black N White

BLACK N WHITE

Learn Today Lead Tomorrow

jag...

NAME	
ROLL NUMBER	
SEMESTER	II
COURSE CODE	DCA6207
COURSE NAME	OBJECT ORIENTED PROGRAMMING USING JAVA

SET-I

Q.1) Explain the features of Java.

Answer .:-

Features of Java

Java is a high-level, object-oriented programming language developed by Sun Microsystems (now owned by Oracle) in 1995. It has gained immense popularity among developers due to its simplicity, reliability, and portability. Below are the key features of Java that make it one of the most widely used programming languages in the world.

1. Simple

Java is designed to be easy to learn and use. It eliminates complex features found in other languages like C and C++, such as pointers and operator overloading. Its syntax is clear and concise, making it beginner-friendly and easy to maintain.

2. Object-Oriented

Java is a purely object-oriented language. It revolves around the concept of objects and classes, which helps in organizing complex programs, promoting code reusability, and improving maintainability. Everything in Java is treated as an object, which makes it ideal for building modular and scalable software.

3. Platform Independent

One of the most significant advantages of Java is its platform independence. Java programs are compiled into bytecode by the Java compiler. This bytecode can be executed on any system that has the Java Virtual Machine (JVM), regardless of the underlying hardware and operating system. This feature is often referred to as **"Write Once, Run Anywhere" (WORA).**

4. Secure

Java provides a high level of security. It doesn't use pointers, which helps prevent unauthorized access to memory. Java also runs programs within a secure sandbox environment, and its bytecode is verified before execution. Features like exception handling, garbage collection, and automatic memory management further enhance security.

5. Robust

Java is a robust language because it emphasizes early error checking, strong memory management, and exception handling. It has features like automatic garbage collection and type checking, which make applications more stable and less prone to crashes.

6. Multithreaded

Java supports multithreading, allowing programs to perform multiple tasks simultaneously. This is particularly useful in developing interactive and high-performance applications like games, multimedia, and web servers.

7. Portable

Java is highly portable due to its platform-independent nature and standardized size of primitive data types. Java code written on one system can run on any other system without modification, provided the JVM is installed.

8. High Performance

Although Java is an interpreted language, it offers high performance through Just-In-Time (JIT) compilers. JIT compilers convert bytecode into native machine code at runtime, which significantly improves the speed of execution.

9. Distributed

Java is designed to build distributed applications. It supports Remote Method Invocation (RMI) and Enterprise JavaBeans (EJB) that enable Java programs to communicate across a network and access resources remotely.

10. Dynamic

Java is a dynamic language that can adapt to an evolving environment. It supports dynamic loading of classes, meaning classes can be loaded at runtime as needed. This flexibility allows developers to update or expand applications easily.

formatted for an assignment or as a PDF/Word file!

Q.2) What are the different types of constructors in Java? Explain each type with code examples.

Answer .

In Java, **constructors** are special methods used to initialize objects. They have the same name as the class and do not have a return type. There are mainly **three types of constructors** in Java:

1. Default Constructor

A default constructor is automatically created by the Java compiler if no constructor is explicitly defined in the class. It does not take any arguments and simply assigns default values to the object.

Example:

```
class Student {
   int id;
   String name;

   // Default constructor
   Student() {
      id = 1;
      name = "John";
   }

   void display() {
      System.out.println(id + " " + name);
   }

   public static void main(String[] args) {
      Student s1 = new Student();
      s1.display();
```

```
    }
}
```
Output:

```
1 John
```

2. Parameterized Constructor

This constructor accepts arguments and is used to initialize an object with custom values at the time of creation.

Example:

```
class Student {
    int id;
    String name;

    // Parameterized constructor
    Student(int i, String n) {
        id = i;
        name = n;
    }

    void display() {
        System.out.println(id + " " + name);
    }

    public static void main(String[] args) {
        Student s1 = new Student(101, "Alice");
        Student s2 = new Student(102, "Bob");
        s1.display();
        s2.display();
    }
}
```
Output:

```
101 Alice
102 Bob
```

3. Copy Constructor

A copy constructor is used to create a new object by copying the fields of another object. Java does not provide a built-in copy constructor, but you can define it manually.

Example:

```
class Student {
    int id;
    String name;

    // Parameterized constructor
    Student(int i, String n) {
        id = i;
        name = n;
    }

    // Copy constructor
    Student(Student s) {
        id = s.id;
        name = s.name;
    }

    void display() {
        System.out.println(id + " " + name);
    }

    public static void main(String[] args) {
        Student s1 = new Student(201, "Ravi");
        Student s2 = new Student(s1);  // using copy constructor
        s1.display();
        s2.display();
    }
}
```

Output:

```
201 Ravi

201 Ravi
```

Summary Table:

Constructor Type	**Parameters**	**Created Automatically**	**Purpose**
Default Constructor	No	Yes (if no constructor)	Assigns default values
Parameterized Constructor	Yes	No	Assigns user-defined values
Copy Constructor	Yes (Object)	No	Creates a duplicate of an object

Q.3.a) What are the different types of control statements?

Answer .:-

Different Types of Control Statements in Java

Control statements in Java are used to control the flow of execution in a program. These statements help in making decisions, repeating tasks, or jumping to another part of the code. Java provides three main types of control statements:

1. Conditional (Decision-making) Statements

These statements allow the program to make decisions based on conditions.

- **if statement**: Executes a block of code if the condition is true.

```
if (a > b) {
   System.out.println("A is greater");
}
```

if-else statement: Executes one block if the condition is true, another if it is false.

```
if (a > b) {
   System.out.println("A is greater");
} else {
   System.out.println("B is greater");
}
```

- **else-if ladder**: Checks multiple conditions in sequence.
- **switch statement**: Selects a block to execute from many options.

```
switch (day) {
   case 1: System.out.println("Monday"); break;
   default: System.out.println("Invalid day");
}
```

2. Looping (Iteration) Statements

These are used when a task needs to be repeated multiple times.

- **for loop**
- **while loop**
- **do-while loop**

Example:

```
for (int i = 1; i <= 5; i++) {
   System.out.println(i);
}
```

3. Jump Statements

They are used to jump from one point in code to another.

- **break**: Exits a loop or switch.
- **continue**: Skips the current iteration.
- **return**: Exits from a method.

These control statements are essential for building logical and dynamic Java programs.

Q.3.b.) Write a Java program to find the sum of 1+3+5+…. for 10 terms in the series.

Answer .:-

```
public class OddSeriesSum {
    public static void main(String[] args) {
        int terms = 10;         // Number of terms in the series
        int sum = 0;            // Variable to hold the total sum
        int current = 1;        // First odd number in the series

        System.out.print("Series: ");
        for (int i = 0; i < terms; i++) {
            System.out.print(current);
            if (i < terms - 1) {
                System.out.print(" + ");
            }
            sum += current;
            current += 2;  // Move to the next odd number
        }

        System.out.println("\nTotal sum of the series: " + sum);
    }
}
```

Out put

```
Series: 1 + 3 + 5 + 7 + 9 + 11 + 13 + 15 + 17 + 19
Total sum of the series: 100
```

SET-II

Q.4.a) How do you implement inheritance in Java?

Answer .:-

Inheritance in Java is a concept where one class (called a **subclass** or **child class**) acquires the properties and behaviors (fields and methods) of another class (called a **superclass** or **parent class**). It promotes **code reusability** and helps in building a hierarchical class structure.

Java supports **single**, **multilevel**, and **hierarchical** inheritance, but not **multiple inheritance** through classes (interfaces can be used for that).

Using the extends Keyword

In Java, the extends keyword is used to inherit from a class.

Example:

```
// Parent class
class Animal {
   void sound() {
      System.out.println("Animals make sounds");
   }
}
// Child class
class Dog extends Animal {
   void bark() {
      System.out.println("Dog barks");
   }
}
// Main class to test
public class InheritanceDemo {
   public static void main(String[] args) {
      Dog myDog = new Dog();
      myDog.sound();  // Inherited method
      myDog.bark();   // Child class method
   }
}
```

Explanation:

- Animal is the **parent class** with a method sound().
- Dog is the **child class** that extends Animal, so it can use both sound() and bark().
- In main(), an object of Dog is created, which can access methods from both classes.

Key Benefits of Inheritance:

- Reuse existing code.
- Improve structure and readability.
- Enables **polymorphism** (method overriding).

Q.4.b.) What are the rules for overriding a method in Java?

Answer .:-

Method overriding in Java occurs when a subclass provides a specific implementation of a method that is already defined in its parent class. It allows the subclass to define behavior that is specific to its type.
To successfully override a method, the following rules must be followed:

1. Same Method Signature

The method in the subclass must have the **same name**, **return type**, and **parameters** as the method in the superclass.

```
class Parent {
   void display() { }
}
class Child extends Parent {
   @Override
   void display() { }  // Correct override
}
```

2. Access Modifier Cannot Be More Restrictive

The access level of the overriding method **must not be more restrictive** than the overridden method.

// If parent has public, child cannot make it private or protected

3. Private Methods Cannot Be Overridden

A private method in the superclass is not visible to the subclass, so it **cannot be overridden**.

4. Final Methods Cannot Be Overridden

If a method is declared as final, it cannot be overridden in the subclass.

5. Static Methods Cannot Be Overridden

Static methods belong to the class, not the instance, so they **cannot be overridden**. You can **hide** them, not override.

6. Constructors Cannot Be Overridden

Constructors are not inherited, so they **cannot be overridden**.

7. Use of @Override Annotation (Optional but Recommended)

Using @Override helps the compiler verify that you're actually overriding a method.

Q.5.a.) What are the differences between an interface and an abstract class?

Answer .:-

Both **interfaces** and **abstract classes** are used to achieve abstraction in Java, but they have different rules and purposes.

1. Method Implementation

- **Interface**: Can only have abstract methods (until Java 7). From Java 8 onward, it can have default and static methods with body.
- **Abstract Class**: Can have both abstract methods and fully implemented methods.

```
interface Vehicle {
   void start(); // abstract
}
abstract class Car {
   abstract void drive(); // abstract
   void fuel() {         // implemented
      System.out.println("Fueling the car");
   }
}
```

2. Inheritance Type

- **Interface**: A class can **implement multiple interfaces**.
- **Abstract Class**: A class can **extend only one abstract class** (Java does not support multiple inheritance with classes).

3. Constructors

- **Interface**: Cannot have constructors.
- **Abstract Class**: Can have constructors, which are called when a subclass object is created.

4. Variables

- **Interface**: All variables are public static final by default (i.e., constants).
- **Abstract Class**: Can have instance variables with any access modifier (private, protected, public).

5. Use Case

- **Interface**: Best when multiple classes share only method declarations (capabilities).
- **Abstract Class**: Best when classes share common code and behavior.

Use **interfaces** when you want to define a contract for classes, and use **abstract classes** when you need partial implementation with shared behavior.
Let me know if you'd like a combined example using both in one program!

Q.5.b.) What is the difference between errors and exceptions?

Answer .:-

Both **errors** and **exceptions** are issues that arise during program execution, but they are handled differently and represent different kinds of problems.

1. Definition

- **Error**: Represents serious issues that occur **outside the control** of the program. These are mostly related to the environment in which the application runs.
- **Exception**: Represents problems that occur **within the program**. These can usually be **caught and handled** by the programmer.

2. Handling

- **Error**: Cannot be recovered from using try-catch blocks. These are generally not meant to be handled.
- **Exception**: Can be managed using try, catch, and finally blocks.

3. Examples

- **Error**:
  - OutOfMemoryError
  - StackOverflowError
  - VirtualMachineError
- **Exception**:
  - NullPointerException
  - ArithmeticException
  - ArrayIndexOutOfBoundsException

4. Inheritance

- Both **Error** and **Exception** classes extend the Throwable class.

```
• Throwable
   ├── Error
   └── Exception
```

5. Use Case

- **Use Error** for critical system failures that should not be caught.
- **Use Exception** for issues you expect and want to handle during execution.

Errors are serious and mostly **unrecoverable**, while exceptions are typically **recoverable** and can be handled gracefully in code.

Q.6) What are the different methods under DataInputStream and DataOutputStream?

Answer .:-

In Java, the DataInputStream and DataOutputStream classes are part of the java.io package. They are used for reading and writing **primitive data types** (like int, float, double, etc.) in a **machine-independent way**, ensuring that data can be stored and retrieved consistently across different platforms.

1. DataInputStream

DataInputStream is used to read primitive Java data types from an input stream in a portable way. It works with InputStream objects.

Constructor:

DataInputStream dis = new DataInputStream(new FileInputStream("data.txt"));

Common Methods:

Method	**Description**
int read()	Reads one byte and returns it as an integer.
boolean readBoolean()	Reads one input byte and returns true or false.
byte readByte()	Reads a signed 8-bit value.
char readChar()	Reads a 16-bit Unicode character.
double readDouble()	Reads a 64-bit double value.
float readFloat()	Reads a 32-bit float value.
int readInt()	Reads a 32-bit integer.
long readLong()	Reads a 64-bit long value.
short readShort()	Reads a 16-bit short value.
String readUTF()	Reads a string encoded using modified UTF-8.
int readUnsignedByte()	Reads an unsigned 8-bit byte.
int readUnsignedShort()	Reads an unsigned 16-bit short.
void close()	Closes the stream and releases resources.

2. DataOutputStream

DataOutputStream allows an application to write primitive Java data types to an output stream in a portable way.

Constructor:

DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.txt"));

Common Methods:

Method	**Description**
void write(int b)	Writes one byte.
void writeBoolean(boolean v)	Writes a boolean value.
void writeByte(int v)	Writes a byte.
void writeChar(int v)	Writes a char as a 2-byte value.
void writeDouble(double v)	Writes a 64-bit double.
void writeFloat(float v)	Writes a 32-bit float.
void writeInt(int v)	Writes a 32-bit integer.
void writeLong(long v)	Writes a 64-bit long.
void writeShort(int v)	Writes a 16-bit short.
void writeUTF(String s)	Writes a string in modified UTF-8 format.
void flush()	Flushes the stream to ensure all data is written.
void close()	Closes the output stream and frees resources.

DataInputStream and DataOutputStream are essential for reading and writing primitive data types in binary format. They are commonly used in file I/O, networking, and database applications where platform independence is critical. Using these classes ensures that data written by one Java program can be read accurately by another, regardless of machine architecture.