PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aulas 17-19

Acesso a colecções através do conteúdo

2 Bancos, bancos, bancos,

Not anymore!

Contas bancárias

- Vamos desenvolver um programa que permita efetuar operações sobre contas bancárias
- Enquadramento geral
  - Temos apenas um banco
  - Existem contas à ordem e contas a prazo
  - Os clientes do banco podem ser titulares de várias contas

Caraterização das contas

- Conta à ordem
  - Uma conta à ordem caracteriza-se por um código (único), o titular, a data de abertura, a data do último movimento, o saldo, bem como todos os movimentos realizados na conta
  - Os movimentos na conta são depósitos ou levantamentos, os quais têm uma data e valor associados
- Conta a prazo
  - Uma conta a prazo caracteriza-se por um código (único), o titular, a data de abertura, a data de início de contagem de juros, a qual é atualizada sempre que os juros são calculados e adicionados ao capital, o montante de capital depositado, o prazo para cálculo de juros e a taxa anual pré-definida

Contas à ordem no banco

- Gestão de contas à ordem
  - Criar uma conta
  - Encerrar uma conta
  - Movimentar uma conta
    - Registo de uma operação de depósito ou levantamento
  - Listar uma conta
  - Listar todas as contas
  - Listar o conjunto de clientes, ordenado se necessário
  - Listar as contas pertencentes a um dado titular
  - Listar as contas com capital superior a um dado valor, ordenadas se necessário
  - Listar as contas que não são movimentadas a partir de determinada data
  - Listar o conjunto de todos os titulares de contas
  - Listar o capital total de contas para cada um dos clientes

Contas a prazo no banco

- Gestão de contas a prazo
  - Criar uma conta
  - Encerrar uma conta
  - Movimentar uma conta no dia de vencimento de juros
    - Calcula os juros e adiciona ao capital, podendo também levantar ou depositar
    - A data do vencimento passa a ser a data de referência para o próximo período de juros
  - Listar uma conta
  - Listar todas as contas
  - Listar o conjunto de clientes, ordenado se necessário
  - Listar as contas pertencentes a um dado titular
  - Listar as contas com capital superior a um dado valor, ordenadas se necessário
  - Listar as contas que vencem juros até determinada data
  - Listar o conjunto de todos os titulares de contas
  - Listar o capital total das contas de cada um dos clientes

Proposta de solução para o problema

10 minutos

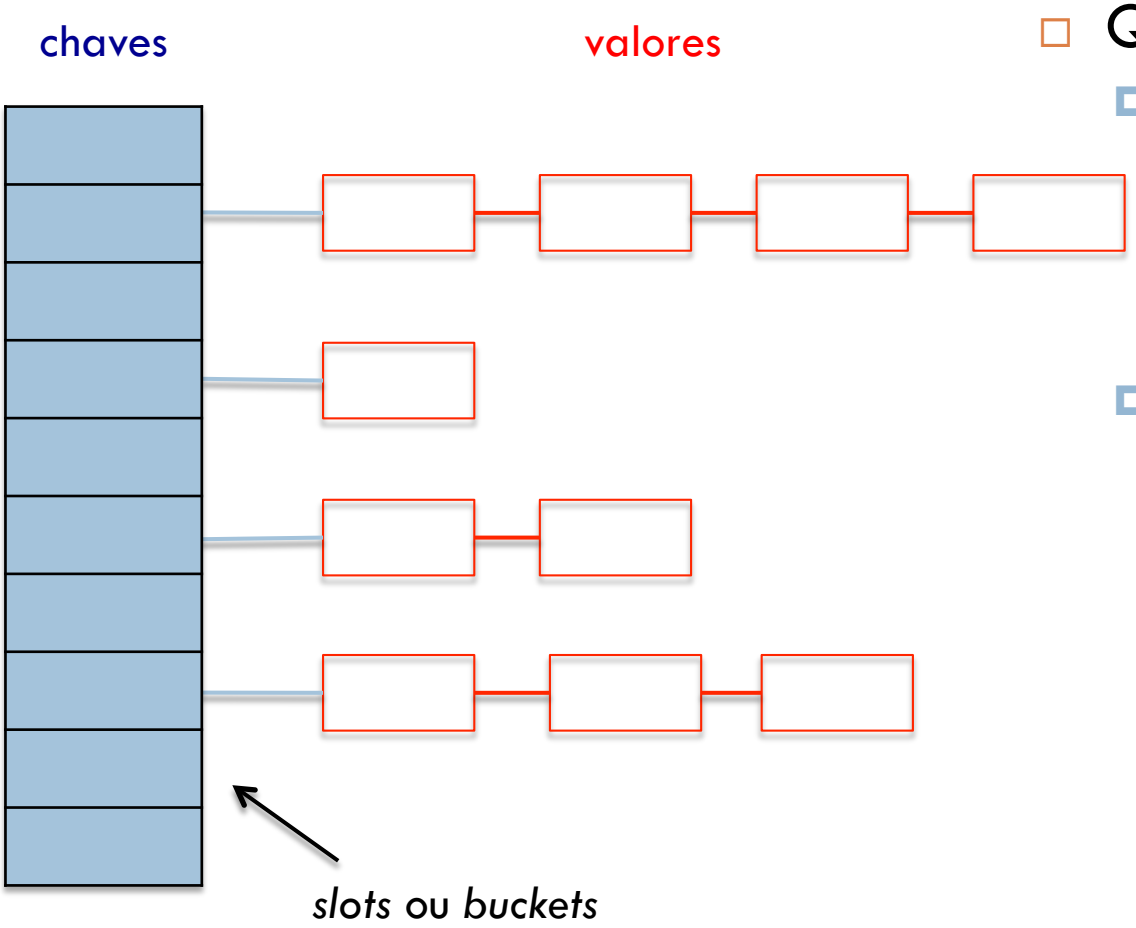
Um pequeno problema com as listas

- Se considerarmos que existem milhares de contas no banco, e guardarmos as contas num ArrayList ou num LinkedList, qual a ordem de grandeza do tempo necessário para obter informação sobre determinada conta?
  - Apesar de a interface List especificar um método `contains`, que permite verificar se determinado elemento está na lista ou não, esta solução é bastante ineficiente
- E se o problema fosse agora pesquisar palavras num dicionário Português-Inglês?
- A solução passa por utilizar estruturas apropriadas para aceder a coleções por conteúdo
  - Ex: tabelas de dispersão (*hash tables*), que serão estudadas detalhadamente na disciplina de Algoritmos e Estruturas de Dados

Conceito de tabela de dispersão

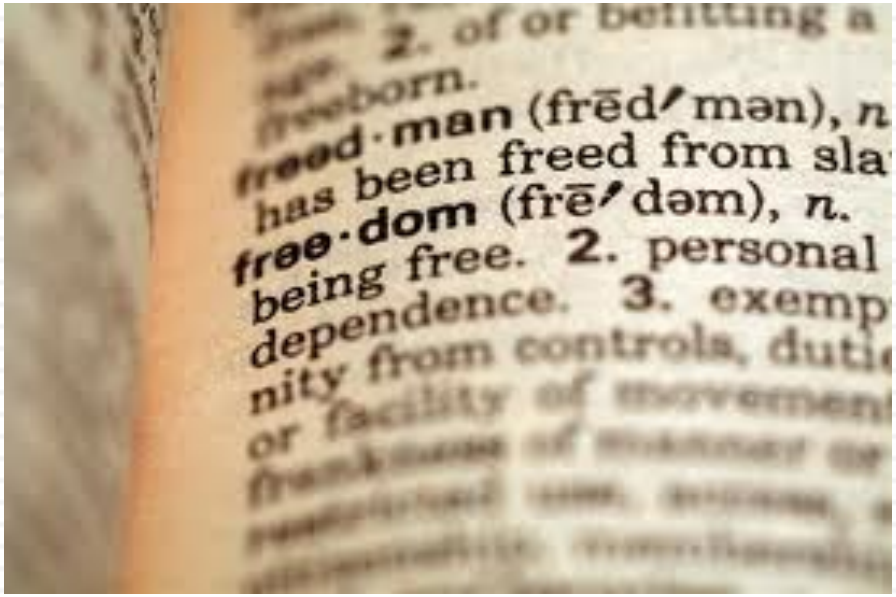
- Estrutura sob a forma de vector de listas ligadas que permite aceder a informação através do respectivo conteúdo
- Para cada elemento, é possível calcular um código inteiro que permite inferir em que lista é que o elemento se encontra
  - Denomina-se função de *hashing*, permitindo associar uma chave ao seu valor (ex: nome de uma pessoa ao seu número de telefone)
- A eficiência depende de vários factores, como por exemplo o número de listas existentes versus o número de entradas na tabela
  - Estes factores são a capacidade inicial e o factor de carga

Conceito de tabela de dispersão



- Quantos *slots*?
  - Talvez 1.5 vezes o número de elementos que se espera guardar na tabelas, mas aproximando a um número primo
  - Quando a tabela atinge um factor de carga considerável, por exemplo 75% de *slots* preenchidos, então devemos reestruturar a tabela, aumentando a sua capacidade

Dicionários



Interface **Map<K,V>**

- Um Map, estrutura definida em java.util, mapeia chaves a valores, com correspondências finitas e unívocas de um para um
  - Não existem chaves duplicadas
  - Cada chave pode mapear no máximo um único valor
  - Para que a pesquisa seja eficiente, o objecto correspondente a uma chave deve implementar os seguintes métodos:
    - `hashCode()`
    - `equals()`

```
public interface Map<K,V>
```

K, o tipo das chaves **V**, o tipo dos valores mapeados

Interface `Map<K,V>`

- Operações fundamentais
  - Inserção de elementos
    - Colocar um par chave-valor
    - Colocar vários pares chave-valor
  - Remoção de elementos
    - Remover o valor associado a uma chave
  - Consulta e comparação de conteúdos
    - Devolver o valor associado a uma chave
    - Verificar se existe uma determinada chave
    - Verificar se está vazio ou não
    - Verificar se existe um determinado valor
    - Devolver a sua dimensão
  - Criação de iteradores, numa perspectiva de Map enquanto colecção de pares
    - Conjunto de chaves
    - Coleção de valores
    - Conjunto de pares chave-valor

Interface `Map<K,V>`

- Algumas observações
  - É preciso ter algum cuidado quando se utilizam objetos mutáveis como chaves
    - Pode afectar comparações intrínsecas ao método `equals`
  - Por vezes esta estrutura também é conhecida por dicionário
    - No entanto, no contexto da linguagem Java, convém fazer a distinção entre a interface Map e a classe abstracta Dictionary, que Map atualmente substitui

Interface `SortedMap<K,V>`

- SortedMap é um Map que mantém as suas entradas em ordem ascendente, e de acordo com a ordem natural das chaves, ou então de acordo com o comparador que tenha sido fornecido aquando da sua criação
- Bastante utilizado em colecções ordenadas de pares chave-valor, como é o caso de dicionários e listas telefónicas
- Para além das operações de Map, permite ainda
  - Aplicar operações a zonas delimitadas do Map
  - Devolver a primeira e a última chave do mapa ordenado
  - Devolver o comparador utilizado na ordenação, se este existir

```
public interface SortedMap<K,V> extends Map<K,V>
```

Classe `HashMap<K,V>`

- A classe HashMap é uma implementação da interface Map baseado numa tabela de dispersão
  - Permite também null como valor e como chave
  - Não garante ordem no Map

```
public class HashMap<K,V> extends AbstractMap<K,V>
                          implements Map<K,V>, ...
```

K, o tipo das chaves **V**, o tipo dos valores mapeados

Classe HashMap<K,V>

Métodos	
V put (K key, V value)	Associa o valor à chave
void putAll(Map<? extends K, ? Extends V> m)	Copia todos os mapeamentos do mapa indicado para este Map
V remove(Object key)	Remove o mapeamento para a chave, se existir
void clear()	Remove todos os mapeamentos do Map
V get(Object key)	Devolve o valor associado à chave, ou null se não existir
boolean containsKey (Object key)	Devolve true se o mapa contém um mapeamento com a chave indicada
boolean isEmpty()	Devolve true se o mapa não contém mapeamentos

Classe `HashMap<K,V>`

… métodos	
boolean containsValue (Object value)	Devolve true se o Map mapeia uma ou mais chaves ao valor indicado
int size()	Devolve o número de mapeamentos existentes no Map
Set<K> keySet()	Devolve um conjunto com as chaves existentes no Map
Collection<V> values()	Devolve uma colecção com os valores existentes no Map
Set<Map.Entry<K,V>> entrySet()	Devolve um conjunto com os mapeamentos existentes no Map
Object clone()	Devolve uma cópia *shallow* da instância (chaves e valores não são clonados)

Interface `Set<E>`

- Um Set modela o conceito matemático de conjunto
  - Exemplo: baralho de cartas
- A interface Set estende Collection com as seguintes restrições
  - Não são admitidos elementos duplicados
  - Não estabelece ordem entre elementos
- A interface não adiciona métodos para além dos que são herdados de Collection

```
public interface Set<E> extends Collection<E>
```

Interface `SortedSet<E>`

- SortedSet é um conjunto que permite estabelecer o conceito de ordem total entre os seus elementos
- Os elementos são ordenados de acordo com a sua ordem natural, através da implementação da interface Comparable, ou então de acordo com o comparador que tenha sido fornecido aquando da sua criação
- O iterador permite percorrer o conjunto por ordem crescente
- As novas operações que utilizam a ordenação são as seguintes:
  - Aplicar operações a zonas delimitadas do Set
  - Devolver o primeiro e o último elemento do conjunto ordenado
  - Devolver o comparador utilizado na ordenação, se este existir

```
public interface SortedSet<E> extends Set<E>
```

Interface `NavigableSet<E>`

- NavigableSet é um conjunto ordenado com métodos adicionais que visam dar respostas aproximadas aos objectivos de pesquisa especificados
  - Exemplo: o elemento imediatamente inferior ou igual a determinado elemento, o menor elemento do conjunto, o maior elemento do conjunto, etc.
- A iteração dos seu elementos pode ser feita por ordem crescente ou decrescente

```
public interface NavigableSet<E> extends SortedSet<E>
```

Classe `HashSet<K,V>`

- A classe HashSet é uma implementação geral e eficiente da interface Set, baseada numa tabela de dispersão
  - Permitem também null como valor e como chave
  - Não garante a ordem de iteração ao longo do tempo

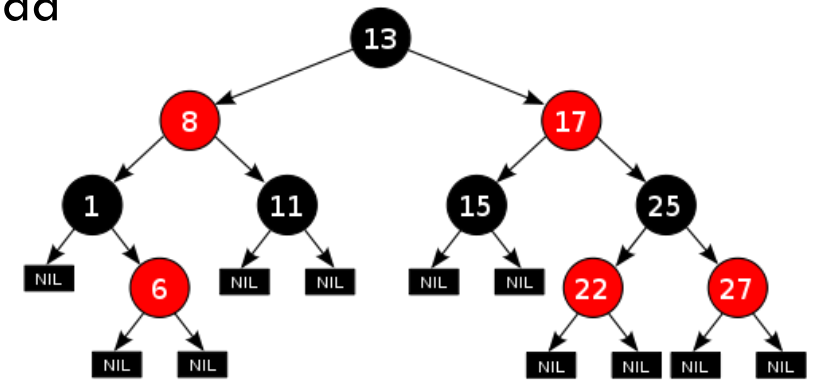
```
public class HashSet<E> extends AbstractSet<E>
                        implements Set<E>, ...
```

Classe `HashSet<E>`

Métodos	
boolean add(E e)	Adiciona o elemento indicado ao conjunto se ainda não existir no conjunto
boolean remove(Object o)	Remove o elemento indicado do conjunto, se este existir no conjunto
void clear()	Remove todos os elementos do conjunto
boolean contains (Object o)	Devolve true se o conjunto contém o elemento indicado
boolean isEmpty()	Devolve true se o conjunto não contém elementos
int size()	Devolve o número de elementos do conjunto
Iterator<E> iterator()	Devolve um iterador sobre os elementos do conjunto
Object clone()	Devolve uma cópia *shallow* da instância (os elementos não são clonados)

Classe `TreeSet<E>`

- A classe TreeSet é uma implementação da interface NavigableSet, com representação através de uma árvore balanceada
  - Exemplo de uma árvore balanceada



- A ordenação dos elementos é feita de acordo com a sua ordem natural ou através de um comparador fornecido aquando da criação da estrutura

```
public class TreeSet<E> extends AbstractSet<E>
                        implements NavigableSet<E>, ...
```

Classe `TreeSet<E>`

Alguns métodos	
boolean add(E e)	Adiciona o elemento indicado ao conjunto se ainda não existir no conjunto
E ceiling(E e)	Devolve o elemento imediatamente superior ou igual ao elemento indicado, se este existir no conjunto, ou null caso contrário
void clear()	Remove todos os elementos do conjunto
boolean contains (Object o)	Devolve true se o conjunto contém o elemento indicado
boolean isEmpty()	Devolve true se o conjunto não contém elementos
int size()	Devolve o número de elementos do conjunto
Iterator<E> iterator()	Devolve um iterador sobre os elementos do conjunto por ordem crescente
Iterator<E> descendingIterator()	Devolve um iterador sobre os elementos do conjunto por ordem decrescente

Classe `TreeSet<E>`

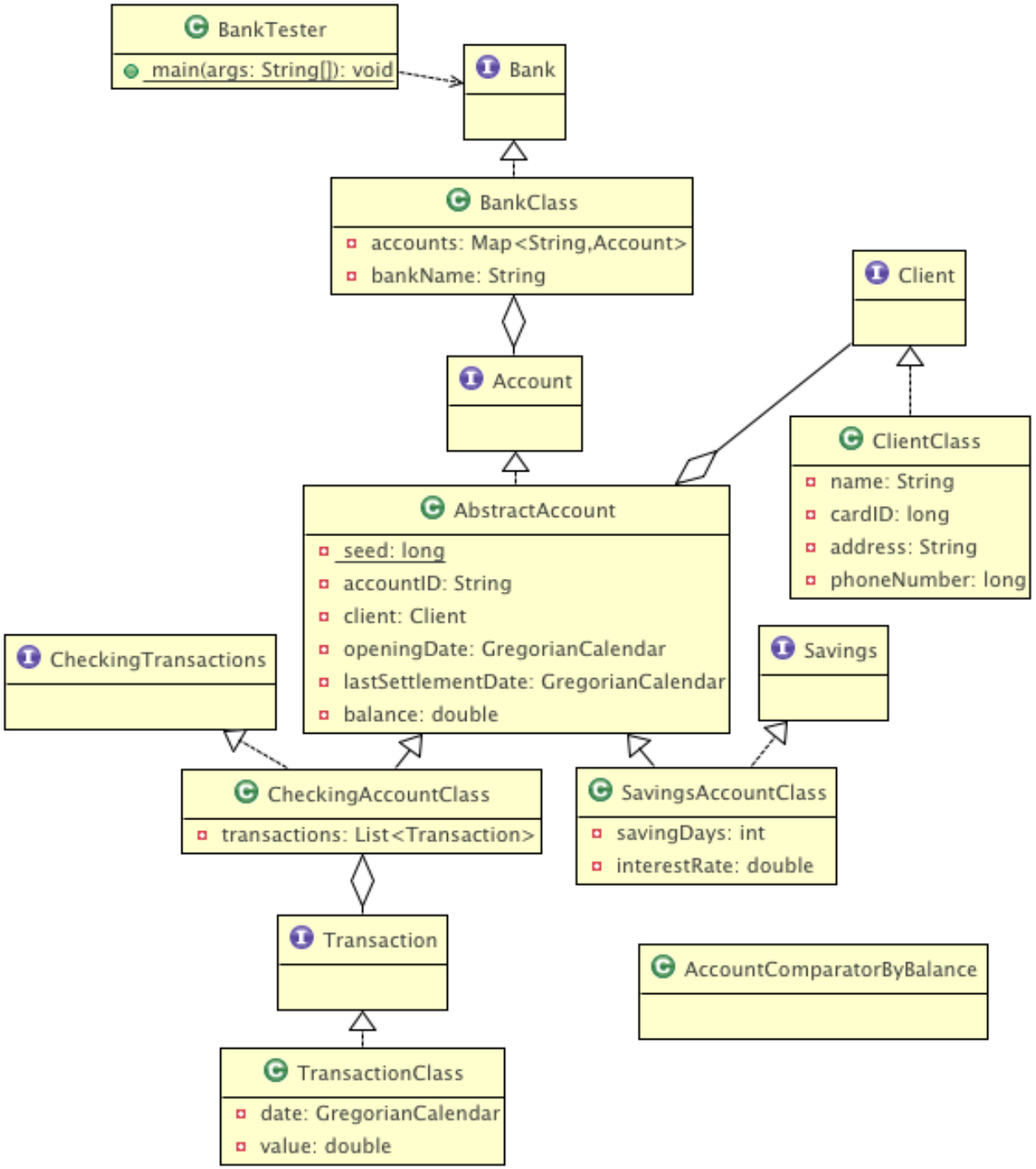
Alguns métodos	
boolean remove(Object o)	Remove o elemento indicado se existir no conjunto
Comparator <? super E> comparator()	Devolve o comparador utilizado na ordenação do conjunto, ou null se é seguida a ordenação natural
NavigableSet<E> descendingSet()	Devolve os elementos do conjunto por ordem inversa
E first ()	Devolve o 1° (menor) elemento do conjunto
E floor(E e)	Devolve o elemento imediatamente menor ou igual ao elemento indicado, ou null se não existir
SortedSet(E) headSet(E e, boolean inclusive)	Devolve uma parte do conjunto cujos elementos são menores (ou iguais com *inclusive* a true) que o elemento indicado
E higher (E e)	Devolve o elemento imediatamente superior ao elemento indicado, ou null se não existir

Classe `TreeSet<E>`

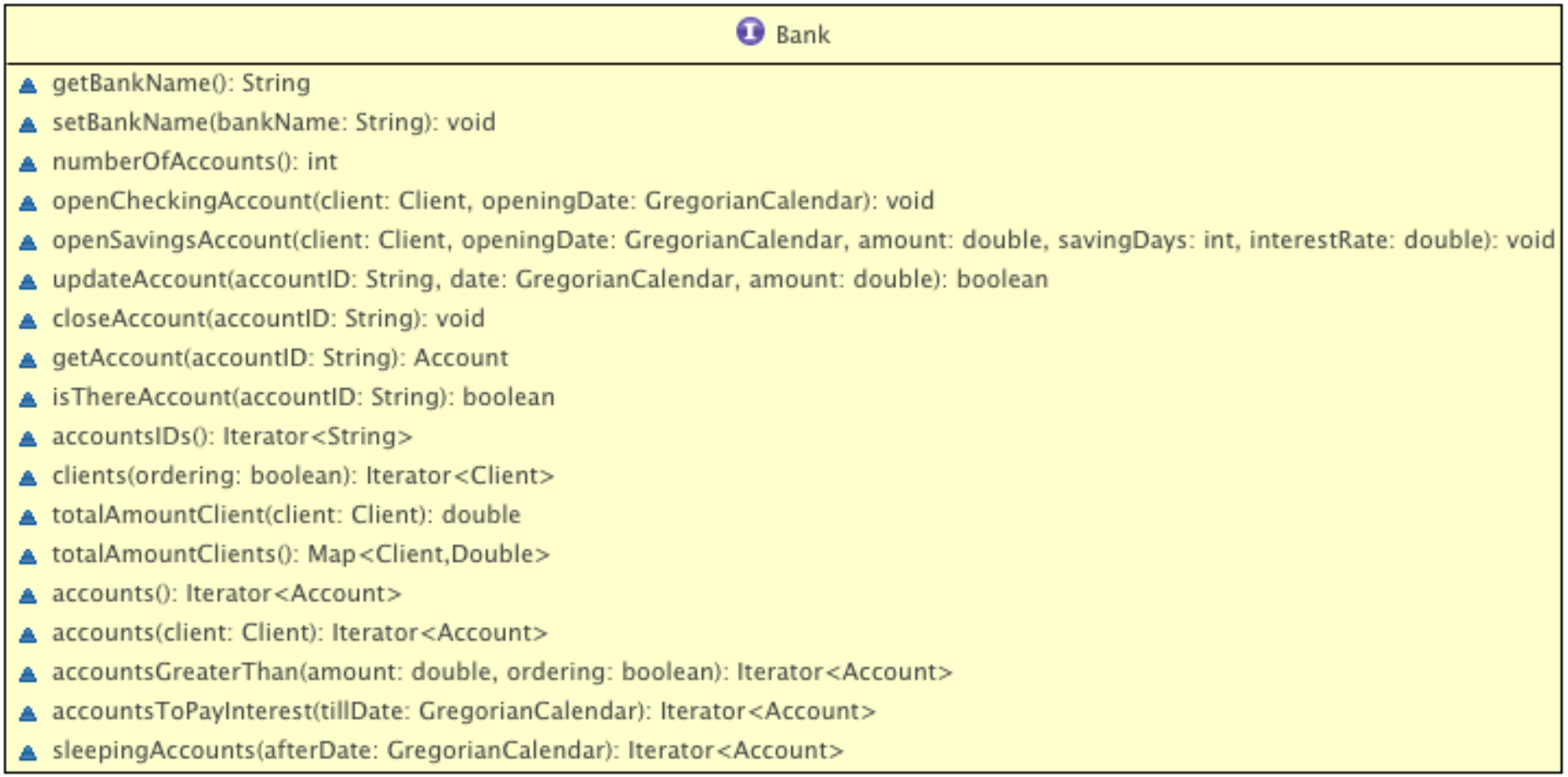
Alguns métodos	
E last()	Devolve o último (maior) elemento do conjunto
E lower(E e)	Devolve o elemento imediatamente inferior ao elemento indicado, ou null se não existir
E pollFirst ()	Devolve e remove o 1° (menor) elemento do conjunto, ou null se o conjunto estiver vazio
E pollLast ()	Devolve e remove o último (maior) elemento do conjunto, ou null se o conjunto estiver vazio
SortedSet<E> subSet(E fromElem, E toElem)	Devolve uma parte do conjunto cujos elementos estão compreendidos entre fromElem, incluído, a toElem, excluído
SortedSet(E) tailSet(E e)	Devolve uma parte do conjunto cujos elementos são maiores ou iguais ao elemento indicado

Voltando ao problema

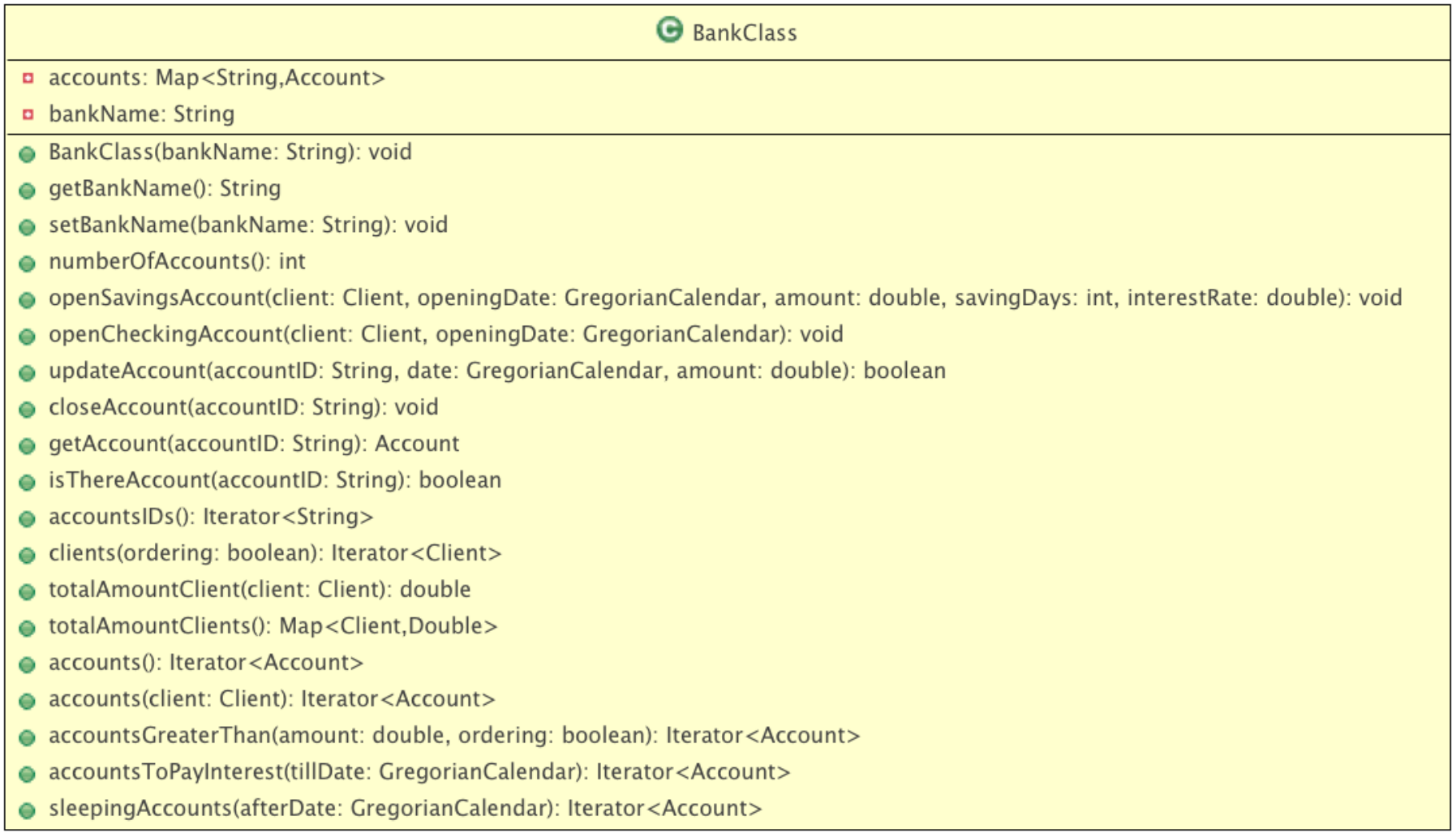
Visão global



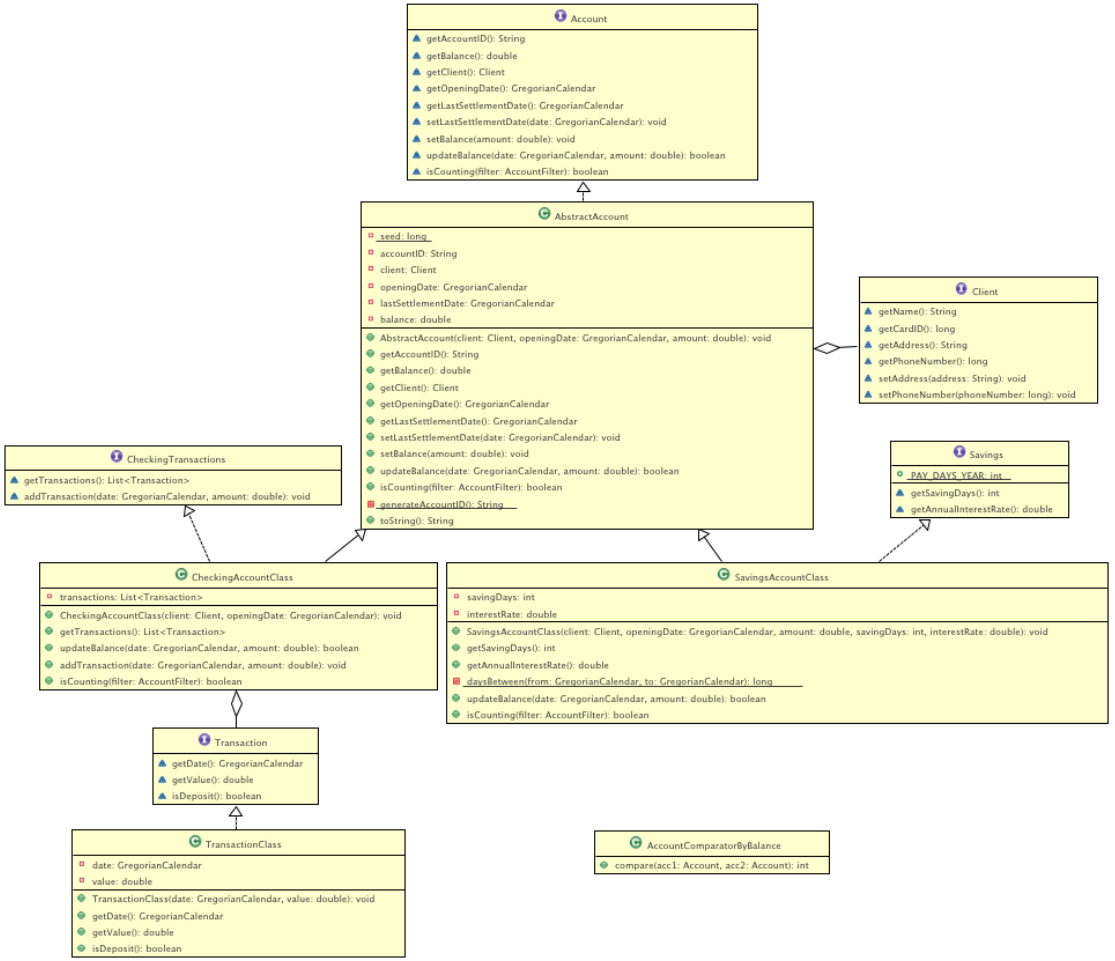
A interface Bank



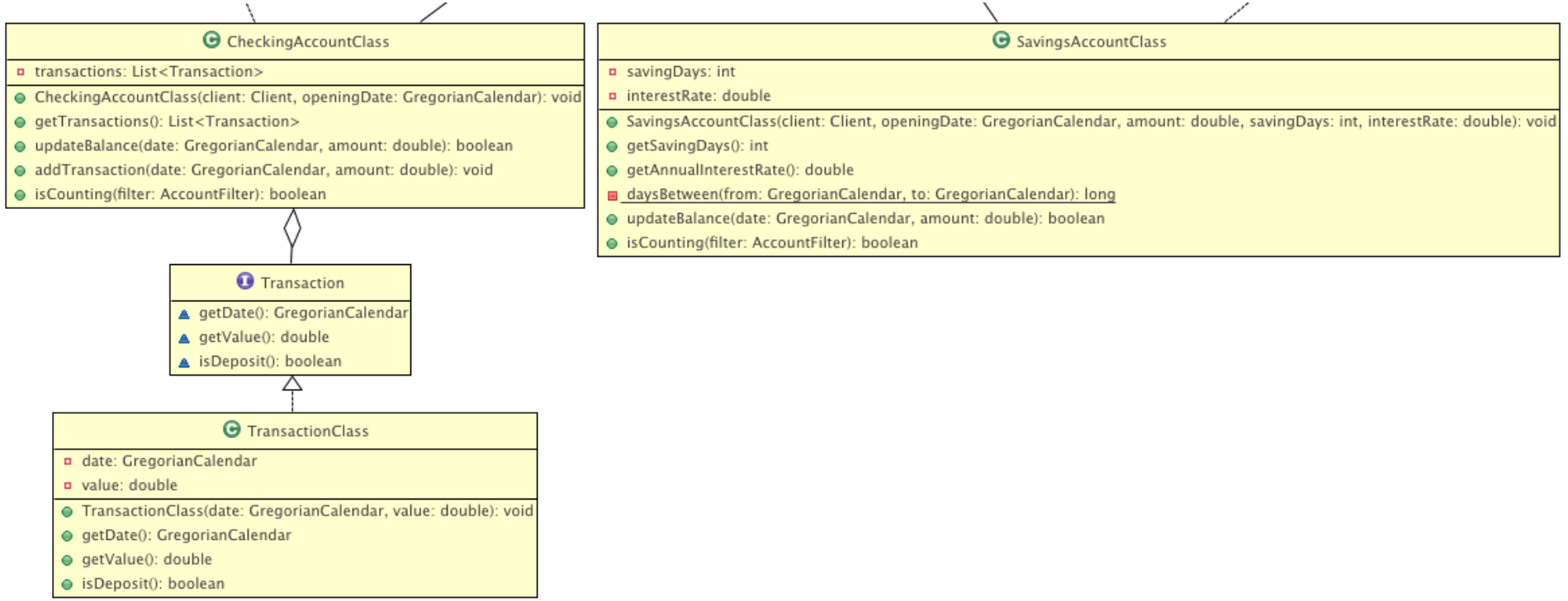
A classe BankClass



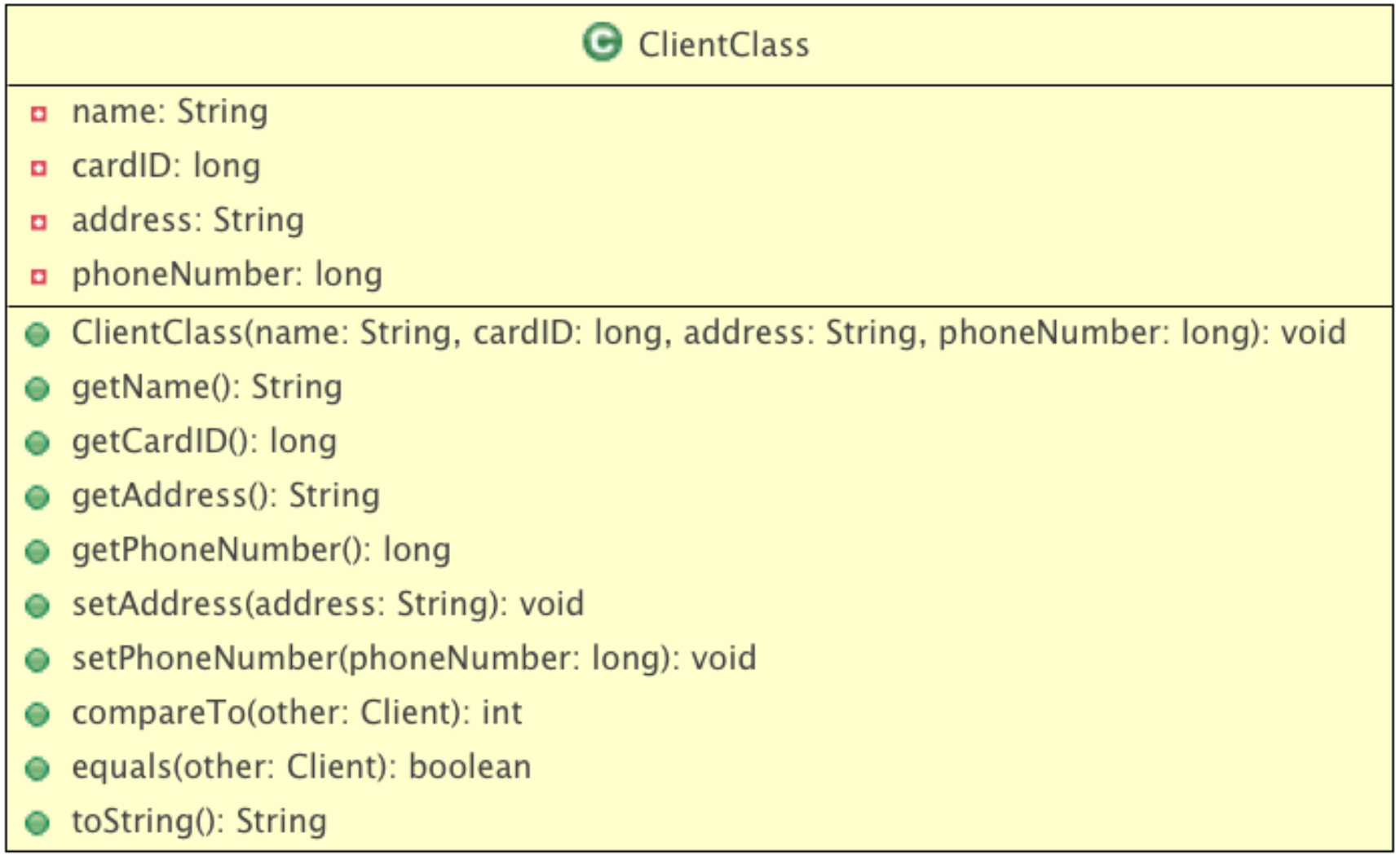
Entidades relacionadas com a interface Account



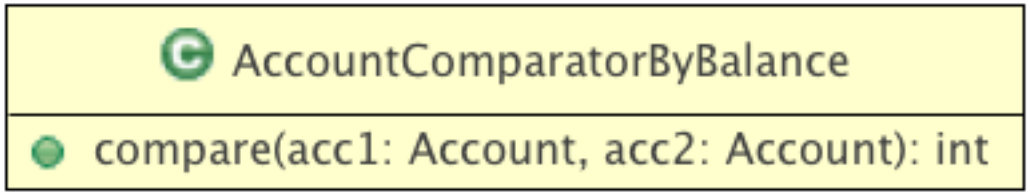

Account
getAccountID(): String
getBalance(): double
getClient(): Client
getOpeningDate(): GregorianCalendar
getLastSettlementDate(): GregorianCalendar
setLastSettlementDate(date: GregorianCalendar): void
setBalance(amount: double): void
updateBalance(date: GregorianCalendar, amount: double): boolean
isCounting(filter: AccountFilter): boolean
AbstractAccount
seed: long
accountID: String
client: Client
openingDate: GregorianCalendar
lastSettlementDate: GregorianCalendar
balance: double
AbstractAccount(client: Client, openingDate: GregorianCalendar, amount: double): void
getAccountID(): String
getBalance(): double
getClient(): Client
getOpeningDate(): GregorianCalendar
getLastSettlementDate(): GregorianCalendar
setLastSettlementDate(date: GregorianCalendar): void
setBalance(amount: double): void
updateBalance(date: GregorianCalendar, amount: double): boolean
isCounting(filter: AccountFilter): boolean
generateAccountID(): String
toString(): String
Client
getName(): String
getCardID(): long
getAddress(): String
getPhoneNumber(): long
setAddress(address: String): void
setPhoneNumber(phoneNumber: long): void
CheckingTransactions
getTransactions(): List<Transaction>
addTransaction(date: GregorianCalendar, amount: double): void
Savings
PAY_DAYS_YEAR: int
getSavingDays(): int
getAnnualInterestRate(): double


CheckingAccountClass
transactions: List<Transaction>
CheckingAccountClass(client: Client, openingDate: GregorianCalendar): void
getTransactions(): List<Transaction>
updateBalance(date: GregorianCalendar, amount: double): boolean
addTransaction(date: GregorianCalendar, amount: double): void
isCounting(filter: AccountFilter): boolean
SavingsAccountClass
savingDays: int
interestRate: double
SavingsAccountClass(client: Client, openingDate: GregorianCalendar, amount: double, savingDays: int, interestRate: double): void
getSavingDays(): int
getAnnualInterestRate(): double
daysBetween(from: GregorianCalendar, to: GregorianCalendar): long
updateBalance(date: GregorianCalendar, amount: double): boolean
isCounting(filter: AccountFilter): boolean
Transaction
getDate(): GregorianCalendar
getValue(): double
isDeposit(): boolean
TransactionClass
date: GregorianCalendar
value: double
TransactionClass(date: GregorianCalendar, value: double): void
getDate(): GregorianCalendar
getValue(): double
isDeposit(): boolean

A classe ClientClass



Outras entidades do programa




```
public enum AccountFilter { CHECKING, SAVINGS }
```

A implementação

Análise detalhada do código Java no ambiente Eclipse ...

Java Framework ... implementações principais

			Implementações			
			Resizable array	Linked list	Balanced tree	Hash table
Interfaces	Collection	Set			✓ TreeSet	✓ HashSet
		List	✓ ArrayList	✓ LinkedList		
		Map			TreeMap	✓ HashMap