CMR INSTITUTE OF TECHNOLOGY

Internal Assesment Test –I

Sub:	Object Oriented Programming using Java	Code:	MMC202

	Question Paper and Solution	Marks	OBE	
			CO	RBT
1)	Discuss Java Buzzwords and explain their significance.	10	CO1	L1
	OR			
2)	Explain Primitive and non-primitive datatypes in Java with an example.	10	CO1	L2
3)	Illustrate the uses of following keywords with suitable example 1. static 2. final	10	CO1	L3
	OR			
4)	What is Type casting and conversion explain with an example	10	CO1	L3
5)	Write a Java program to demonstrate constructor overloading and method overloading using a class named Student. Include at least two constructors and two overloaded methods.	10	CO1	L3
	OR			
6)	Write a Java class Calculator with overloaded methods add() to add: 1. Two integers 2. Two doubles 3. Three integers (Demonstrate overloading by calling all methods in the main() method)	10	CO1	L3
7)	What is inheritance in Java? Explain with an example including super and constructor chaining.	10	CO2	L2
	OR			
8)	Discuss method overriding with suitable Java code examples. How does it support run-time polymorphism?	10	CO2	L3
9)	Differentiate between String, StringBuffer, and StringBuilder with examples.	10	CO2	L2
	OR			
10)	Write a program to demonstrate string comparison, string length, and substring extraction. Explain the output.	10	CO2	L2

1) Discuss Java Buzzwords and explain their significance.

Java is a high-level, object-oriented, platform-independent programming language developed by Sun Microsystems (now owned by Oracle). It was created by James Gosling and others in 1991 and officially released in 1995. Originally designed for consumer electronics, Java became widely popular with the rise of the Internet due to its portability and security.

Java was inspired by C and C++ but eliminated many of their complex features. It provides a cleaner and more robust programming environment suitable for networked, distributed, and multi-threaded applications.

Key Features of Java (also known as Java Buzzwords)
Simple

- Easy to learn if you're familiar with C/C++.
- Removes complex and error-prone features like pointers and operator overloading.

Object-Oriented

- Everything in Java is treated as an object (except primitives).
- Uses principles like encapsulation, inheritance, and polymorphism.

Robust

- Emphasizes early error checking, garbage collection, and exception handling.
- Prevents memory leaks by managing memory automatically.

Platform-Independent (Portable)

- Java programs are compiled into bytecode, which runs on any system with a Java Virtual Machine (JVM).
- “Write once, run anywhere” philosophy.

Secure

- Provides a secure execution environment (sandbox model).
- Prevents unauthorized access and malicious code execution.

Multithreaded

- Supports concurrent execution of two or more threads.
- Makes it easier to develop interactive and responsive applications.

Architecture-Neutral

- Bytecode is not tied to any specific machine architecture.
- Ensures long-term code stability.

Interpreted

- Java bytecode is interpreted by the JVM.
- Also supports Just-In-Time (JIT) compilation for improved performance.

High Performance

- JIT compiler optimizes code at runtime.
- Faster than purely interpreted languages, while maintaining portability.

Distributed

- Java supports network programming and protocols like TCP/IP.
- Includes features like Remote Method Invocation (RMI).

Dynamic

- Java programs carry extensive runtime information.
- Allows dynamic loading and linking of new classes.

2) Explain Primitive and non-primitive datatypes in Java with an example.

Java is a strongly typed language, meaning every variable must be declared with a data type. These types define the size and type of data a variable can store.

Types of Data Types in Java
Java data types are divided into two main categories:

1. Primitive Data Types (8 types)
These are built-in data types.

Type	Size	Description	Example
byte	1 byte	Whole numbers from -128 to 127	byte a = 10;
short	2 bytes	Whole numbers from -32,768 to 32,767	short s = 1000;
int	4 bytes	Common integer type	int x = 500;
long	8 bytes	Larger range of integers	long l = 100000L;
float	4 bytes	Decimal numbers (less precision)	float f = 3.14f;
double	8 bytes	Decimal numbers (high precision)	double d = 3.14159;
char	2 bytes	Single 16-bit Unicode character	char c = 'A';
boolean	1 bit	Only true or false	boolean b = true;

2. Non-Primitive Data Types

These include **objects, arrays, strings**, and user-defined classes.

Type	Description	Example
String	Sequence of characters	String name = "Java";
Array	Collection of similar data types	int[] arr = {1, 2, 3};
Class	User-defined data structure	MyClass obj = new MyClass();
Interface	Abstract reference type	interface Printable {}

Example:

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

      // Primitive Data Types
      int age = 25;                // integer
      double height = 5.9;         // floating-point number
      char grade = 'A';            // character
      boolean isStudent = true;    // boolean

      // Non-Primitive Data Types
      String name = "Dhivya";      // String object
      int[] marks = {85, 90, 95};  // Array object

      // Output
      System.out.println("Name: " + name);
      System.out.println("Age: " + age);
      System.out.println("Height: " + height);
      System.out.println("Grade: " + grade);
      System.out.println("Is Student? " + isStudent);
      System.out.println("Marks: " + marks[0] + ", " + marks[1] + ", " + marks[2]);
   }
}
```

3) Illustrate the uses of following keywords with suitable example

1. static 2. final

i) Static

- To create a member that can be used by itself, without reference to a specific instance, precede its declaration with the keyword static."
- When a member is declared static, it can be accessed before any objects of its class are created.
- You can declare both methods and variables as static.

Key Properties of static Members:

- Static members belong to the class, not individual objects.
- All instances of the class share the same static variable.
- Static methods:
    - Can access only other static members.
    - Cannot use this or super.
- Static blocks can be used for static initialization.

Example:

```
class UseStatic {
   static int a = 3;
   static int b;

   static void meth(int x) {
      System.out.println("x = " + x);
      System.out.println("a = " + a);
      System.out.println("b = " + b);
   }

   static {
      System.out.println("Static block initialized.");
      b = a * 4;
   }

   public static void main(String[] args) {
      meth(42);
   }
}
```

The static block executes once when the class is loaded.
The meth() method and b are accessed using the class, not an object.
This is a demonstration of how static elements provide a class-wide scope.

ii) final

The keyword final in Java is used to restrict modification. It can be applied to:

1. Variables
2. Methods
3. Classes
4. Parameters

1. final with Variables

A field can be declared as final. Doing so prevents its contents from being modified, making it essentially a constant.

Example:

```
`final int FILE_NEW = 1;

 final int FILE_OPEN = 2;

final int FILE_SAVE = 3;

final int FILE_SAVEAS = 4;

final int FILE_QUIT = 5;
```

- *These constants can be used like: FILE_OPEN throughout the program.*
- *Java convention: final variables are usually written in UPPERCASE.*

2. final with Methods

To disallow a method from being overridden, specify final as a modifier. Methods declared as final cannot be overridden.

Example:

```
 class A {

    final void meth() {

        System.out.println("This is a final method.");

    }

}

class B extends A {

    // void meth() { System.out.println("Cannot override"); } // ❌ Not allowed

}
```

3. final with Classes

To prevent a class from being inherited, declare it as final. A final class cannot be subclassed

Example:

```
final class A {

    void show() {

        System.out.println("This is a final class.");

    }
```

```
}


class B extends A {

   // ❌ Error: Cannot inherit from final class

}
```

4. final with Method Parameters and Local Variables

Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned more than once.

Example:

```
void display(final int x) {

   // x = 5;  // ❌ This would cause a compile-time error

}
```

Summary:

Usage	Restriction Applied
final variable	Cannot be reassigned after initialization
final parameter	Cannot be changed inside the method
final method	Cannot be overridden in subclass
final class	Cannot be extended (no subclass allowed)

4) What is Type casting and conversion explain with an example

In Java, type conversion and type casting refer to changing a variable from one data type to another.

- ► Type Conversion (Widening or Implicit Casting)
- ► Type Casting (Narrowing or Explicit Casting)

long bigNum = num;
byte → short → int → long → float → double
↑ ↑ ↑ ↑
char (same order applies)

Type Casting (Narrowing or Explicit Casting)

```
double value = 99.99;
int intValue = (int) value;  // Narrowing conversion with casting
System.out.println(intValue); // Output: 99 (decimal truncated)
```

Concept	Type	Syntax	Data Loss	Example
Type Conversion	Widening (implicit)	Automatic	No	`int a = 10; long b = a;`
Type Casting	Narrowing (explicit)	With casting `(type)`	Yes (possible)	`double d = 5.5; int i = (int) d;`

Example:

```
package Practiceinclass;

public class TypeCoversion {

        public static void main(String[] args) {

    int a = 100;
    double d = a;
    System.out.println("Implicit conversion: " + d);

    double x = 123.456;
    int y = (int) x;
    System.out.println("Explicit casting: " + y);
      }

}
```

5) Write a Java program to demonstrate constructor overloading and method overloading using a class named Student. Include at least two constructors and two overloaded methods.

```
public class Student {
   String name;
   int age;
   int rollNo;

   // Constructor 1: No arguments
   Student() {
      name = "Unknown";
      age = 0;
      rollNo = 0;
   }

   // Constructor 2: With parameters
   Student(String name, int age, int rollNo) {
      this.name = name;
      this.age = age;
      this.rollNo = rollNo;
```

```
    }

    // Method 1: display student details
    void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Roll No: " + rollNo);
    }

    // Method 2: display message with custom label
    void display(String message) {
        System.out.println(message + ": " + name + ", Age " + age + ", Roll No " + rollNo);
    }

    // Main method to test
    public static void main(String[] args) {
        // Using default constructor
        Student student1 = new Student();
        student1.display(); // calling method 1
        student1.display("Student Info"); // calling method 2

        System.out.println("----------------------");

        // Using parameterized constructor
        Student student2 = new Student("John", 21, 101);
        student2.display(); // calling method 1
        student2.display("Details"); // calling method 2
    }
}
```

Output:

```
Name: Unknown
Age: 0
Roll No: 0
Student Info: Unknown, Age 0, Roll No 0
----------------------
Name: John
Age: 21
Roll No: 101
Details: Dhivya, Age 21, Roll No 101
```

6) Write a Java class Calculator with overloaded methods add() to add: 1. Two integers 2. Two doubles 3. Three integers (Demonstrate overloading by calling all methods in the main() method)

```
public class Calculator {

    // Method 1: Add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method 2: Add two doubles
    double add(double a, double b) {
        return a + b;
```

```
    }

    // Method 3: Add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Main method to test method overloading
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        // Calling overloaded methods
        int sum1 = calc.add(10, 20);            // two integers
        double sum2 = calc.add(5.5, 4.5);       // two doubles
        int sum3 = calc.add(1, 2, 3);           // three integers

        // Display results
        System.out.println("Sum of two integers: " + sum1);
        System.out.println("Sum of two doubles: " + sum2);
        System.out.println("Sum of three integers: " + sum3);
    }
}
```

Output:
Sum of two integers: 30
Sum of two doubles: 10.0
Sum of three integers: 6

7) What is inheritance in Java? Explain with an example including super and constructor chaining.

Inheritance is one of the core concepts of Object-Oriented Programming in Java. It allows a class (called the subclass or child class) to inherit fields and methods from another class (called the superclass or parent class).

This promotes code reusability, and method overriding allows customization of inherited behavior.

Key Concepts:

- extends: Keyword used to create a subclass.
- super: Refers to the parent class. Used to call the parent class's constructor or methods.
- Constructor Chaining: Using super() to call the constructor of the superclass.

Example:

```
// Superclass
class Person {
    String name;
    int age;

    // Constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Person constructor called");
    }
```

```
    void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

// Subclass
class Student extends Person {
    int rollNo;

    // Constructor chaining with super()
    Student(String name, int age, int rollNo) {
        super(name, age); // Calls Person constructor
        this.rollNo = rollNo;
        System.out.println("Student constructor called");
    }

    // Overriding method
    @Override
    void displayInfo() {
        super.displayInfo(); // Optional: Call parent method
        System.out.println("Roll No: " + rollNo);
    }
}

// Main class
public class InheritanceExample {
    public static void main(String[] args) {
        Student s = new Student("John", 21, 101);
        s.displayInfo();
    }
}
```

Output:

```
Person constructor called
Student constructor called
Name: John
Age: 21
Roll No: 101
```

8) Discuss method overriding with suitable Java code examples. How does it support run-time polymorphism?

Method Overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method in the child class must have:

- Same name
- Same return type
- Same parameter list

Key Rules of Overriding

- Only inherited methods can be overridden.
- The overridden method cannot have a more restrictive access modifier.
- Supports Run-time Polymorphism (dynamic method dispatch).

How it Supports Run-time Polymorphism

- In Java, method calls are resolved at runtime based on the object type, not the reference type. This is called dynamic dispatch.

Example:

```
// Superclass
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass 1
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

// Subclass 2
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

// Main class
public class OverrideExample {
    public static void main(String[] args) {
        Animal a;

        a = new Dog(); // Upcasting
        a.sound();     // Calls Dog's version – runtime decision

        a = new Cat(); // Upcasting
        a.sound();     // Calls Cat's version – runtime decision
    }
}
```

Output:

```
Dog barks
Cat meows
```

- Animal is the parent class.
- Dog and Cat override the sound() method.
- Reference variable a is of type Animal, but it holds objects of Dog and Cat.
- Which sound() method to execute is determined at runtime, not compile time. This is run-time polymorphism.

9) Differentiate between String, StringBuffer, and StringBuilder with examples.

In Java, String, StringBuffer, and StringBuilder are used to store and manipulate sequences of characters. However, they differ in terms of mutability**,** thread safety, and performance**.**

1. String

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

Performance: Modifying strings repeatedly creates new objects, making it slower.

Thread Safety: Safe in multithreaded environments due to immutability.

Example:

String str = "Hello";

str.concat(" World");

System.out.println(str); // Output: Hello

2. StringBuffer

Mutable: The contents of the string can be changed.

Thread Safe: All methods are synchronized, so it's safe in multithreaded programs.

Performance: Slower than StringBuilder due to synchronization.

Example:

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World");

System.out.println(sb); // Output: Hello World

3. StringBuilder

Mutable: Just like StringBuffer, its contents can be modified.

Not Thread Safe: Not synchronized, so not safe for multithreaded access.

Performance: Faster than StringBuffer and String in single-threaded scenarios.

Example:

StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");

System.out.println(sb); // Output: Hello World

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread-safe	Yes	Yes	No
Performance	Slow	Medium	Fast
Use Case	Constant text	Multithreading	Single-threading

10) Write a program to demonstrate string comparison, string length, and substring extraction. Explain the output.

Java program that demonstrates the following string operations:

1. String Comparison
2. String Length
3. Substring Extraction

Example:

```
public class StringOperations {

    public static void main(String[] args) {

        // Declare strings

        String str1 = "JavaProgramming";

        String str2 = "JavaProgramming";

        String str3 = "javaProgramming";

        // 1. String Comparison

        System.out.println("str1 equals str2: " + str1.equals(str2));          // true

        System.out.println("str1 equals str3: " + str1.equals(str3));          // false (case-sensitive)

        System.out.println("str1 equalsIgnoreCase str3: " + str1.equalsIgnoreCase(str3)); // true

        // 2. String Length

        System.out.println("Length of str1: " + str1.length());  // 15

        // 3. Substring Extraction

        String sub1 = str1.substring(0, 4);  // "Java"
```

```
        String sub2 = str1.substring(4);     // "Programming"

        System.out.println("Substring (0,4): " + sub1);

        System.out.println("Substring (4 to end): " + sub2);

    }

}
```

Output:

str1 equals str2: true

str1 equals str3: false

str1 equalsIgnoreCase str3: true

Length of str1: 15

Substring (0,4): Java

Substring (4 to end): Programming

Output Explanation:

1. **String Comparison**:
    - equals() checks if contents are the same with case sensitivity.
    - equalsIgnoreCase() ignores case differences.
2. **String Length**:
    - length() returns the number of characters in the string.
3. **Substring Extraction**:
    - substring(start, end) extracts characters from start index (inclusive) to end (exclusive).
    - substring(start) extracts from start to the end of the string.