

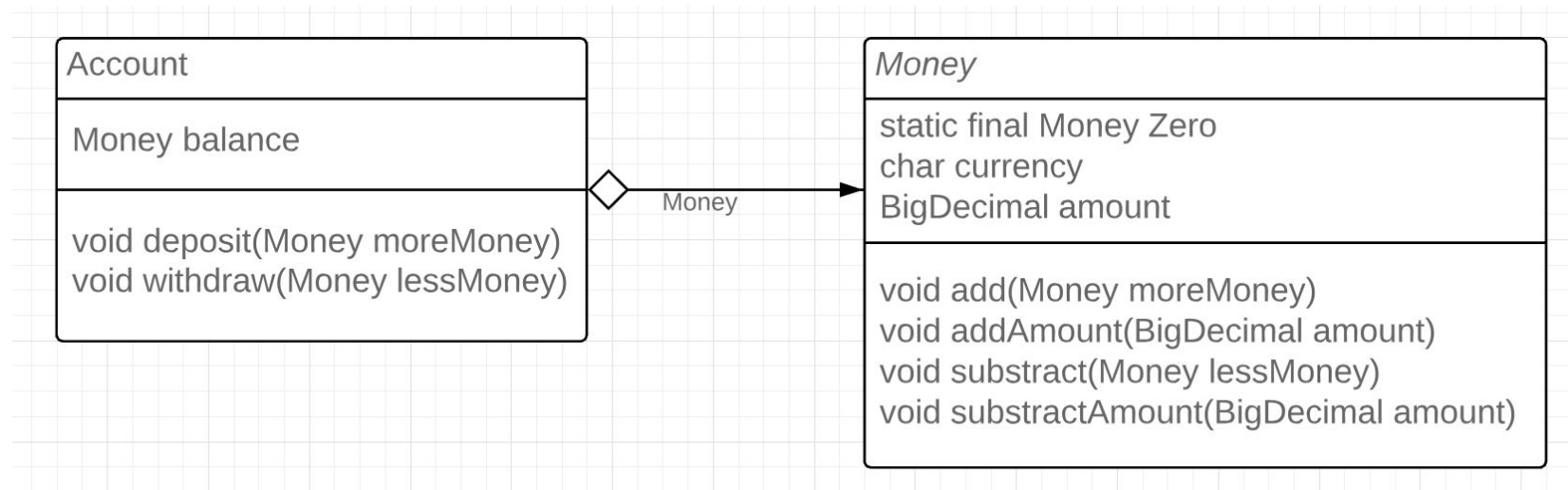
Value Object Pattern

Xiaohan Zhang, Hao Zhang, Chen Luo

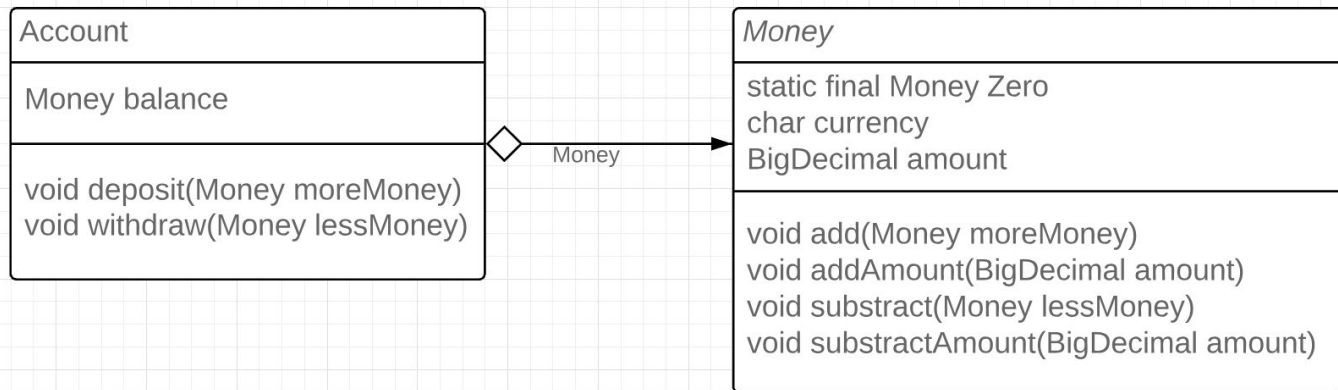
Intent

Implement datatypes as **immutable classes** so that their instances can be handled like built-in values.

Motivation



Motivation

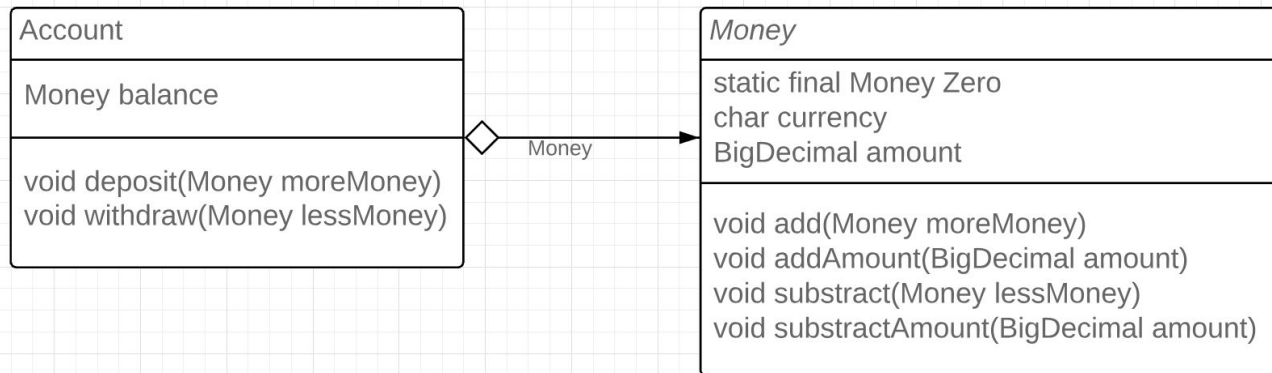


```
public static synchronized void transfer(Money money, Account from, Account to) {
    Money fromBalance = from.getBalance();
    fromBalance.subtract(money);
    if (fromBalance.isLowerThan(Money.ZERO)) {
        throw new RuntimeException("Insufficient funds!");
    }

    from.withdraw(money);
    to.deposit(money);
}
```

Check if the from account has enough money, and lead to withdraw amount twice

Motivation

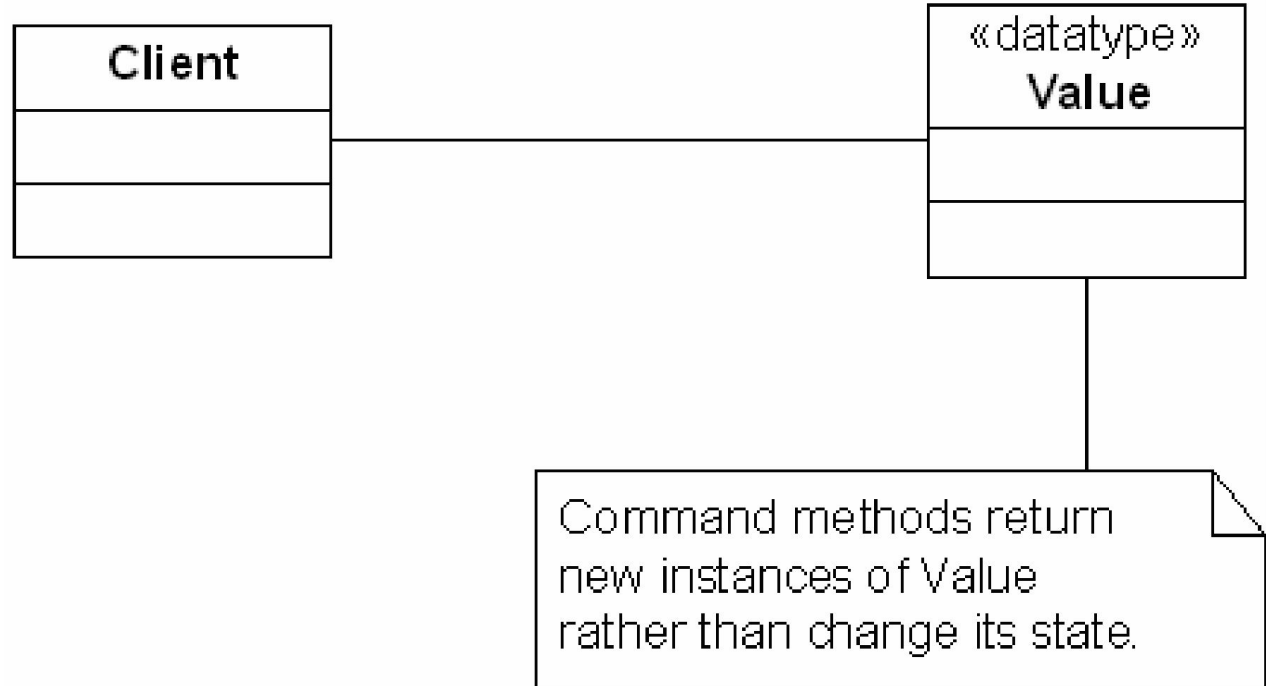


Make each **Money** object immutable, which means you make it return a new object every time where otherwise it would change its state.

```
public Money addAmount(BigDecimal otherAmount) {
    BigDecimal newAmount = amount.add(otherAmount);
    return new Money(currency, newAmount);
}
```

```
public Money subtractAmount(BigDecimal otherAmount) {
    BigDecimal newAmount = amount.subtract(otherAmount);
    return new Money(currency, newAmount);
}
```

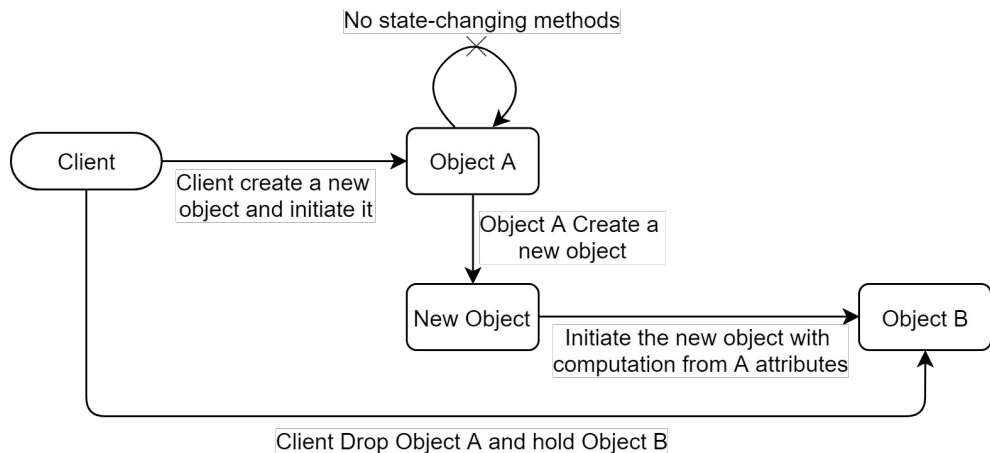
Participants



- Clients: The Client class uses Value instances like built-in datatypes, you write code like `attr = attr.calculate();` rather than `attr.calculate()`
- Value: The Value class represents a domain-specific value type.

Collaborations

- A client creates new Value instances.
- Necessary initialization data is provided in constructor.
- **No state-changing** methods.
- A Value instance provides information to a client through **read-only methods** (a.k.a. observer methods).
- Instead of using state-changing methods, the value object **creates a new instance** of its class and initializes it with the computation result.
- A client will receive a new Value instance back. (**Drop old** value object, **Keep new** value object.)
- Most notably, if the value is stored in an attribute of the client, the client replaces the value object representing the attribute's value with the new value object.



Applicability

Use the Value Object pattern if:

1. The domain concept you are implementing represents a value type (Currency, Date, Domain Address, Percentages, etc)
2. The resulting class does not become too heavyweight as to slow down performance significantly. (The pattern may keep creating and dropping objects.)

Consequences

Pros

- Better domain modeling and understanding.
- Safer programs.
- Potentially better performance.
- Concurrency.
- Persistence.
- Serialization.
- Distribution.

Cons

- More complicated code.
- Changes in coding style.
- Potentially lower performance.

Know uses

- In the financial services domain are monetary amounts and currencies.
- Internet domain they are protocol names, domain names, and URLs.
- In engineering they are the metric system and its units.
- Java JDK implementations. The String, Integer, and Float classes are close-to-the-system Value Object implementations.

Implementation

- Identity, equality and hash code.
- Immutability.

Sample Code

```
public class Pokemon {
    public static final Pokemon Initial = new Pokemon(0, 0, 0, "", "");
    protected int Id;
    protected int HP;
    protected int Attack;
    protected String Type;
    protected String Race;
    private Pokemon(int Id, int HP, int Attack, String Type, String Race) {
        this.Id = Id;
        this.HP = HP;
        this.Attack = Attack;
        this.Type = Type;
        this.Race = Race;
    }
    // Static factory method to create new instances.
    public static Pokemon valueOf(int Id, int HP, int Attack, String Type, String Race) {
        return new Pokemon(Id, HP, Attack, Type, Race);
    }

    public int getId() {
        return Id;
    }

    public void setId(int Id){
        this.Id = Id;
    }
}
```

Sample Code

```
public class Person {  
    // protected Pokemon Initial = Pokemon.Initial;  
    protected Pokemon Partner;  
    public synchronized void Catch(Pokemon pokemon) {  
        Partner.setHP(pokemon.HP);  
        Partner.setId(pokemon.Id);  
        Partner.setAttack(pokemon.Attack);  
        Partner.setRace(pokemon.Race);  
        Partner.setType(pokemon.Type);  
    }  
}
```

Sample Code

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    Pokemon other = (Pokemon) obj;
    if (HP != other.HP) {
        return false;
    }
    if (Attack != other.Attack) {
        return false;
    }
    if (!this.equalsRace(other)) {
        return false;
    }
    return true;
}
```

```
Target is Pokemon [Id=25, HP=60, Attack=10, Type=Electric, Race=Mouse]
Pokemon is Pokemon [Id=25, HP=60, Attack=10, Type=Electric, Race=Mouse]
This is not the same one we want to find.
```

```
Target is Pokemon [Id=25, HP=60, Attack=10, Type=Electric, Race=Mouse]
Pokemon is Pokemon [Id=25, HP=60, Attack=10, Type=Electric, Race=Mouse]
This is the same one we want to find.
```