ALLAMA IQBAL OPEN UNIVERSITY

USMAN NOTES
www.usmannotes.com

Name:

Registration No.:

Course Code: 3411

Program: BS

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Programming Language-II (3411) • Level: BS

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 units of the course book, understand the concepts and the logic of every program, practise the programs on a computer, and then write the assignment in their own words. Copying this guide word for word is not allowed; the printed instructions on the assignment booklet also require answers in the student's own words. All Java programs in this guide are complete and self-contained: type them exactly, compile them with javac and run them to confirm the sample output before writing them in the assignment.

USMAN NOTES www.usmannotes.com

Question 1: Interfaces and Polymorphism (20 Marks)

Question as printed: Write a Java program to demonstrate the use of interfaces with multiple classes implementing the same interface. Explain why interfaces support polymorphism. (20)

Introduction

An interface is one of the most important tools in Java for building flexible and loosely connected programs. Where a class describes what an object is and how it works inside, an

interface describes only what an object can do, in the form of a list of method headings without any working code for them. Any number of unrelated classes may then sign this contract by implementing the interface, each supplying the promised methods in its own way. Because many different classes can stand behind one interface type, a single reference of the interface type can point to objects of many different classes, and one method call can produce many different behaviours — and that is exactly polymorphism. This answer explains what an interface is and how it is declared and implemented, presents a complete program in which three classes, Circle, Rectangle and Triangle, all implement one interface named Drawable, and then explains carefully why interfaces are such a natural and powerful support for polymorphism in Java.

What Is an Interface?

An interface in Java is a reference type, declared with the keyword interface, that contains abstract method declarations and, usually, constants. An abstract method in an interface has a name, parameters and a return type, but no body; every method of an interface is public and abstract by default, whether or not those words are written, and every field declared in an interface is public, static and final, that is, a shared constant. A class that wants to use the interface writes the keyword implements after its name and must then provide a public body for every method of the interface; if it leaves any method unimplemented, the class itself must be declared abstract. An interface cannot be used to create objects directly with new, because it has no complete methods and no constructors, but a reference of the interface type can hold an object of any class that implements it. Since Java 8, interfaces may also contain default

methods with a body and static methods, but the basic idea remains the same: an interface is a contract of behaviour that separates what is promised from how each class performs it.

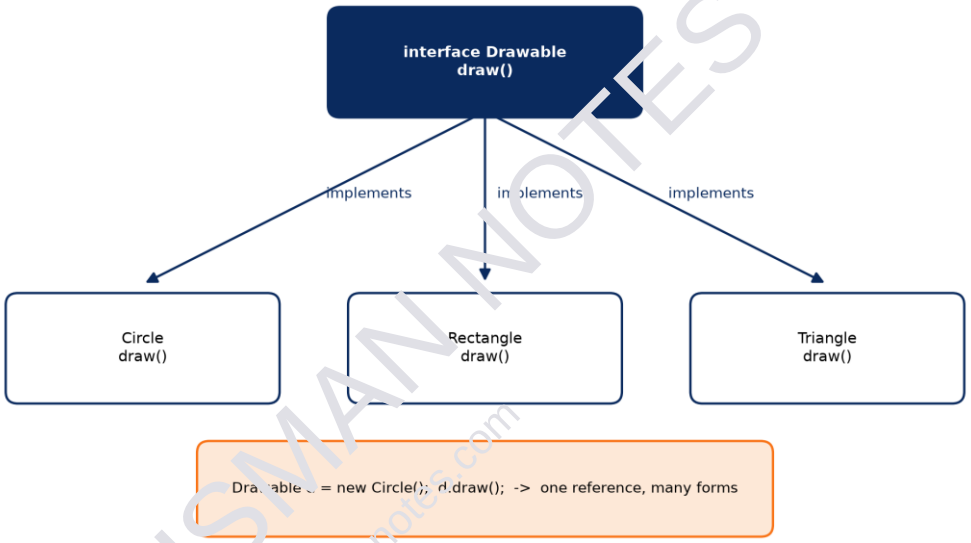


Figure 1: One interface, many implementations. Circle, Rectangle and Triangle each implement Drawable in their own way, and one Drawable reference can hold any of them.

The Java Program: One Interface, Three Implementing Classes

In the program below, the interface Drawable declares a single method, draw(). Three separate classes, Circle, Rectangle and Triangle, each implement Drawable and each provides its own version of draw(), printing a message that describes how that particular shape is drawn. The main class, InterfaceDemo, never works with the concrete classes directly for drawing: it declares a reference of the interface type, Drawable d, points it first to a Circle object, then to a Rectangle object and then to a Triangle object, and calls d.draw() each time. It then goes one step further and stores the three objects in an array of type Drawable and draws them all with one loop. Every call looks identical in the code, yet three different methods run — the program

is a small but complete demonstration of an interface at work. The file must be saved as InterfaceDemo.java to match the public class.

Java Program — InterfaceDemo.java

```
// The interface: a contract with one method and no body
interface Drawable {
   void draw();   // public and abstract by default
}

// First class implementing the interface
class Circle implements Drawable {
   public void draw() {
      System.out.println("Drawing a Circle: using a compass, one round
      line.");
   }
}

// Second class implementing the SAME interface
class Rectangle implements Drawable {
   public void draw() {
      System.out.println("Drawing a Rectangle: four straight lines, opposite
      sides equal.");
   }
}

// Third class implementing the SAME interface
```

USMAN NOTES www.usmannotes.com

```
class Triangle implements Drawable {
   public void draw() {
      System.out.println("Drawing a Triangle: three straight lines joined at
      corners.");
   }
}

public class InterfaceDemo {
   public static void main(String[] args) {
      // One interface reference, three different objects
      Drawable d;

      d = new Circle();
      d.draw();

      d = new Rectangle();
      d.draw();

      d = new Triangle();
      d.draw();

      System.out.println("--- Drawing all shapes through an array ---");
      Drawable[] shapes = { new Circle(), new Rectangle(), new Triangle() };
      for (int i = 0; i < shapes.length; i++) {
         shapes[i].draw();
```

USMAN NOTES www.usmannotes.com

```
        }
    }
}
```

Sample Output

```
Drawing a Circle: using a compass, one round line.
Drawing a Rectangle: four straight lines, opposite sides equal.
Drawing a Triangle: three straight lines joined at corners.
--- Drawing all shapes through an array ---
Drawing a Circle: using a compass, one round line.
Drawing a Rectangle: four straight lines, opposite sides equal.
Drawing a Triangle: three straight lines joined at corners.
```

USMAN NOTES www.usmannotes.com

Why Interfaces Support Polymorphism

Polymorphism means one name, many forms: the same call behaves differently according to the object it is made on. Interfaces support polymorphism for three connected reasons. The first reason is the common type: because Circle, Rectangle and Triangle all implement Drawable, they all belong to the interface type as well as to their own types, so one Drawable reference, variable, parameter or array can legally hold any of them. The second reason is dynamic method dispatch: when d.draw() is executed, the Java Virtual Machine does not decide from the reference type Drawable, which has no body for draw() at all; it looks at the actual object stored in d at that moment and runs the draw() method of that object class. Changing the object therefore changes the behaviour of the same written call, which is polymorphism in

its purest form. The third reason is loose coupling: code written against the interface, such as the loop over the shapes array, does not mention Circle, Rectangle or Triangle at all, so tomorrow a fourth class, Pentagon, can implement Drawable and join the array without changing one line of the drawing code. This is why real frameworks are designed around interfaces: the interface fixes the conversation, while unlimited classes can take part in it.

Table 1: Interface and abstract class — the two tools of abstraction

Feature	Interface	Abstract Class
Purpose	Declares what a class can do (a contract of behaviour)	Declares what a class is, with partial implementation
Methods	Abstract by default; default and static methods allowed since Java 8	Both abstract and complete methods allowed
Fields	Only constants: public, static and final	Normal instance variables allowed
How a class uses it	With the keyword implements; a class may implement many interfaces	With the keyword extends; a class may extend only one class
Objects	Cannot be instantiated	Cannot be instantiated
Multiple inheritance of type	Yes — this is how Java supports it safely	No

USMAN NOTES www.usmannotes.com

Explanation of the Program

A few points in the program deserve attention. Each implementing class writes public before its draw() method, because an interface method is public, and Java does not allow an implementation to reduce the visibility of a method it promises: forgetting the word public is the most common beginner error in this topic and produces a compile-time error. The classes Circle, Rectangle and Triangle are not related to each other by inheritance at all — they share no parent except Object — yet the interface unites them into one usable family, which shows that interfaces create a relationship based on behaviour, not on ancestry. The reassignment of the single reference d to three different objects, with three different results from the same call, is the direct evidence of polymorphism, and the final array loop shows the practical payoff: a whole collection of different shapes is processed by four lines of code that will never need to change as new shapes are added.

Conclusion

To conclude, an interface is a contract of abstract methods that any class may implement, providing its own bodies for the promised methods while remaining free in all other respects. Because many classes can implement one interface, the interface type becomes a common doorway through which all of them can be used, and because the Java Virtual Machine selects the method of the actual object at run time, one call such as d.draw() shows many forms — which is precisely polymorphism. The program above, in which Circle, Rectangle and Triangle implement the single interface Drawable and are drawn through one reference and one array, demonstrates both halves of the question completely: the mechanics of interfaces with

multiple implementing classes, and the reason interfaces and polymorphism belong together in every well-designed Java program.

Question 2: Garbage Collection and Automatic Memory Management in Java (20 Marks)

Question as printed: Discuss the concept of garbage collection in Java. Write a Java program that explicitly requests garbage collection and describes how Java manages memory automatically. (20)

Introduction

Memory is a limited resource, and every object a program creates occupies some of it. In older languages such as C and C++, the programmer who allocated memory also had to release it by hand, and forgetting to do so caused memory leaks, while releasing it too early caused crashes — two of the most common and most dangerous errors in software. Java removed this entire burden from the programmer by managing memory automatically, and the heart of that system is the garbage collector: a background part of the Java Virtual Machine that finds objects which are no longer reachable and reclaims their memory so that it can be reused. This answer discusses the concept of garbage collection, explains how the JVM divides and manages memory, shows how an unreachable object arises, and presents a complete program that creates objects, makes them eligible for collection and explicitly requests garbage collection with System.gc(), before describing honestly what such a request does and does not guarantee.

How Java Manages Memory Automatically

Java program memory is divided mainly into two areas. The stack holds the frames of methods currently running: each frame stores the local variables and primitive values of one method call, and the frame is removed automatically the moment the method finishes, so stack memory needs no collector at all. The heap is the large shared area where all objects created with new are stored, together with their instance variables and arrays. References held in stack frames, in static variables and in other live objects point into the heap. An object is alive, or reachable, as long as some chain of references from these roots can still reach it, because the program could still use it. The moment the last reference to an object is removed — for example by assigning null to the only variable that pointed to it, or by letting that variable go out of scope — the object becomes unreachable garbage: no code in the program can ever touch it again, yet it still occupies heap space. Freeing exactly this kind of object, and only this kind, is the job of the garbage collector, which runs inside the JVM in a low-priority background thread, so that programs are cleaned continuously without the programmer writing a single line of memory-release code.

USMAN NOTES www.usmannotes.com

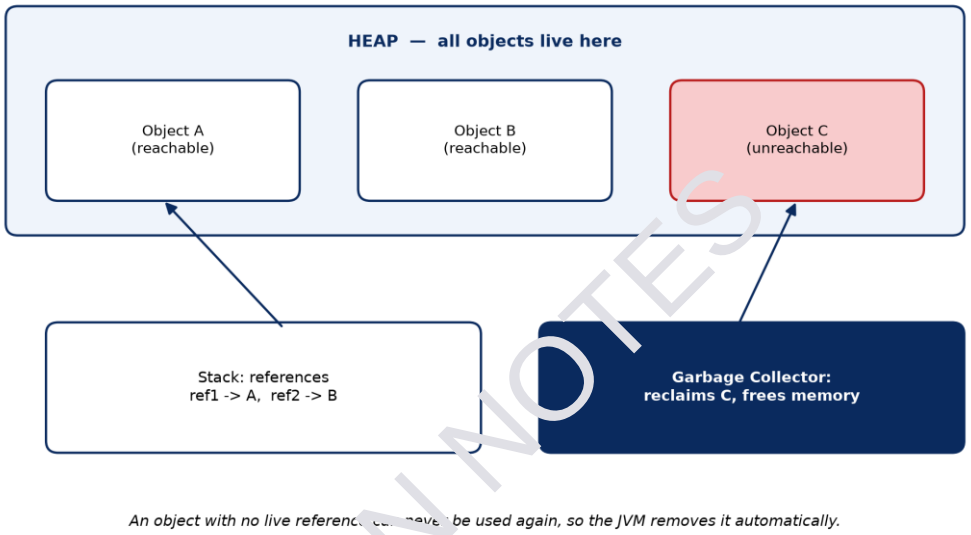


Figure 1: Memory management in Java. Objects A and B on the heap are still referenced from the stack and stay alive; object C has lost its last reference, so the garbage collector reclaims it.

How Garbage Collection Works

The collector works in two conceptual steps. In the marking step, it starts from the roots — the live references on the stacks of running threads and the static variables — and follows every reference, marking each object it can reach as alive. In the sweeping step, it reclaims the memory of all unmarked objects and makes that space available for new allocations, in many implementations also compacting the surviving objects so that free memory is not left in small scattered fragments. Modern JVM collectors refine this idea with generations, because experience shows that most objects die very young: the heap is divided into a young generation, where new objects are placed and cheap, frequent collections occur, and an old generation, where long-lived objects are promoted and collected less often. The details vary between collector implementations, but the promise to the programmer is constant:

unreachable objects will be removed sooner or later, heap space will be recycled, and the programmer never calls a free or delete operation. If the heap truly fills with live objects, the JVM throws OutOfMemoryError, but it never allows a program to use memory that has already been reclaimed for another object, which is the safety that manual languages could not guarantee.

Requesting Garbage Collection and the finalize() Method

Although collection is automatic, Java permits a program to request it in two equivalent ways: by calling the static method System.gc(), or by calling Runtime.getRuntime().gc(). Both are only requests or suggestions to the JVM: the JVM normally honours them promptly, but the language specification does not guarantee the exact moment of collection, because scheduling the collector is the JVM's decision. A second related feature is the finalize() method, defined in the Object class. In classic Java, if a class overrides finalize(), the collector calls it once on an object before reclaiming that object, giving the object a last chance to release resources, and textbooks use it to demonstrate that collection has happened. Students should know that finalize() is deprecated in modern Java because its timing is unpredictable, and real programs should close files and connections explicitly, for example with try-with-resources; nevertheless, for the purpose of this assignment it remains the clearest visible proof that an object has been collected, and the program below uses it for exactly that demonstration purpose. Setting references to null, as the program does, is called nullifying a reference and is the deliberate way to make an object eligible for collection.

The Java Program

The program defines a small class Demo whose objects carry a name, and whose finalize() method prints a message containing that name when the object is collected, so the class itself reports its own removal. The main method creates three objects, uses them, then makes two of them eligible by assigning null to their references and lets the third become unreachable when its local scope is passed, and finally calls System.gc() to request collection explicitly. The exact order and timing of the printed finalize messages depend on the JVM and may vary between runs; messages for both abandoned objects normally appear, and the explanatory point — that the abandoned objects are collected while referenced objects are not — is what the student should observe and write about.

Java Program — GCDemo.java

```
class Demo {
   private String name;

   public Demo(String name) {
      this.name = name;
   }

   public void show() {
      System.out.println("Object in use: " + name);
   }

   // Called by the garbage collector before the object is removed
```

```
    protected void finalize() {
        System.out.println("Garbage collected object: " + name);
    }
}

public class GCDemo {
    public static void main(String[] args) {
        Demo d1 = new Demo("Object-1");
        Demo d2 = new Demo("Object-2");
        Demo d3 = new Demo("Object-3");

        d1.show();
        d2.show();
        d3.show();

        // Make d1 and d2 unreachable: eligible for garbage collection
        d1 = null;
        d2 = null;
        // d3 is still referenced, so it will NOT be collected

        System.out.println("Requesting garbage collection...");
        System.gc();   // explicit request to the JVM

        System.out.println("Main method finished.");
    }
```

USMAN NOTES www.usmannotes.com

```
}
```

Sample Output

```
Object in use: Object-1
Object in use: Object-2
Object in use: Object-3
Requesting garbage collection...
Garbage collected object: Object-1
Garbage collected object: Object-2
Main method finished.
```

USMAN NOTES www.usmannotes.com

Explanation of the Program and Sample Output

The output tells the whole story of the question. The first three lines show all three objects alive and usable, because each has a live reference. After d1 = null and d2 = null are executed, Object-1 and Object-2 become unreachable: no variable in the program can ever reach them again, so they are garbage by definition, even though their bytes are still physically sitting on the heap. The call System.gc() then requests a collection, and the two finalize messages prove that the collector found and removed exactly those two objects. No message appears for Object-3, because the reference d3 still points to it at the end of main; this negative result is as instructive as the positive ones, because it demonstrates the reachability rule that decides life and death on the heap. The sample output shown above is a typical run; since a gc() call is a request rather than a command, a particular JVM may print the messages at a slightly different

point, and an honest student will mention this behaviour of the request in the assignment, because it is part of the correct concept.

Table 1: Key terms of Java memory management

Term	Meaning
Heap	The memory area where all objects and arrays are stored
Stack	The memory area holding method frames, local variables and references; freed automatically when a method returns
Reachable object	An object that can still be reached through live references; it is never collected
Unreachable object (garbage)	An object with no live reference; eligible for garbage collection
System.gc() / Runtime.getRuntime().gc()	Two ways to request garbage collection explicitly; the JVM decides the exact timing
finalize()	Classic method called once before an object is reclaimed; deprecated in modern Java, used here only for demonstration

Advantages and Limitations

Automatic garbage collection is one of the main reasons Java became a safe and productive language. It prevents memory leaks caused by forgotten releases, it prevents the opposite and worse error of using memory after freeing it, it removes a whole category of security faults, and

it lets the programmer concentrate on the logic of the program instead of bookkeeping. Programs in Java also cannot corrupt the heap by bad pointer arithmetic, because Java has no pointers to abuse. The limitations should also be stated fairly: the collector itself consumes processor time, a collection pause can briefly delay a running program, and automatic collection manages memory only — files, database connections and network sockets must still be closed by the programmer. A good Java programmer therefore cooperates with the collector: objects are kept alive only as long as needed, needlessly large collections are avoided, and explicit gc() requests are used rarely, mainly in tests and demonstrations such as this one, because the JVM normally knows the best time to collect.

Conclusion

To conclude, garbage collection is the automatic process by which the Java Virtual Machine finds unreachable objects on the heap and reclaims their memory for reuse, freeing the programmer from manual memory management and from its two classic errors, leaks and premature release. Java manages memory by dividing it between the stack, which is cleaned as methods return, and the heap, which the collector maintains by the simple and safe rule of reachability. A program may explicitly request collection through System.gc() or Runtime.getRuntime().gc(), as the GCDemo program does after nullifying two references, and the finalize messages of exactly those two objects, and of no other, demonstrate the rule in visible form.

Question 3: Reading Data from a Text File (20 Marks)

Question as printed: Write a Java program to read data from a text file and display it on the screen. Explain the classes used for file input operations. (20)

Introduction

Console input disappears when a program ends, but files store data permanently on disk, so almost every real program must read files: configuration files, saved records, marks sheets, logs and text documents. Java provides a rich set of classes for file input in the java.io package, organised as streams and readers that can be combined like pipes. For text files, the standard and easiest combination is the FileReader class, which opens the file and reads its characters, wrapped in a BufferedReader, which adds buffering and the extremely convenient readLine() method that returns one complete line at a time. This answer first explains the main classes used for file input operations, then presents a complete program that opens a text file named student.txt, reads it line by line and displays every line on the screen, handling the errors that file work can produce. The contents of the sample file and the output of the program are both shown, so that a student can create the same file, run the program and obtain the same result.

Classes Used for File Input Operations

The File class, in java.io, represents the name and properties of a file or folder on the disk: it can test whether a file exists, report its size and path, and create or delete files, but it does not itself read any contents. The actual reading is done by input classes, which fall into two families. Byte streams, whose names end in InputStream, read raw bytes and suit binary data such as images; the class FileInputStream opens a file as a byte stream. Character streams, whose names end in

Reader, read Unicode characters and are the correct choice for text; the class FileReader opens a file as a character stream, reading one character or an array of characters at a time. On top of a basic reader, Java offers filter readers that add abilities: BufferedReader wraps another reader, reads a large block of characters into memory at once for speed, and provides readLine(), which returns a whole line as a String, or null when the end of the file is reached — the standard signal that reading is complete. The InputStreamReader class is a bridge that converts a byte stream into a character stream, and the Scanner class of java.util, already used for keyboard input, can also be connected to a File object and read words, lines and numbers with methods such as nextLine() and nextInt(). Almost all of these classes signal problems with checked exceptions, mainly IOException and its subclass FileNotFoundException, which the program is forced to handle with try-catch or by declaring them.

Table 1: Main classes used for file input in Java

Class	Package	What It Does
File	java.io	Represents a file path; checks existence, size and other properties; does not read contents
FileInputStream	java.io	Opens a file as a byte input stream; suitable for binary data
FileReader	java.io	Opens a text file as a character input stream; reads characters
BufferedReader	java.io	Wraps a reader with a buffer for

		fast reading; provides readLine() for line-by-line input
InputStreamReader	java.io	Converts a byte stream into a character stream (bridge class)
Scanner	java.util	Reads tokens, words, lines and numbers from a file, keyboard or string with simple methods

How the Reading Process Works

Reading a file follows the same five steps in every program. First, the file is opened by creating a reader object and giving it the file name; if the file does not exist, this step throws FileNotFoundException at once. Second, the reader is wrapped in a BufferedReader so that reading is fast and line-based. Third, the program reads repeatedly in a loop, testing every result: the condition while ((line = br.readLine()) != null) both fetches the next line and stops the loop automatically at the end of the file, because readLine() returns null there. Fourth, each line is processed as it arrives — in this program it is simply printed, but in real software it might be searched, totalled or stored in objects. Fifth, the readers are closed with close(), which releases the file for other programs and flushes operating-system resources; forgetting to close files is a common fault that disciplined Java code avoids, nowadays often with the try-with-resources statement that closes automatically. Relative file names such as student.txt are searched in the working folder from which the program is run, so the sample file must be created in the same

folder as the compiled class, a practical point every student should verify when testing the program.

The Java Program

The program below is deliberately written in the clearest textbook style. It creates a FileReader for student.txt, wraps it in a BufferedReader, reads and prints every line, and closes both readers at the end. Two catch blocks protect it: FileNotFoundException reports a missing file in friendly words instead of crashing, and the general IOException catches any other reading problem. Before running the program, create the file student.txt with any text editor in the same folder and type the four lines shown under the heading Sample File. The program should be saved as ReadFileDemo.java.

Sample file student.txt (create this file in the same folder as the program): Ali Raza, Roll No 101, Class BS Computer Science. Sara Khan, Roll No 102, Class BS Computer Science. Ahmed Iqbal, Roll No 103, Class BS Computer Science. Fatima Noor, Roll No 104, Class BS Computer Science. Each of these four records is typed on its own separate line in the file, so the file contains exactly four lines, and the program is expected to display exactly those four lines.

Java Program — ReadFileDemo.java

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;


public class ReadFileDemo {
```

```
public static void main(String[] args) {
   try {
      // Open the text file for reading characters
      FileReader fr = new FileReader("student.txt");

      // Wrap it in a BufferedReader for fast, line-by-line reading
      BufferedReader br = new BufferedReader(fr);

      String line;
      System.out.println("Contents of student.txt:");

      // Read lines until readLine() returns null (end of file)
      while ((line = br.readLine()) != null) {
         System.out.println(line);
      }

      // Close the readers to release the file
      br.close();
      fr.close();

   } catch (FileNotFoundException e) {
      System.out.println("File not found: " + e.getMessage());
   } catch (IOException e) {
      System.out.println("Error reading the file: " + e.getMessage());
   }
```

```
    }
}
```

Sample Output

```
Contents of student.txt:
Ali Raza, Roll No 101, Class BS Computer Science
Sara Khan, Roll No 102, Class BS Computer Science
Ahmed Iqbal, Roll No 103, Class BS Computer Science
Fatima Noor, Roll No 104, Class BS Computer Science
```

USMAN NOTES www.usmannotes.com

Explanation of the Program

The whole program turns on three objects. The FileReader fr connects the program to the bytes of student.txt and converts them into characters; if the file is missing, its constructor throws FileNotFoundException immediately, and control jumps to the matching catch block, where a clear message is printed instead of a stack of errors. The BufferedReader br sits on top of fr: internally it reads large blocks of characters in one disk operation, which is far faster than reading the file character by character, and externally it offers readLine(). The while loop is the engine of the program: each test of the condition stores the next line in the variable line and checks it against null in the same expression, so a normal line is printed and the loop continues, while the null that marks the end of the file ends the loop cleanly without any counter or file-size calculation. The two close() calls then release the file in the reverse spirit of opening it. A student who wants to test the error handling should temporarily rename the file and run the

program again: the friendly File not found message that appears is itself part of the required behaviour of a well-written file program.

Conclusion

To conclude, Java reads text files through the character stream classes of java.io, principally FileReader for opening the file and BufferedReader for fast line-by-line reading with readLine(), supported by the File class for file information, FileInputStream and InputStreamReader for byte-based and bridge input, and Scanner for simple token-based reading. The complete pattern — open, wrap, read in a loop until null marks the end of file, process each line, close, and handle FileNotFoundException and IOException — is demonstrated in full by the ReadFileDemo program above, which displays the entire contents of student.txt on the screen exactly as stored. This same pattern, with different processing inside the loop, is the foundation of all file-reading work in Java.

Question 4: Wrapper Classes — Boxing and Unboxing (20 Marks)

Question as printed: Explain the concept of wrapper classes. Write a Java program to convert primitive data types into objects and convert them back into primitives. (20)

Introduction

Java works with two kinds of values. Primitive data types such as int, double, char and boolean are simple, fast values that are not objects and have no methods. Objects, on the other hand, belong to classes, can call methods, can be stored in collections and can hold the special value null. Many parts of Java, above all the collection classes such as ArrayList and HashMap, accept

only objects, so a program frequently needs to treat a simple number as an object — and that need is met by the wrapper classes. A wrapper class wraps one primitive value inside an object: Integer wraps an int, Double wraps a double, Character wraps a char, and so on, one wrapper for each of the eight primitive types. Converting a primitive into its wrapper object is called boxing, converting the object back into the primitive is called unboxing, and since Java 5 both conversions often happen automatically. This answer explains the wrapper classes and their uses, and gives a complete program that converts several primitives into objects and back again by both the explicit and the automatic routes.

The Eight Wrapper Classes

Every primitive type has a corresponding wrapper class, and all of them live in the java.lang package, so no import statement is ever needed for them. Byte wraps byte, Short wraps short, Integer wraps int, Long wraps long, Float wraps float, Double wraps double, Character wraps char and Boolean wraps boolean. Notice that two names are not simple capitalisations: int becomes Integer and char becomes Character, which beginners must remember carefully. A wrapper object is immutable, meaning its wrapped value cannot be changed after creation, just as a String cannot be changed. Besides storing the value, each wrapper class provides very useful services: constants such as Integer.MAX_VALUE and Integer.MIN_VALUE, conversion methods such as intValue() and doubleValue() that extract primitives, static parsing methods such as Integer.parseInt() that convert a numeric String into a primitive, and toString() methods that convert values into text. Wrapper classes also allow a value to be absent: an Integer variable can be null, while an int must always hold some number, a difference that matters greatly in databases and forms where data may be missing.

Table 1: Primitive types and their wrapper classes

Primitive Type	Wrapper Class	Example of Boxing	Example of Unboxing
byte	Byte	Byte b = Byte.valueOf((byte) 10);	byte x = b.byteValue();
short	Short	Short s = Short.valueOf((short) 200);	short x = s.shortValue();
int	Integer	Integer i = Integer.valueOf(42);	int x = i.intValue();
long	Long	Long l = Long.valueOf(1000L);	long x = l.longValue();
float	Float	Float f = Float.valueOf(3.5f);	float x = f.floatValue();
double	Double	Double d = Double.valueOf(12.75);	double x = d.doubleValue();
char	Character	Character c = Character.valueOf(ch);	char x = c.charValue();
boolean	Boolean	Boolean bl = Boolean.valueOf(true);	boolean x = bl.booleanValue();

USMAN NOTES www.usmannotes.com

Boxing, Unboxing, Autoboxing and Their Uses

Boxing is the conversion of a primitive value into its wrapper object, performed explicitly with the static valueOf method, for example Integer obj = Integer.valueOf(20). Unboxing is the reverse conversion, performed explicitly with a method such as intValue(), for example int n = obj.intValue(). Since Java 5, the compiler performs both conversions automatically where the context requires them, which is called autoboxing and automatic unboxing: writing Integer obj = 20 boxes the value without valueOf appearing in the code, and writing int n = obj unboxes it without intValue(). The main uses of wrapper classes follow from their object nature. Collections store objects only, so an ArrayList of marks must be an ArrayList of Integer, not of int, and autoboxing makes adding numbers to it painless. Wrapper classes are needed wherever a type parameter is required, because generics do not accept primitive types. They provide the parsing and conversion utilities that turn user-typed text into numbers, which almost every input-handling program uses. And they represent missing values through null in database and web applications. The cost of these benefits is that wrapper objects occupy more memory and are slightly slower than primitives, so plain arithmetic in tight loops should still use primitive types.

The Java Program

The program below demonstrates the complete two-way conversion for four representative types: int, double, char and boolean. In the first half, each primitive is boxed into its wrapper object twice over in spirit — the valueOf method shows the explicit route that the student must be able to write and explain — and the objects are printed, which automatically calls their

toString() methods. In the second half, each object is unboxed back into a primitive variable with the matching value method, and the primitives are printed and used in a small calculation to prove they are genuine primitives again. Two extra demonstrations round off the program: autoboxing and automatic unboxing are shown in single statements, and Integer.parseInt converts the string 100 into the number 100, the most common practical use of a wrapper class. The file is saved as WrapperDemo.java.

Java Program — WrapperDemo.java

```
public class WrapperDemo {
    public static void main(String[] args) {

        // ----- Primitive values -----
        int a = 20;
        double d = 12.5;
        char c = 'A';
        boolean flag = true;

        // ----- Boxing: primitives converted into objects -----
        Integer intObj = Integer.valueOf(a);
        Double doubleObj = Double.valueOf(d);
        Character charObj = Character.valueOf(c);
        Boolean boolObj = Boolean.valueOf(flag);

        System.out.println("Values stored in wrapper objects:");
        System.out.println("Integer object   : " + intObj);
```

```
System.out.println("Double object    : " + doubleObj);
System.out.println("Character object : " + charObj);
System.out.println("Boolean object   : " + boolObj);

// ----- Unboxing: objects converted back into primitives -----
int x = intObj.intValue();
double y = doubleObj.doubleValue();
char z = charObj.charValue();
boolean f = boolObj.booleanValue();

System.out.println("Values converted back to primitives:");
System.out.println("int value     : " + x);
System.out.println("double value  : " + y);
System.out.println("char value    : " + z);
System.out.println("boolean value : " + f);
System.out.println("Calculation with unboxed values: x + 5 = " + (x +
5));

// ----- Autoboxing and automatic unboxing -----
Integer autoObj = 50;        // autoboxing: int -> Integer
int autoValue = autoObj;     // auto-unboxing: Integer -> int
System.out.println("Autoboxed object : " + autoObj);
System.out.println("Auto-unboxed int : " + autoValue);

// ----- Useful wrapper utility: parsing a String -----
int parsed = Integer.parseInt("100");
```

```
        System.out.println("Parsed from String "100" : " + parsed);
    }
}
```

Sample Output

```
Values stored in wrapper objects:
Integer object   : 20
Double object    : 12.5
Character object : A
Boolean object   : true
Values converted back to primitives:
int value     : 20
double value  : 12.5
char value    : A
boolean value : true
Calculation with unboxed values: x + 5 = 25
Autoboxed object : 50
Auto-unboxed int : 50
Parsed from String "100" : 100
```

Explanation of the Program

The output is arranged in the same three stages as the code, so it can be explained stage by stage. First, the four wrapper objects print exactly the values that were boxed into them, because printing an object calls its toString() method, which every wrapper class overrides to

return the text form of its wrapped value; at this stage the values are objects, and could for example be added to an ArrayList. Second, the unboxed values print identically, but they are now true primitives: the proof is the calculation x + 5 = 25, because arithmetic is performed on primitives, and Java has quietly used the unboxed int. Third, the autoboxing lines show the modern short form of the same two conversions, with the compiler inserting valueOf and intValue() behind the scenes, and the final line shows parsing, where the wrapper class works as a utility library rather than as a container. One warning belongs in every explanation of unboxing: unboxing a wrapper that holds null throws a NullPointerException, because a primitive cannot represent nothing, so programs that accept possibly-null wrapper values must test them before unboxing.

Conclusion

To conclude, wrapper classes are the object counterparts of the eight primitive types — Byte, Short, Integer, Long, Float, Double, Character and Boolean, all in java.lang — and they exist so that primitive values can be used wherever objects are required, especially in collections and generics, and so that useful constants, parsing methods and the possibility of null come with them. Converting a primitive into a wrapper object is boxing, converting it back is unboxing, and both happen automatically through autoboxing in modern code. The program above performs the full round trip for int, double, char and boolean values — boxing with valueOf, unboxing with intValue, doubleValue, charValue and booleanValue, plus autoboxing and parsing — which is a complete practical demonstration of the concept asked in the question.

Question 5: The HashMap Class — Key-Value Storage (20 Marks)

Question as printed: Write a Java program to demonstrate the use of the HashMap class. Insert key–value pairs, retrieve values, and display the entire map. (20)

Introduction

Most collections store values in a sequence, reached by position, but many real problems store information in pairs: a roll number with a student name, a country with its capital, a product code with its price. For such paired data Java provides the Map family, and its most widely used member is the HashMap class of the java.util package. A HashMap stores entries as key-value pairs, in which every key is unique and points to exactly one value, and it finds values by key extremely quickly because it places each entry into a bucket chosen by the hash code of its key. This answer explains the HashMap class and its most important methods, and presents a complete program that creates a HashMap of student roll numbers and names, inserts several pairs, retrieves individual values by key, checks and measures the map, and finally displays the entire map in two different ways, followed by output and explanation.

What Is a HashMap?

A HashMap is declared with two type parameters, the type of the keys and the type of the values, for example HashMap<Integer, String> students = new HashMap<>(); creates a map whose keys are Integer objects and whose values are Strings. Keys must be unique: if put() is called with a key that already exists, the old value is quietly replaced by the new one, and the size of the map does not increase. Values, by contrast, may repeat freely, and a HashMap permits one null key and any number of null values. Internally, when an entry is inserted, the

hashCode() of the key is computed and converted into the index of a bucket, a slot in an internal array, where the entry is stored; when a value is requested by key, the same calculation leads straight to the correct bucket, so searching does not examine the other entries at all. This is why HashMap operations are described as very fast, on average constant time, and why HashMap is preferred over a list whenever data is naturally looked up by a key. One property must be remembered: HashMap does not promise any order of its entries, so the display order can differ from the insertion order and may vary between implementations.

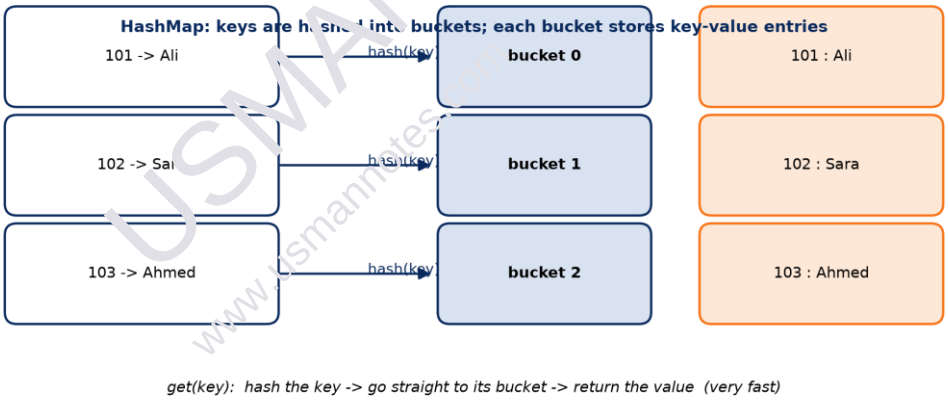


Figure 1: How a HashMap works. The hash of each key selects a bucket, and each bucket holds the key-value entry, so retrieval by key goes straight to the right bucket.

Important Methods of HashMap

The essential methods are few and easy to learn. The put(key, value) method inserts a pair, or replaces the value if the key is already present, and it is the only way entries enter the map. The get(key) method returns the value stored against the given key, or null if the key is absent, and

it is the retrieval operation the question specifically requires. The containsKey(key) and containsValue(value) methods test membership and are the safe way to check before retrieving, while size() reports how many pairs are stored and isEmpty() tests whether the map is empty. The remove(key) method deletes one pair, and clear() deletes all pairs. For displaying or processing the whole map, keySet() returns all the keys, values() returns all the values, and entrySet() returns all the pairs as Map.Entry objects each of which answers getKey() and getValue(); looping over the entry set is the standard, clearest way to print a complete map. Because HashMap stores objects, primitive keys such as roll numbers are autoboxed into Integer objects automatically when put() and get() are called, which connects this topic directly with the wrapper classes of the previous question.

Table 1: Important HashMap methods

Method	Purpose
put(key, value)	Inserts a key-value pair; replaces the value if the key already exists
get(key)	Returns the value stored against the key, or null if the key is absent
containsKey(key)	Returns true if the key exists in the map
containsValue(value)	Returns true if the value exists in the map
size()	Returns the number of key-value pairs stored
remove(key)	Removes the pair stored against the given key
keySet()	Returns a set of all keys
entrySet()	Returns a set of all key-value pairs (Map.Entry

	objects) for display and processing
clear()	Removes all pairs from the map

The Java Program

The program below creates a HashMap in which Integer roll numbers are keys and String student names are values. It inserts four pairs with put(), retrieves the value of one key with get() and prints it with a label, demonstrates containsKey() and size(), then displays the entire map twice: first in one line by printing the map object itself, which uses the standard {101=Ali Raza, ...} format produced by its toString() method, and second pair by pair through a loop over entrySet(), which is the format suitable for reports and assignments. Since HashMap guarantees no order, the entries in an actual run may appear in a different sequence from the sample output below; the set of pairs, which is what matters, will be exactly the same. The file must be saved as HashMapDemo.java.

Java Program — HashMapDemo.java

```
import java.util.HashMap;
import java.util.Map;


public class HashMapDemo {
   public static void main(String[] args) {


      // Create a HashMap: Integer keys, String values
      HashMap<Integer, String> students = new HashMap<>();
```

```
        // Insert key-value pairs
        students.put(101, "Ali Raza");
        students.put(102, "Sara Khan");
        students.put(103, "Ahmed Iqbal");
        students.put(104, "Fatima Noor");

        // Retrieve a value by its key
        String name = students.get(102);
        System.out.println("Student with roll number 102 is: " + name);

        // Check a key and count the pairs
        System.out.println("Is roll number 103 present? " +
        students.containsKey(103));
        System.out.println("Total students in the map: " + students.size());

        // Display the entire map in one line
        System.out.println("Complete map: " + students);

        // Display the entire map pair by pair
        System.out.println("Roll No -> Student Name");
        for (Map.Entry<Integer, String> entry : students.entrySet()) {
           System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
```

```
}
```

Sample Output

```
Student with roll number 102 is: Sara Khan
Is roll number 103 present? true
Total students in the map: 4
Complete map: {101=Ali Raza, 102=Sara Khan, 103=Ahmed Iqbal, 104=Fatima Noor}
Roll No -> Student Name
101 -> Ali Raza
102 -> Sara Khan
103 -> Ahmed Iqbal
104 -> Fatima Noor
```

USMAN NOTES www.usmannotes.com

Explanation of the Program

Each part of the output corresponds to one required operation. The four put() calls insert the four pairs; behind the scenes each roll number is autoboxed to an Integer, hashed, and placed in its bucket. The retrieval line shows the central idea of a map: the program did not search through the students in order, it simply asked get(102), and the hash of 102 led directly to Sara Khan, which is why maps are used for directories, indexes and records of every kind. The containsKey(103) test returns true and size() returns 4, confirming the state of the map before display. The one-line display works because HashMap overrides toString() to print its pairs in braces with equals signs joining keys and values. The final loop is the professional display method: entrySet() hands the program every pair as a Map.Entry, getKey() and getValue()

separate the two halves, and the arrow format makes a readable table. A useful experiment for the student is to add a second put(102, "New Name") before the display section: the size stays 4 and Sara Khan is replaced, which demonstrates the uniqueness of keys more convincingly than any definition.

Conclusion

To conclude, HashMap is the standard Java class for storing key-value pairs with unique keys, placing each entry in a bucket selected by the hash of its key so that insertion and retrieval by key are very fast, while promising no particular order of entries. Its working methods are put() for insertion, get() for retrieval, containsKey() and size() for checking, and entrySet() for complete display, all of which are demonstrated in the program above on a map of roll numbers and student names: four pairs are inserted, the value of key 102 is retrieved directly, and the entire map is displayed both in one line and pair by pair. Wherever information is naturally identified by a unique key, HashMap is the correct and efficient choice in Java.

USMAN NOTES www.usmannotes.com