Proxy Pattern

Susanna Ryttyläinen
Jingjing Yang

Agenda

- Proxy category and purpose
- Compare to other structural patterns

- General idea of proxy pattern
- Remote proxy to monitor gumball machines

- Use cases
- Drawbacks

- Exercise of protection proxy

Proxy Pattern in general

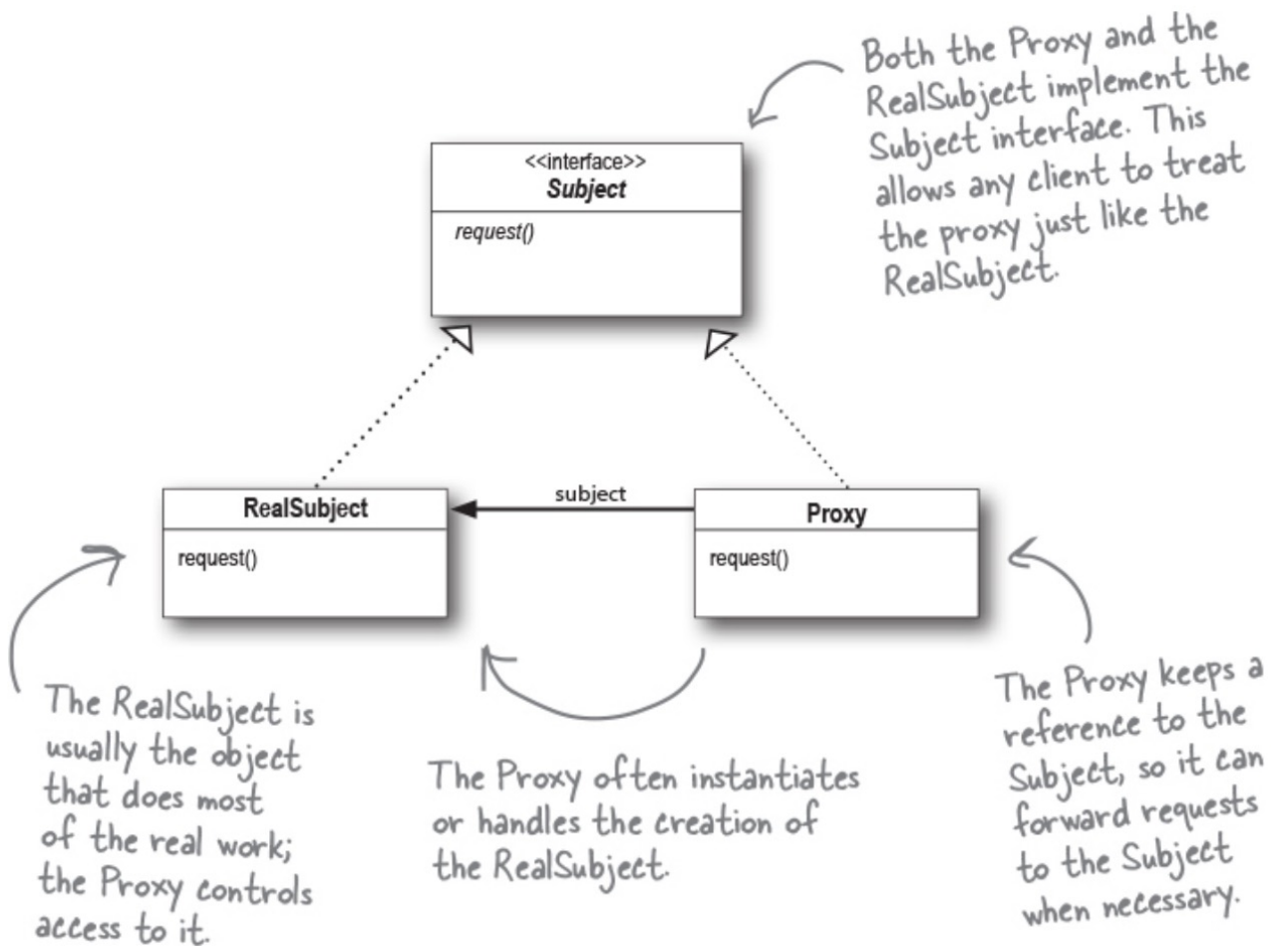
- Category: structural pattern provides a surrogate or placeholder for an object to control access to it.
- Purpose: It acts as an intermediary between the client and the real object, allowing additional functionality to be added without changing the underlying code of the real object.
- https://learning.oreilly.com/library/view/head-first-design/9781492077992/ch11.html

Relationship to other patterns

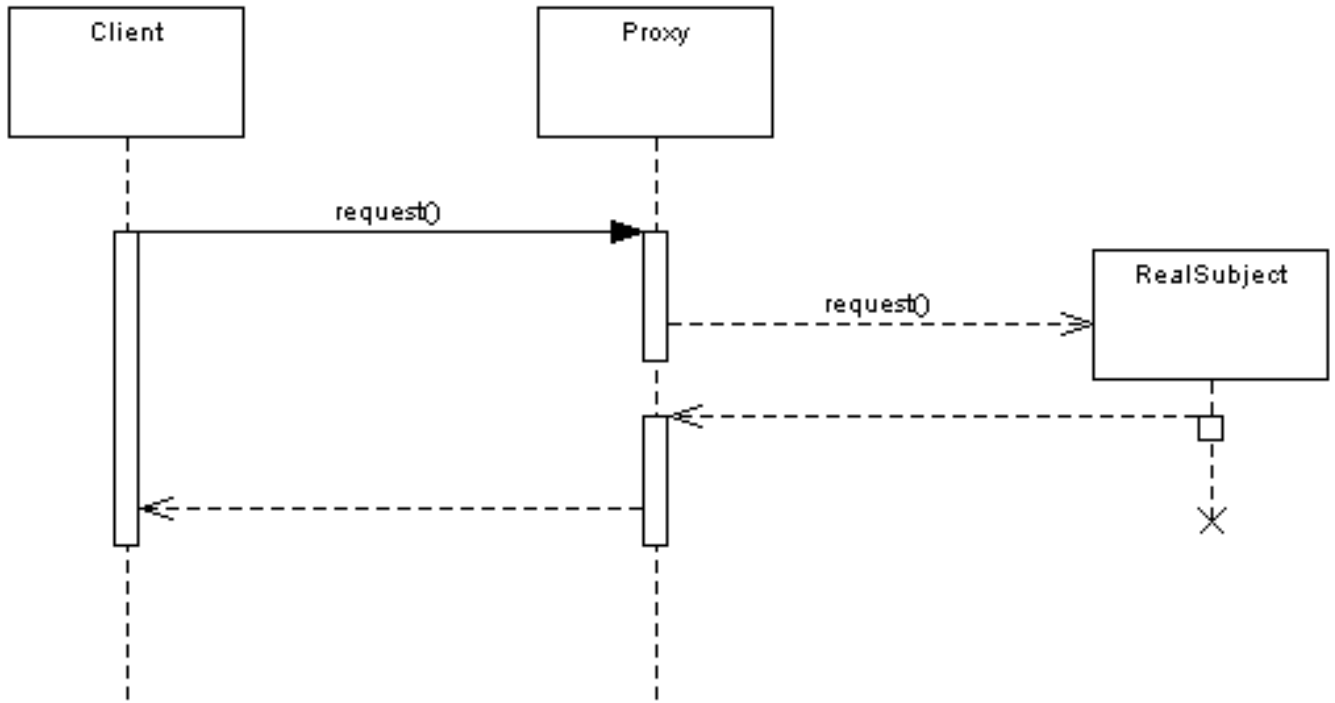
Pattern	Similarity	Difference
Decorator Pattern	Both involve wrapping an object with another object to add functionality	Decorator pattern adds new behaviour to the object, Proxy pattern controls access to the real object.
Adapter Pattern	Both involve wrapping an object	Adapter pattern provides a different interface for the wrapped object, Proxy pattern controls access to the real object.
Facade Pattern	Both provide an interface to simplify interactions with a complex system	Facade pattern provides a unified interface to a group of objects or subsystems, Proxy pattern focuses on controlling access to a single object

Proxy Pattern

Class diagram



Sequence diagram



How it works

```
// Define the Subject interface
interface Subject {
    void request();
}

// Define the Real Subject
class RealSubject implements Subject {
    public void request() {
        System.out.println("RealSubject: Handling the request");
    }
}

// Define the Proxy
class Proxy implements Subject {
    private RealSubject realSubject;

    public Proxy() {
        this.realSubject = new RealSubject();
    }

    public void request() {
        // Perform additional operations or access control logic here
        System.out.println("Proxy: Intercepting the request");
        realSubject.request();
    }
}
```

How it works

```java
// Client code
public class Client {
    public static void clientCode(Subject subject) {
        subject.request();
    }

    public static void main(String[] args) {
        RealSubject realSubject = new RealSubject();
        Proxy proxy = new Proxy();

        // Access the Real Subject directly
        System.out.println("Accessing the Real Subject directly:");
        clientCode(realSubject);

        // Access the Real Subject through the Proxy
        System.out.println("\\nAccessing the Real Subject through the Proxy:");
        clientCode(proxy);
    }
}
```

```
Accessing the Real Subject directly:
RealSubject: Handling the request

Accessing the Real Subject through the Proxy:
Proxy: Intercepting the request
RealSubject: Handling the request
```

Remote proxy

- Remotely monitor the state and inventory of multiple Gumball Machines.

https://gitlab.tamk.cloud/proxy-pattern-demo-code/remote-proxy-implementation-of-the-gumball-machine

Java RMI

Remote Method Invocation is a Java API that allows objects in different Java Virtual Machines (JVMs) to communicate and interact remotely.

eg. Gumball Machine and the GumballMonitor client

It enables the invocation of methods on remote objects as if they were local objects, simplifying the development of distributed systems.

Step 1: Create a remote interface

Define the remote methods that the GumballMonitor will use to retrieve information from the remote Gumball Machine.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface GumballMachineRemote extends Remote {
    int getCount() throws RemoteException;
    String getLocation() throws RemoteException;
    State getState() throws RemoteException;
}
```

Step 2: Implement the interface in a concrete class

- Extend the UnicastRemoteObject class to make the GumballMachine object available for remote method invocation.

```java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class GumballMachine extends UnicastRemoteObject implements GumballMachineRemote {
    private int count;
    private String location;
    // Other instance variables and methods for the gumball machine

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        this.count = numberGumballs;
        this.location = location;
    }

    @Override
    public int getCount() throws RemoteException {
        return count;
    }

    @Override
    public String getLocation() throws RemoteException {
        return location;
    }

    public State getState() {
        return state;
    }
    // Implement other remote methods for the gumball machine
}
```

Step 3: Create the GumballMonitor class

To monitor and report the state and inventory of the Gumball Machine

```
public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
            System.out.println("Current state: " + machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Step 4: Register the class with the RMI registry

Use the Naming.rebind() method to bind the GumballMachine object to a name in the RMI registry.

```java
import java.rmi.Naming;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class GumballMachineServer {
  public static void main(String[] args) {
    try {
      // Create the first instance of the gumball machine
      GumballMachine gumballMachine1 = new GumballMachine("Main Street", 100);
      gumballMachine1.insertQuarter();

      // Create the second instance of the gumball machine
      GumballMachine gumballMachine2 = new GumballMachine("Park Avenue", 200);
      gumballMachine2.insertQuarter();
      gumballMachine2.turnCrank();

      // Start the RMI registry on the default port (1099)
      Registry registry = LocateRegistry.createRegistry(1099);

      // Bind the first gumball machine to a name in the registry using Naming.rebind()
      Naming.rebind("rmi://localhost/GumballMachine1", gumballMachine1);

      // Bind the second gumball machine to another name in the registry using Naming.rebind()
      Naming.rebind("rmi://localhost/GumballMachine2", gumballMachine2);

      System.out.println("Gumball machines are ready.");
    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

```
(base) jingjingyang@jingjings-MacBook-Pro src % javac headfirst/designpatterns/proxy/gumball/GumballMachineServer.java
(base) jingjingyang@jingjings-MacBook-Pro src % java headfirst/designpatterns/proxy/gumball/GumballMachineServer
You inserted a quarter
You inserted a quarter
You turned...
YOU'RE A WINNER! You get two gumballs for your quarter
A gumball comes rolling out the slot...
A gumball comes rolling out the slot...
Gumball machines are ready.
^C%
(base) jingjingyang@jingjings-MacBook-Pro src % javac headfirst/designpatterns/proxy/gumball/GumballMachineServer.java
(base) jingjingyang@jingjings-MacBook-Pro src % java headfirst/designpatterns/proxy/gumball/GumballMachineServer
You inserted a quarter
You inserted a quarter
You turned...
A gumball comes rolling out the slot...
Gumball machines are ready.
```

Step 5: Connecting to the Remote Gumball Machine

Connect to the remote Gumball Machine using the Naming.lookup() method;

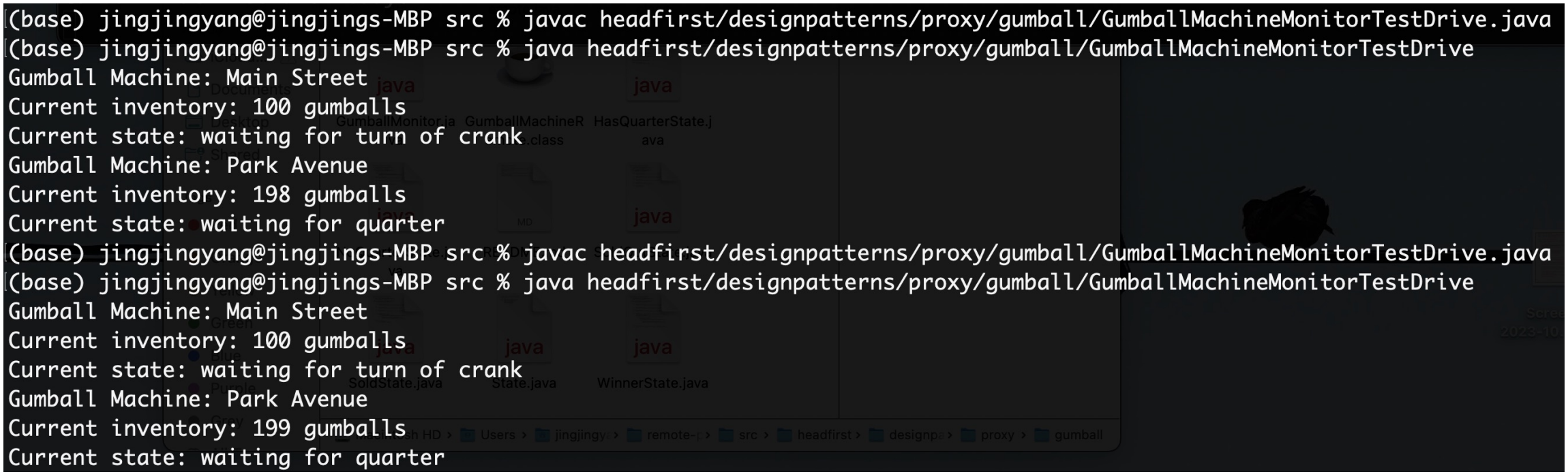
Retrieve information from the remote Gumball Machine.

```java
import java.rmi.Naming;

public class GumballMachineMonitorTestDrive {
    public static void main(String[] args) {
        try {
            // Get a reference to the Gumball Machine remote objects from the registry using Naming.lookup()
            GumballMachineRemote gumballMachine1Remote = (GumballMachineRemote) Naming.lookup("rmi://localhost/GumballMachine1");
            GumballMachineRemote gumballMachine2Remote = (GumballMachineRemote) Naming.lookup("rmi://localhost/GumballMachine2");

            // Create GumballMonitor objects for each remote machine
            GumballMonitor gumballMonitor1 = new GumballMonitor(gumballMachine1Remote);
            GumballMonitor gumballMonitor2 = new GumballMonitor(gumballMachine2Remote);

            // Generate reports for each remote machine
            gumballMonitor1.report();
            gumballMonitor2.report();
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```


(base) jingjingyang@jingjings-MBP src % javac headfirst/designpatterns/proxy/gumball/GumballMachineMonitorTestDrive.java
(base) jingjingyang@jingjings-MBP src % java headfirst/designpatterns/proxy/gumball/GumballMachineMonitorTestDrive
Gumball Machine: Main Street
Current inventory: 100 gumballs
Current state: waiting for turn of crank
Gumball Machine: Park Avenue
Current inventory: 198 gumballs
Current state: waiting for quarter
(base) jingjingyang@jingjings-MBP src % javac headfirst/designpatterns/proxy/gumball/GumballMachineMonitorTestDrive.java
(base) jingjingyang@jingjings-MBP src % java headfirst/designpatterns/proxy/gumball/GumballMachineMonitorTestDrive
Gumball Machine: Main Street
Current inventory: 100 gumballs
Current state: waiting for turn of crank
Gumball Machine: Park Avenue
Current inventory: 199 gumballs
Current state: waiting for quarter

Use cases

- ✓Remote proxy: Use a Proxy as a local representative of a remote object, allowing the client to interact with the remote object without dealing with the complexities of network communication.
- ✓Virtual proxy: Use a Proxy to create a placeholder for an expensive object, delaying its creation or initialization until it is actually needed.
- ✓Protection proxy: Use a Proxy to control access to sensitive or critical resources, enforcing security measures such as authentication, authorization, and encryption.

Use cases

- ✓Caching proxy: Use a Proxy to cache the results of expensive operations, reducing the need to recompute the same results multiple times.
- ✓Lazy loading: Use a Proxy object to load expensive resources only when they are actually needed, improving performance and memory usage.
- ✓Access control: Use a Proxy to enforce access control restrictions on an object, allowing or denying certain operations based on the user's permissions or other criteria.
- ✓Log proxy: Use a Proxy to add logging functionality to an object, capturing information about method calls, parameters, and results for debugging or auditing purposes.

Drawbacks

Overhead: Introducing a Proxy object can add some overhead to the system due to the additional layer of indirection. This can impact performance, especially in situations where frequent interactions with the Real Subject are required.
Care should be taken to optimize the Proxy implementation to minimize any performance impact.

Complexity: The Proxy pattern introduces an additional layer of abstraction, which can increase the complexity of the system. Developers need to carefully design and implement the Proxy to ensure it properly handles the interactions between the Client and the Real Subject, without introducing unnecessary complexity or bugs.

Maintenance: Adding a Proxy object to the system introduces an additional component that needs to be maintained and kept in sync with the Real Subject. Any changes or updates to the Real Subject may require corresponding changes to the Proxy, which can increase the maintenance effort.

Exercise

- Implement a proxy class for a banking system that controls access to a real bank account object.
- The proxy should provide additional functionality to log all transactions made on the account and enforce a maximum withdrawal limit of $500 per day.
- The proxy should also handle the authentication of the client before allowing any transactions.

To Do

1. Define an interface or abstract class BankAccount with methods such as deposit(), withdraw(), and getBalance().
2. Implement a RealBankAccount class that represents the actual bank account. This class should implement the BankAccount interface and handle the core functionality of the account.
3. Create a BankAccountProxy class that acts as a proxy for the RealBankAccount. The proxy should implement the same BankAccount interface and delegate the method calls to the real account object.
4. In the BankAccountProxy, add additional functionality to log all transactions made on the account and enforce the maximum withdrawal limit of $500 per day.
5. Implement a method in the proxy to handle client authentication before allowing any transactions.
6. Test the proxy by creating instances of both the RealBankAccount and BankAccountProxy classes. Perform various transactions and check if the proxy behaves as expected, logging transactions and enforcing the withdrawal limit.

```java
public interface BankAccount {
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();
}
```

```java
public class RealBankAccount implements BankAccount {
    private String accountHolderName;
    private String accountNumber;
    private double balance;

    public RealBankAccount(String accountHolderName, String accountNumber) {
        this.accountHolderName = accountHolderName;
        this.accountNumber = accountNumber;
        this.balance = 0.0;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        } else {
            System.out.println("Insufficient funds");
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

```java
public class BankAccountProxy implements BankAccount {
    private BankAccount realAccount;
    private List<String> transactions;

    public BankAccountProxy(BankAccount realAccount) {
        this.realAccount = realAccount;
        this.transactions = new ArrayList<>();
    }

    public void deposit(double amount) {
        realAccount.deposit(amount);
        transactions.add("Deposit: " + amount);
    }

    public void withdraw(double amount) {
        if (amount <= 500) {
            realAccount.withdraw(amount);
            transactions.add("Withdrawal: " + amount);
        } else {
            System.out.println("Maximum withdrawal limit exceeded");
        }
    }

    public double getBalance() {
        return realAccount.getBalance();
    }

    public List<String> getTransactions() {
        return transactions;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccountProxy("John Doe", "1234567890");

        // Make some transactions
        account.deposit(1000);
        account.withdraw(200);
        account.withdraw(700);

        // Print the account balance
        System.out.println("Account Balance: $" + account.getBalance());
    }
}
```

Thanks!