Simple Case Study of a Money Class

[Skrien Ch. 6] What are the advantages and disadvantages of these representations for a quantity of money?

- Floating-point number
- Two integers, one for dollars and one for cents
- One long integer, with an implied decimal point.

One limitation of all of these is

Let’s design a USMoney class. Our first decision is, Should the values be mutable, like Points, or immutable, like Strings?

What are the advantages and disadvantages of mutability?

How do you make a class’s instances immutable?

- Make all ________________ private.
- Declare the class, or all its methods,
- Don’t provide any _________ methods.

What are some responsibilities of a USMoney object?

- Represent a positive or negative quantity of dollars and cents.
- Know the number of dollars and cents that it represents.
- Print itself as a string in a desired format, such as “$3.14” or “3.14USD.”
- Determine whether it represents a quantity that is equal to, greater than, or less than the quantity represented by another USMoney object.
- Add or subtract itself to or from another USMoney object.
- Multiply or divide itself by a given positive or negative number.
• Negate the amount it represents.

Here are methods that provide those behaviors.

```java
public class USMoney implements Comparable<USMoney> {
    public USMoney(int dollars, int cents)
    public USMoney(long cents)
    public long getAmount();
    public String toString();
    public int compareTo(USMoney o);
    public boolean equals(Object o);
    public int hashCode();
    public USMoney plus(USMoney);
    public USMoney minus(USMoney);
    public USMoney times(double factor);
    public USMoney dividedBy(double denominator);
    public USMoney negate();
}
```

Do we need to override the inherited `equals()` method?

Notice how arithmetic operations return values instead of changing the values on which they operate.

What are some benefits of having a class for `USMoney`, rather than using `longs`? Skrien cites two. Please comment on them.

• Encapsulation. Explain this, and explain why it may be a questionable advantage.

• Compiler can detect more errors. Explain.

Representing different currencies

[Skrien § 6.3] Should we use different classes for, say, euros and pesos?

We could create an abstract `Money` class:
public abstract class Money implements Comparable<Money> {
    public long getAmount()
    public String toString()
    public int compareTo(Money m)
    public boolean equals(Object o)
    public int hashCode()
    public Money plus(Money)
    public Money minus(Money)
    public Money times(double factor)
    public Money dividedBy(double divisor)
    public Money negate()
}

Since one performs arithmetic on all kinds of money in the same way, a lot of methods could be implemented in Money.

There would be few differences, but many subclasses.

Or, we could have a single Money class, with an instance variable representing currency:

public class Money {
    private long amount;
    private String currency;

    public Money(long amount, String currency) {
        this.amount = amount;
        this.currency = currency;
    }

    public String toString() {
        return amount/100 + "." + amount%100 + currency;
    }

    // other methods
}
However, different currencies use different conventions for decimal points.

- Iraq & Jordan use 3 digits to the right of the decimal point.
- Japan doesn’t use fractional yen at all.

It would be quite a mess to keep track of these differences in code (e.g., with an if statement).

Fortunately, Java has a **Currency** class. What pattern does it implement?

With this **Currency** class, we can easily display the money using the currency symbol or the currency’s ISO 4217 code.

Here is an implementation of a method that displays the money using the currency code and a decimal point:

```java
public String toString() {
    String sign = (amount < 0 ? "-" : "");
    Long absAmount = (amount < 0 ? -amount : amount);
    String code = currency.getCurrencyCode();
    int fractDigits = currency.getDefaultFractionalDigits();
    int unit = tenToPower(fractDigits);
    if (fractDigits > 0)
        return sign + absAmount/unit + "." + fill(fractDigits, absAmount % unit) + code;
    else
        return sign + absAmount + code;
}
```

where **tenToPower** and **fill** are auxiliary methods.

Here’s our new implementation of **Money**:

```java
public class Money implements Comparable<Money> {  
    private long amount;
    private Currency currency;
    public Money(long amount, Currency currency) {
        this.amount = amount;
        this.currency = currency;
    }
    public long getAmount() { return amount; }
```
public Currency getCurrency() {
    return currency;
}
public String toString() { ...above... }
public int compareTo(Money m) { ... }
public boolean equals(Object o) { ... }
public int hashCode() { ... }
public Money plus(Money m) { ... }
public Money minus(Money m) { ... }
public Money times(double factor) { ... }
public Money dividedBy(double divisor) { ... }
public Money negate() { ... }
}

Dealing with mixed currencies

[Skrien § 6.5] Our Money class has assumed that all instances would be in the same currency.

Suppose we have some dollars and some euros, for example. How should we represent them?

We could create a MixedMoney class, which serves as a “wallet” containing quantities of different currencies.

Would MixedMoney and Money have the same operations?

Then what should be the relationship between the two classes?

We could rename Money to SimpleMoney and use these declarations:

```java
public interface Money extends Comparable<Money> { 
    public int compareTo(Money o);
    public Money plus(Money money);
    public Money minus(Money money);
    public Money times(double factor);
    public Money dividedBy(double divisor);
    public Money negate();
}
```
public class MixedMoney implements Money {
    public MixedMoney() { ... }
    public String toString() { ... }
    public boolean equals(Object o) { ... }
    public int hashCode() { ... }
    public int compareTo(Money o) { ... }
    public Money plus(Money) { ... }
    public Money minus(Money) { ... }
    public Money times(double factor) { ... }
    public Money dividedBy(double divisor) { ... }
    public Money negate() { ... }
    public Collection getCurrencies() { ... }
    public long getAmount(Currency currency) { ... }
}

Are there any methods of SimpleMoney that MixedMoney does not have?

Converting between currencies
[Skrien § 6.6] What should it mean for two MixedMoney objects to be equal?

But
- which class should have the responsibility of converting?
- which should maintain the exchange rates?

Should it be SimpleMoney? Why or why not?

Should it be MixedMoney? Why or why not?

public class MoneyConverter {
    // returns the conversion rate between the two currencies
    public double getRate(Currency from, Currency to)
    // sets conversion rate to new rate for the 2 currencies
    public void setRate(Currency from, Currency to, double rate)
// returns a Money object with value equal to the given 
// money within the given currency 
public Money convertTo(Money money, Currency to) 
} 

However, when we try to implement `convertTo(...)` , we come up with code that looks something like this:

```java
// in the MoneyConverter class 
public Money convertTo(Money money, Currency to) {
    if (money instanceof SimpleMoney) {
        SimpleMoney simpleMoney = (SimpleMoney) money;
        long newAmount = simpleMoney.getAmount() * 
            getRate(simpleMoney.getCurrency(), to);
        return new SimpleMoney(newAmount, to);
    }
    else { 
        MixedMoney mixedMoney = (MixedMoney) money;
        long totalAmount = 0;
        for (Currency currency: mixedMoney.getCurrencies()) {
            long currentAmount = mixedMoney.getAmount(currency);
            totalAmount +=
                (long) (getRate(currency, to) * currentAmount);
        }
        return new SimpleMoney(totalAmount, to);
    }
}
```

What’s wrong with this?

It would be better for the `Money` object to get rates from a `MoneyConverter` and use the rates to calculate total value.

So `Money` should have this method:

```java
Money convertTo(Currency currency, MoneyConverter converter)
```

If the money subclasses provide this method, then the money converter method is simplified:

```java
// in the MoneyConverter class 
public Money convertTo(Money money, Currency to) {
    return money.convertTo(to, this);
}
```
This solves both of the problems mentioned above.

But, does the MoneyConverter really need a convertTo method?

Now MoneyConverter has just getRate(...) and setRate(...) methods.

Original communication:

MoneyConverter → Money

New communication:

Money → MoneyConverter

Implementation issues

[Skrien § 6.8] How should MixedMoney represent its amounts of individual currencies?

- It could use a Collection of SimpleMoney objects.
- It could use a HashMap with the currency as the key.

But, arithmetic (e.g., addition) on Money objects would need to check the class of the object that is being added to it.

And there’s another problem. Suppose you want to create a MixedMoney object by adding two SimpleMoney objects:

```java
Money dollars = new SimpleMoney(5, Currency.getInstance("USD"));
Money francs = new SimpleMoney(3, Currency.getInstance("CHF"));
Money total = dollars.plus(francs);
```

If we allow this addition, then a lot of class-checking will need to be done.

Well, how 'bout eliminating SimpleMoney? This allows us to store the values as a HashMap keyed by currency.
This does simplify the class structure, but it causes some complications. Think about using a HashMap instead. Would that be a good idea?

A more elegant implementation is to store the data in a tree whose leaves are `SimpleMoney` objects and whose internal nodes are `MixedMoney` objects.

```java
public class MixedMoney extends Money {
    private Money left;
    private Money right;
    ...
    public long getAmount(Currency currency) {
        return left.getAmount(currency) +
                right.getAmount(currency);
    }
    public Money plus(Money other) {
        return new MixedMoney(this, other);
    }
}
```

```java
public class SimpleMoney extends Money {
    private long amount = 0;
    private Currency currency;
    ...
    public long getAmount(Currency currency) {
        if (this.currency.equals(currency))
            return amount;
        else
            return 0;
    }
    public Money plus(Money other) {
        return new MixedMoney(this, other);
    }
    ...
    public Money convertTo(Currency to, MoneyConverter moneyConverter) {
        double rate = moneyConverter.rate(currency, to);
        return new SimpleMoney((int)(amount*rate), to);
    }
}
```

Now we can add different currencies without checking classes!
In fact, the users don’t need to manipulate *SimpleMoney* objects. They can just create *Money* objects via a Factory Method.

We add this method to the *Money* class.

```java
public static Money getMoney(long amount,
        String currency) {
    return new SimpleMoney(amount,
            currency.getInstance(currency));
}
```

Then the user can create money as follows:

```java
Money dollars = Money.getMoney(5, "USD");
Money francs = Money.getMoney(7, "CHF");
Money total = dollars.plus(francs);
```