Q1. Write a simple Banking System program by using OOPs concept where you can get account Holder name balance, etc.:

```
class BankAccount {
   private String accountHolder;
   private double balance;

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

   public String getAccountHolder() {
      return accountHolder;
   }

   public double getBalance() {
      return balance;
   }

   public void deposit(double amount) {
      balance += amount;
      System.out.println("Amount deposited: " + amount);
   }

   public void withdraw(double amount) {
      if (amount <= balance) {
         balance -= amount;
         System.out.println("Amount withdrawn: " + amount);
      } else {
         System.out.println("Insufficient balance!");
      }
   }
}

public class BankingSystem {
   public static void main(String[] args) {
      BankAccount account = new BankAccount("John Doe", 1000.0);
      System.out.println("Account Holder: " + account.getAccountHolder());
      System.out.println("Balance: " + account.getBalance());

      account.deposit(500.0);
      account.withdraw(200.0);

      System.out.println("Updated Balance: " + account.getBalance());
   }
}
```

Q2. Write a Program where you inherit a method from a parent class and show method Overridden Concept:

```
class Parent {
   public void display() {
      System.out.println("This is the parent class.");
   }
}

class Child extends Parent {
   @Override
   public void display() {
      System.out.println("This is the child class.");
   }
}

public class MethodOverriding {
   public static void main(String[] args) {
      Parent parentObj = new Parent();
      parentObj.display(); // Output: This is the parent class.

      Child childObj = new Child();
      childObj.display(); // Output: This is the child class.

      Parent parentRef = new Child();
      parentRef.display(); // Output: This is the child class.
   }
}
```

Q3. Write a program to show runtime polymorphism in Java:

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

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

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

public class RuntimePolymorphism {
   public static void main(String[] args) {
      Animal animal = new Animal();
      animal.sound(); // Output: Animal makes a sound.

      Animal dog = new Dog();
```

```
        dog.sound(); // Output: Dog barks.

        Animal cat = new Cat();
        cat.sound(); // Output: Cat meows.
    }
}
```

Q4. Write a program to show compile-time polymorphism in Java:

```
public class CompileTimePolymorphism {
    public static int add(int a, int b) {
        return a + b;
    }

    public static int add(int a, int b, int c) {
        return a + b + c;
    }

    public static void main(String[] args) {
        int sum1 = add(2, 3);
        System.out.println("Sum1: " + sum1); // Output: Sum1: 5

        int sum2 = add(2, 3, 4);
        System.out.println("Sum2: " + sum2); // Output: Sum2: 9
    }
}
```

Q5. Achieve loose coupling in Java by using OOPs concept:

In Java, loose coupling can be achieved through the following OOPs concepts:

1. Abstraction: Encapsulate the implementation details of classes and provide only necessary interfaces or abstract classes for interaction. This allows objects to be decoupled from the specifics of their implementation.
2. Inheritance: Use inheritance to define a general base class or interface and let the derived classes provide specific implementations. This allows objects to be used interchangeably through their common base type, promoting loose coupling.
3. Polymorphism: Program to interfaces or abstract classes rather than concrete implementations. This allows different implementations to be used interchangeably based on their common interface, reducing dependencies on specific implementations.
4. Dependency Injection: Instead of creating dependencies within a class, use dependency injection to provide the necessary dependencies from outside. This allows the class to be loosely coupled with its dependencies, as the dependencies can be easily replaced or modified without affecting the class.

Q6. What is the benefit of encapsulation in Java?

encapsulation enhances code modularity, improves data security and integrity, simplifies code usage, and promotes code reusability and maintainability.

Q7. Is Java a 100% Object-oriented Programming language? If no, why?

No, Java is not considered a 100% object-oriented programming language. Java supports both object-oriented programming (OOP) and procedural programming paradigms. While Java incorporates key features of OOP, such as classes, objects, inheritance, and polymorphism, it also includes non-object-oriented elements, such as

primitive data types and static methods.

Q8. What are the advantages of abstraction in Java?

Abstraction is an important concept in object-oriented programming that allows you to create simplified and more manageable representations of complex systems. The advantages of abstraction in Java include:

5. Simplifies Complexity: Abstraction helps in breaking down a complex system into smaller, more manageable components. By focusing on the essential features and hiding unnecessary details, abstraction simplifies the understanding and implementation of the system.
6. Enhances Modularity: Abstraction promotes modularity by encapsulating related data and behavior into classes. This modular design makes the codebase easier to understand, maintain, and extend. Changes to one module do not affect other modules as long as the public interface remains the same.
7. Provides Security and Control: Abstraction allows you to define the access level and visibility of data and methods in a class. By encapsulating data and providing limited access through methods, abstraction provides security and control over the data, preventing unauthorized modifications or access.
8. Promotes Code Reusability: Abstraction enables code reusability by creating abstract classes and interfaces. Abstract classes provide a common base for related classes, allowing them to inherit and reuse code. Interfaces define a contract that classes can implement, promoting code reuse across different implementations.
9. Supports Maintainability and Extensibility: Abstraction improves code maintainability by separating the high-level concepts and functionality from the low-level implementation details. This separation allows easier modification, bug fixing, and enhancements without affecting the entire system.

Q9. What is an abstraction explained with an example?
Abstraction is the process of representing complex systems by focusing on the essential features and hiding unnecessary details. It allows you to create abstract classes and interfaces that define the common behavior and properties of related objects.

```
public abstract class Account {
   private String accountNumber;
   private String accountHolderName;
   private double balance;

   public abstract void calculateInterest();

   public abstract void withdraw(double amount);

   // Common methods and getters/setters
}

public class SavingsAccount extends Account {
   // Implement abstract methods specific to SavingsAccount
}

public class CheckingAccount extends Account {
   // Implement abstract methods specific to CheckingAccount
}
```

Q10. What is the final class in Java?

In Java, a final class is a class that cannot be subclassed or inherited. It means that once a class is declared as final, it

cannot be extended by other classes. This is achieved by using the final keyword before the class declaration.

```
final class FinalClass {
    // Class implementation
}
```