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

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

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

    public String getAccountHolderName() {
        return accountHolderName;
    }

    public double getBalance() {
        return balance;
    }

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

    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            System.out.println(amount + " withdrawn successfully.");
        } 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 Name: " + account.getAccountHolderName());
        System.out.println("Account Balance: " + account.getBalance());

        account.deposit(500);
        account.withdraw(200);

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

Q2. Write a Program where you inherit a method from the parent class and show method Overridden Co ncept?

```
class Animal {
```

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

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

public class MethodOverridingExample {
    public static void main(String[] args) {
        Animal animal = new Animal();
        Dog dog = new Dog();

        animal.sound(); // Output: Animal makes a sound
        dog.sound(); // Output: Dog barks
    }
}
```

Q3. Write a program to show run-time polymorphism in Java?

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

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

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

public class RuntimePolymorphismExample {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();

        animal1.sound(); // Output: Dog barks
        animal2.sound(); // Output: Cat meows
    }
}
```

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

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

    static double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        int sum1 = add(2, 3);
        double sum2 = add(2.5, 3.5);

        System.out.println("Sum1: " + sum1); // Output: Sum1: 5
        System.out.println("Sum2: " + sum2); // Output: Sum2: 6.0
    }
}
```

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

Loose coupling is achieved in Java by using interfaces or abstract classes. By programming to interfaces or abstract classes, you can ensure that the classes are decoupled from each other, and the implementati on details can be hidden. This makes the code more flexible and maintainable.

Example:

```
interface PaymentMethod {
    void makePayment(double amount);
}

class CreditCardPayment implements PaymentMethod {
    @Override
    public void makePayment(double amount) {
        System.out.println("Credit card payment: $" + amount);
    }
}

class CashPayment implements PaymentMethod {
    @Override
    public void makePayment(double amount) {
        System.out.println("Cash payment: $" + amount);
    }
}

class ShoppingCart {
    private PaymentMethod paymentMethod;

    public void setPaymentMethod(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }
```

```
    public void checkout(double totalAmount) {
        paymentMethod.makePayment(totalAmount);
    }
}

public class LooseCouplingExample {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentMethod(new CreditCardPayment());
        cart.checkout(100.0); // Output: Credit card payment: $100.0

        cart.setPaymentMethod(new CashPayment());
        cart.checkout(50.0); // Output: Cash payment: $50.0
    }
}
```

Q6. What is the benefit of encapsulation in Java?

Encapsulation in Java is the process of hiding the internal details of an object and providing access to the object's properties and methods through public interfaces. The benefits of encapsulation include:

Data Hiding: Encapsulation allows you to hide the internal data of an object, preventing direct access to it from outside the class. This protects the data from unauthorized access and modifications.

Modularity: Encapsulation promotes modularity by allowing you to define separate parts of the class for d ata and behavior. This makes the code easier to understand, maintain, and debug.

Code Reusability: Encapsulation enables you to define reusable classes that can be used in different part s of the program without affecting other parts of the code.

Flexibility: Encapsulation allows you to change the internal implementation of a class without affecting its external interface. This provides flexibility and makes it easier to make changes to the code in the future.

Security: By hiding the internal details, encapsulation provides a level of security to the data and functiona lity of an object.

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

Java is often considered a mostly object-oriented programming language, but it is not 100% object-oriente d. The main reason is that Java supports primitive data types like int, float, char, etc., which are not object s. These primitive types are not instances of classes and do not have all the features of objects.

However, Java does provide wrapper classes for primitive data types (e.g., Integer, Float, Character) that allow you to treat primitives as objects. This concept is known as autoboxing and unboxing.

While Java is not 100% object-oriented due to the presence of primitive data types, it still follows most of t he principles and concepts of object-oriented programming, such as encapsulation, inheritance, polymorp hism, and abstraction.

Q8. What are the advantages of abstraction in Java?

Abstraction is one of the four fundamental principles of object-oriented programming in Java. The advantages of abstraction include:

Simplification of Complex Systems: Abstraction allows you to focus on the essential features of an object or system while hiding unnecessary details. It simplifies the design and implementation of complex systems.

Code Reusability: By using abstract classes and interfaces, you can define common properties and behaviors that can be shared among multiple classes. This promotes code reusability and reduces code duplication.

Flexibility and Maintainability: Abstraction provides a clear separation between the interface and implementation of a class. This allows you to change the internal implementation of a class without affecting its external interface. It improves the maintainability and flexibility of the code.

Security: Abstraction allows you to hide the internal implementation details of an object, providing a level of security to the data and methods of the class.

Encapsulation Support: Abstraction complements encapsulation by allowing you to define abstract methods and hide their implementation details. This helps in creating loosely coupled classes.

Design Abstraction: Abstraction helps in creating high-level design of software systems, making it easier to understand and manage complex systems.

Overall, abstraction is an essential concept in Java that promotes code organization, reusability, and maintainability. It helps in creating efficient, flexible, and scalable software systems.