Classes vs. Objects in Java

1. Definition

- **Class:**
    - A class is a blueprint or template that defines the structure and behavior of objects. It specifies the properties (attributes) and methods (functions) that the objects created from it will have.
- **Object:**
    - An object is an instance of a class. It represents a specific entity with actual values for the properties defined in the class.

2. Purpose

- **Class:**
    - Classes are used to define the structure and behavior of objects. They encapsulate data and methods into a single unit, promoting reusability and organization.
- **Object:**
    - Objects are used to represent real-world entities. Each object contains state (attributes) and behavior (methods) defined by its class.

3. Memory Allocation

- **Class:**
    - When a class is defined, memory is not allocated until an object of that class is created. The class itself is stored in the method area of the Java Virtual Machine (JVM).
- **Object:**
    - When an object is created, memory is allocated for it in the heap. Each object has its own memory space for storing instance variables.

Summary of Differences

Feature	Class	Object
Definition	Blueprint for creating objects	Instance of a class
Purpose	Defines structure and behavior	Represents a specific entity
Memory Allocation	No memory allocated until an object is created	Memory allocated in the heap for each object

Syntax	Defined using the `class` keyword	Created using the `new` keyword
Encapsulation	Contains attributes and methods	Holds actual values for the attributes
Reusability	Promotes code reusability	Represents unique instances

Classes vs. Objects in a Banking Project

Class: `BankAccount`

- **Attributes:**
  - `accountNumber`: A unique identifier for the account.
  - `accountHolder`: The name of the account holder.
  - `balance`: The current balance in the account.
- **Methods:**
  - `deposit(double amount)`: Adds a specified amount to the account balance.
  - `withdraw(double amount)`: Deducts a specified amount from the account balance.
  - `getBalance()`: Returns the current balance.

Java Code Example

```
public class BankAccount {
    // Attributes
    private String accountNumber;
    private String accountHolder;
    private double balance;

    // Constructor
    public BankAccount(String accountNumber, String accountHolder,
double initialBalance) {
        this.accountNumber = accountNumber;
        this.accountHolder = accountHolder;
        this.balance = initialBalance;
    }

    // Method to deposit money
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println(amount + " deposited. New balance: " +
balance);
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }

    // Method to withdraw money
    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println(amount + " withdrawn. New balance: " +
balance);
        } else {
            System.out.println("Insufficient funds or invalid
withdrawal amount.");
        }
    }

    // Method to get current balance
    public double getBalance() {
        return balance;
    }

    // Method to display account details
    public void displayAccountInfo() {
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Account Holder: " + accountHolder);
        System.out.println("Current Balance: " + balance);
    }
}
```

Creating Objects

Now, let's create some bank account objects:

```
public class Main {
    public static void main(String[] args) {
        // Creating objects of BankAccount
        BankAccount johnsAccount = new BankAccount("123456789", "John
Doe", 1000.00);
        BankAccount janesAccount = new BankAccount("987654321", "Jane
Smith", 1500.00);

        // Using the objects
        johnsAccount.deposit(500);            // John deposits $500
        johnsAccount.withdraw(200);           // John withdraws $200
        System.out.println("John's balance: " +
johnsAccount.getBalance()); // Check John's balance

        janesAccount.withdraw(2000);          // Attempt to withdraw
more than the balance
        janesAccount.deposit(300);             // Jane deposits $300
        janesAccount.displayAccountInfo();     // Display Jane's
account details
    }
}
```

Explanation of the Example

1. **Class Definition:**
   - The BankAccount class contains attributes and methods that define the structure and behavior of a bank account.
2. **Creating Objects:**
   - In the Main class, we create two objects: johnsAccount and janesAccount. Each object represents a specific bank account with its unique data.
3. **Using the Objects:**
   - Methods are called on these objects to manipulate their states:
     - deposit() adds money to the account.

        - `withdraw()` deducts money from the account, checking for sufficient funds.
        - `getBalance()` retrieves the current balance.
        - `displayAccountInfo()` shows the account details.

Summary of Differences

- **Blueprint vs. Instance:**
    - **Class:** `BankAccount` acts as a blueprint defining the structure and behavior.
    - **Object:** `johnsAccount` and `janesAccount` are instances of the `BankAccount` class, each holding specific data.
- **Unique State vs. Shared Structure:**
    - **Class:** Defines the structure common to all bank accounts.
    - **Object:** Each object has its own values for account number, holder, and balance.
- **Method Operations:**
    - **Class Methods:** Defined operations applicable to all accounts, like `deposit()` and `withdraw()`.
    - **Object Methods:** Invoked on specific account objects to manipulate their unique states.

School Management System Example

In this system, we typically have several classes: `Student`, `Teacher`, `Course`, and `School`. Let's define these classes and explore their relationships and communication.

Class Definitions

1. **Student Class**
   - Represents a student in the school.

```
public class Student {
    private String name;
    private String studentId;

    public Student(String name, String studentId) {
        this.name = name;
        this.studentId = studentId;
    }

    public String getName() {
        return name;
    }

    public String getStudentId() {
        return studentId;
    }
}
```

2. **Teacher Class**
   - Represents a teacher in the school.

```
public class Teacher {
    private String name;
    private String teacherId;

    public Teacher(String name, String teacherId) {
        this.name = name;
        this.teacherId = teacherId;
    }
    public String getName() {
        return name;
    }

    public String getTeacherId() {
        return teacherId;
    }
}
```

3. **Course Class**
    - Represents a course that students can enroll in.

```
import java.util.ArrayList;
import java.util.List;

public class Course {
    private String courseName;
    private Teacher instructor;
    private List<Student> enrolledStudents;

    public Course(String courseName, Teacher instructor) {
        this.courseName = courseName;
        this.instructor = instructor;
        this.enrolledStudents = new ArrayList<>();
    }

    public void enrollStudent(Student student) {
        enrolledStudents.add(student);
        System.out.println(student.getName() + " has been
enrolled in " + courseName);
    }

    public void showEnrolledStudents() {
        System.out.println("Students enrolled in " + courseName +
":");
        for (Student student : enrolledStudents) {
            System.out.println("- " + student.getName());
        }
    }

    public Teacher getInstructor() {
        return instructor;
    }
}
```

4. **School Class**
    - Manages the courses and the students.

```
import java.util.ArrayList;
import java.util.List;

public class School {
    private String schoolName;
    private List<Course> courses;

    public School(String schoolName) {
        this.schoolName = schoolName;
        this.courses = new ArrayList<>();
    }

    public void addCourse(Course course) {
        courses.add(course);
        System.out.println("Course " + course.courseName + " has
been added to " + schoolName);
    }

    public void showCourses() {
        System.out.println("Courses offered by " + schoolName +
":");
        for (Course course : courses) {
            System.out.println("- " + course.courseName + "
(Instructor: " + course.getInstructor().getName() + ")");
        }
    }
}
```

Object Relationships

1. **Association:**
    - A `Course` is associated with a `Teacher` (one-to-one relationship).
    - A `Course` can have multiple `Student` objects enrolled (one-to-many relationship).
2. **Aggregation:**
    - The `School` aggregates `Course` objects. The school can exist independently of the courses it offers.

Example Usage

Let’s demonstrate how these objects interact in a main class:

```
public class Main {
    public static void main(String[] args) {
        // Create a school
        School school = new School("Sunnyvale High School");

        // Create teachers
        Teacher teacher1 = new Teacher("Mr. Smith", "T001");
        Teacher teacher2 = new Teacher("Ms. Johnson", "T002");

        // Create courses
        Course mathCourse = new Course("Mathematics", teacher1);
        Course scienceCourse = new Course("Science", teacher2);

        // Add courses to the school
        school.addCourse(mathCourse);
        school.addCourse(scienceCourse);

        // Create students
        Student student1 = new Student("Alice", "S001");
        Student student2 = new Student("Bob", "S002");
        // Enroll students in courses
        mathCourse.enrollStudent(student1); // Alice enrolls in
Mathematics
        mathCourse.enrollStudent(student2); // Bob enrolls in
Mathematics
        scienceCourse.enrollStudent(student1); // Alice enrolls
in Science

        // Show enrolled students for each course
        mathCourse.showEnrolledStudents(); // Show students in
Mathematics
        scienceCourse.showEnrolledStudents(); // Show students in
Science

        // Show all courses offered by the school
        school.showCourses();
    }
}
```

Summary of Object Relationships and Communication in the School System

- **Association:**
  - `Course` is associated with a `Teacher` and can have many `Student` objects enrolled.
- **Aggregation:**
  - `School` holds and manages multiple `Course` objects but can exist independently.
- **Communication:**
  - Objects communicate through method calls, such as enrolling students in courses or showing enrolled students.