely to print a line and continue with corrupt data. Used this way, throw supplies the alarm, throws supplies the warning on the door, and try-catch supplies the response plan, and a program's error handling becomes as deliberate as its normal logic.

Conclusion

To conclude, throw and throws differ in nature, position and effect. Throw is an executable statement inside a method body that creates and launches one exception object when a problem is detected, immediately diverting the flow of control to a handler. Throws is a declaration in a method header that lists the checked exception classes the method may pass to its caller, enforcing Java's handle-or-declare rule and forming part of the method's public contract. The demonstration program uses both exactly as they are meant to be used together: checkAge declares with throws, fires with throw when the age is invalid, and its caller handles the travelling exception with try and catch, so valid input succeeds, invalid input is reported clearly, and the program ends normally in both cases.

Question 5: Polymorphism Using an Interface (20 Marks)

Question as printed: Write a Java program to demonstrate polymorphism using an interface and multiple implementing classes. Explain how interface references support runtime polymorphism. (20)

Introduction

Polymorphism, from Greek words meaning many forms, is the object-oriented principle that one reference, one method call, or one interface can stand for objects of several different classes, with the actual behaviour decided by the object involved rather than by the code that issues the call. It is the property that lets a program treat a circle, a rectangle and a triangle uniformly as shapes while each one draws and measures itself in its own way. Java supports polymorphism through both inheritance and interfaces, and the interface form, required by this

question, is the more flexible of the two, because a class may implement many interfaces even though it may extend only one class. This answer explains the idea, presents a complete program in which an interface is implemented by three different classes and used through interface references held in an array, and then explains in detail how an interface reference makes runtime polymorphism possible — what the compiler checks, what the Java Virtual Machine decides, and why this design makes programs easier to extend without modification.

Interfaces and Implementing Classes

An interface is a pure contract: it declares what methods a class must provide, normally without saying how those methods work. A class that signs the contract with the keyword implements must supply a concrete body for every abstract method of the interface, or be declared abstract itself. In the program below the interface Shape declares two methods, draw and area. The classes Circle, Rectangle and Triangle each implement Shape, and each writes the two methods differently, because drawing and measuring genuinely differ from shape to shape. Nothing in the interface mentions circles or triangles; nothing in any implementing class mentions the others. The only shared knowledge is the contract itself. That separation is the foundation of everything that follows: code written against the contract can work with any present or future class that honours it, including classes that have not yet been written when the using code is compiled.

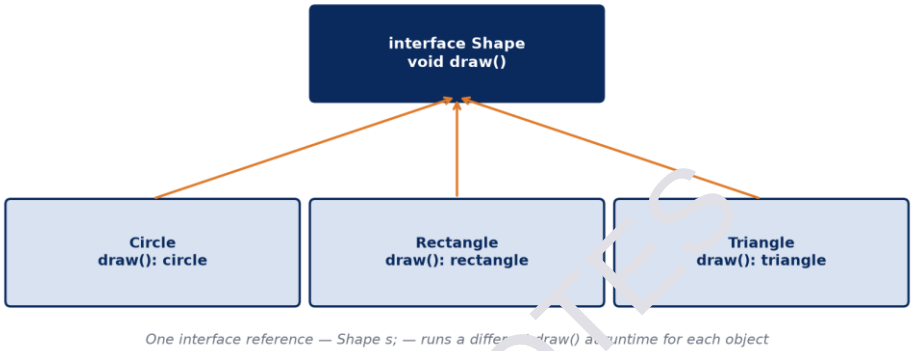


Figure 1: One interface, three implementing classes. A Shape reference can hold a Circle, a Rectangle or a Triangle object, and the same draw() call runs a different body for each.

Java Program: Polymorphism Through an Interface

```
// The INTERFACE: a pure contract
interface Shape {
    void draw();
    double area();
}

// Implementing class 1
class Circle implements Shape {
    private double radius;
    Circle(double radius) { this.radius = radius; }
    public void draw() {
        System.out.println("Drawing a CIRCLE of radius " + radius);
    }
    public double area() {
        return 3.1416 * radius * radius;
```

```
    }
}

// Implementing class 2
class Rectangle implements Shape {
    private double length, width;
    Rectangle(double length, double width) {
        this.length = length; this.width = width;
    }
    public void draw() {
        System.out.println("Drawing a RECTANGLE " + length + " x " +
width);
    }
    public double area() {
        return length * width;
    }
}

// Implementing class 3
class Triangle implements Shape {
    private double base, height;
    Triangle(double base, double height) {
        this.base = base; this.height = height;
    }
    public void draw() {
        System.out.println("Drawing a TRIANGLE base " + base
                           + ", height " + height);
```

```
    }
    public double area() {
        return 0.5 * base * height;
    }
}


public class PolymorphismDemo {
    public static void main(String[] args) {
        // INTERFACE REFERENCES holding different objects:
        Shape[] shapes = {
            new Circle(5),
            new Rectangle(4, 6),
            new Triangle(4, 3)
        };

        // ONE loop, ONE kind of call — but each object
        // behaves in its OWN way at runtime:
        for (Shape s : shapes) {
            s.draw();   // which draw()? decided at RUNTIME
            System.out.println("   Area = " + s.area());
        }
    }
}
```

Output of the program:

```
Drawing a CIRCLE of radius 5.0
```

```
   Area = 78.54
Drawing a RECTANGLE 4.0 x 6.0
   Area = 24.0
Drawing a TRIANGLE base 4.0, height 3.0
   Area = 6.0
```

How Interface References Support Runtime Polymorphism

The mechanism operates in two stages, one at compile time and one at runtime, and both stages depend on the interface reference. At compile time, the variable s is declared with the interface type Shape. The compiler therefore allows only calls to methods that the interface declares — draw and area — and it checks the loop purely against the contract, without knowing or caring which concrete classes will ever appear in the array. This is why the same loop would compile unchanged if a fourth class, Pentagon, were added to the program next year. At runtime, the picture reverses: each element of the array is an object of a different concrete class, and when s.draw() executes, the Java Virtual Machine inspects the actual object that s currently refers to and dispatches the call to that object's class — Circle's draw on the first iteration, Rectangle's on the second, Triangle's on the third. This selection, made while the program runs and based on the object rather than the reference type, is called dynamic method dispatch or late binding, and it is the technical heart of runtime polymorphism. The interface reference is what makes both stages possible simultaneously: it is general enough at compile time to accept any implementing object, yet it carries no behaviour of its own, so the behaviour must come from the object at runtime.

Three consequences of this mechanism deserve emphasis, because they explain why professional Java is designed around interfaces. First, extensibility without modification: adding a new shape means writing one new class that implements Shape; the loop, and every other piece of code written against Shape, works with it immediately and untouched. Programmers call this the open-closed principle — open for extension, closed for modification — and it is achievable only because callers depend on the interface, not on concrete classes. Second, uniformity with diversity: a collection such as the array in the program, or an ArrayList of Shape, can hold a mixture of objects and process them with one elegant loop, while each object still performs its specialised behaviour, so the program gains simplicity at the calling side without losing variety at the performing side. Third, loose coupling and testability: because code depends on a contract rather than on a particular implementation, implementations can be replaced — a real database shape store by an in-memory one, a real payment gateway by a test double — without rewriting the dependent code. Two limits should also be stated honestly. Through a Shape reference only the interface's methods can be called; a Circle-specific method such as setRadius is invisible through s unless the reference is cast back to Circle, which sacrifices the generality. And polymorphism is not free of design duty: it works only when implementing classes genuinely honour the contract's meaning, so that any Shape can be trusted to draw itself and report its area sensibly wherever a Shape is expected.

Table 1: Compile-time and runtime roles in interface polymorphism

Stage	What is known	What is decided
Compile time	Only the interface type of the reference (Shape)	Which method calls are allowed: those declared in the

		interface
Object creation	The concrete class named after new (Circle, Rectangle, Triangle)	Which object, with which class and data, is stored in the reference
Runtime (call)	The actual class of the object currently referred to	Which class's method body executes — dynamic method dispatch
Adding a new class	The new class implements the same interface	Nothing in the existing calling code needs to change

Runtime Polymorphism Contrasted with Compile-Time Polymorphism

For completeness, runtime polymorphism through interfaces should be distinguished from its compile-time namesake, method overloading. In overloading, several methods in one class share a name but differ in their parameters, and the compiler selects among them while the program is being compiled, purely from the arguments written in the call; no object-dependent choice remains at runtime. In the overriding-based polymorphism of this answer, by contrast, the signature is identical in every implementing class, the compiler can do no more than verify that the call is permitted by the interface, and the choice among Circle, Rectangle and Triangle is postponed until the call actually executes and the real object is known. Examiners value this distinction because it shows that the word polymorphism covers two different binding times, and only the runtime form — the form interfaces support — delivers the extensibility, loose coupling and uniform handling of mixed collections demonstrated by the program above.

Conclusion

To conclude, polymorphism through an interface lets one reference type stand for many implementing classes while each object keeps its own behaviour. In the demonstration program, the interface Shape declares draw and area; Circle, Rectangle and Triangle each implement those methods in their own way; and a single loop calling draw and area through Shape references produces three different, correct behaviours. Interface references support runtime polymorphism because the compiler checks calls against the contract alone, while the Java Virtual Machine, at the moment of each call, dispatches to the method of the object's actual class — late binding based on the object, not the reference. The result is code that is uniform to write, diverse in behaviour, loosely coupled, and open to new implementing classes without modification, which is why designing to interfaces is one of the most valued habits in professional Java programming.

USMAN NOTES www.usmannotes.com