Problem 2043: Simple Bank System

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You have been tasked with writing a program for a popular bank that will automate all its incoming transactions (transfer, deposit, and withdraw). The bank has

n

accounts numbered from

1

to

n

. The initial balance of each account is stored in a

0-indexed

integer array

balance

, with the

(i + 1)

th

account having an initial balance of

balance[i]

.

Execute all the

valid

transactions. A transaction is

valid

if:

The given account number(s) are between

1

and

n

, and

The amount of money withdrawn or transferred from is

less than or equal

to the balance of the account.

Implement the

Bank

class:

Bank(long[] balance)

Initializes the object with the

0-indexed

integer array

balance

.

boolean transfer(int account1, int account2, long money)

Transfers

money

dollars from the account numbered

account1

to the account numbered

account2

. Return

true

if the transaction was successful,

false

otherwise.

boolean deposit(int account, long money)

Deposit

money

dollars into the account numbered

account

. Return

true

if the transaction was successful,

false

otherwise.

boolean withdraw(int account, long money)

Withdraw

money

dollars from the account numbered

account

. Return

true

if the transaction was successful,

false

otherwise.

Example 1:

Input

["Bank", "withdraw", "transfer", "deposit", "transfer", "withdraw"] [[[10, 100, 20, 50, 30]], [3, 10], [5, 1, 20], [5, 20], [3, 4, 15], [10, 50]]

Output

[null, true, true, true, false, false]

Explanation

Bank bank = new Bank([10, 100, 20, 50, 30]); bank.withdraw(3, 10); // return true, account 3 has a balance of $20, so it is valid to withdraw $10. // Account 3 has $20 - $10 = $10. bank.transfer(5, 1, 20); // return true, account 5 has a balance of $30, so it is valid to transfer $20. // Account 5 has $30 - $20 = $10, and account 1 has $10 + $20 = $30. bank.deposit(5, 20); // return true, it is valid to deposit $20 to account 5. // Account 5 has $10 + $20 = $30. bank.transfer(3, 4, 15); // return false, the current balance of account 3 is $10, // so it is invalid to transfer $15 from it. bank.withdraw(10, 50); // return false, it is invalid because account 10 does not exist.

Constraints:

n == balance.length

1 <= n, account, account1, account2 <= 10

5

0 <= balance[i], money <= 10

12

At most

10

4

calls will be made to

each

function

transfer

,

deposit

,

withdraw

.

Code Snippets

C++:

```
class Bank {
public:
Bank(vector<long long>& balance) {

}

bool transfer(int account1, int account2, long long money) {

}

bool deposit(int account, long long money) {

}

bool withdraw(int account, long long money) {

}
};
```

```
/**
* Your Bank object will be instantiated and called as such:
* Bank* obj = new Bank(balance);
* bool param_1 = obj->transfer(account1,account2,money);
* bool param_2 = obj->deposit(account,money);
* bool param_3 = obj->withdraw(account,money);
*/
```

Java:

```
class Bank {

public Bank(long[] balance) {

}

public boolean transfer(int account1, int account2, long money) {

}

public boolean deposit(int account, long money) {

}

public boolean withdraw(int account, long money) {

}
}

/**
* Your Bank object will be instantiated and called as such:
* Bank obj = new Bank(balance);
* boolean param_1 = obj.transfer(account1,account2,money);
* boolean param_2 = obj.deposit(account,money);
* boolean param_3 = obj.withdraw(account,money);
*/
```

Python3:

```
class Bank:

def __init__(self, balance: List[int]):
```

```
def transfer(self, account1: int, account2: int, money: int) -> bool:


def deposit(self, account: int, money: int) -> bool:


def withdraw(self, account: int, money: int) -> bool:



# Your Bank object will be instantiated and called as such:
# obj = Bank(balance)
# param_1 = obj.transfer(account1,account2,money)
# param_2 = obj.deposit(account,money)
# param_3 = obj.withdraw(account,money)
```

Python:

```
class Bank(object):

def __init__(self, balance):
"""
:type balance: List[int]
"""



def transfer(self, account1, account2, money):
"""
:type account1: int
:type account2: int
:type money: int
:rtype: bool
"""



def deposit(self, account, money):
"""
:type account: int
:type money: int
:rtype: bool
```

```
"""


def withdraw(self, account, money):
"""
:type account: int
:type money: int
:rtype: bool
"""



# Your Bank object will be instantiated and called as such:
# obj = Bank(balance)
# param_1 = obj.transfer(account1,account2,money)
# param_2 = obj.deposit(account,money)
# param_3 = obj.withdraw(account,money)
```

JavaScript:

```
/**
* @param {number[]} balance
*/
var Bank = function(balance) {

};

/**
* @param {number} account1
* @param {number} account2
* @param {number} money
* @return {boolean}
*/
Bank.prototype.transfer = function(account1, account2, money) {

};

/**
* @param {number} account
* @param {number} money
* @return {boolean}
*/
```

```
Bank.prototype.deposit = function(account, money) {

};

/**
* @param {number} account
* @param {number} money
* @return {boolean}
*/
Bank.prototype.withdraw = function(account, money) {

};

/**
* Your Bank object will be instantiated and called as such:
* var obj = new Bank(balance)
* var param_1 = obj.transfer(account1,account2,money)
* var param_2 = obj.deposit(account,money)
* var param_3 = obj.withdraw(account,money)
*/
```

TypeScript:

```
class Bank {
constructor(balance: number[]) {

}

transfer(account1: number, account2: number, money: number): boolean {

}

deposit(account: number, money: number): boolean {

}

withdraw(account: number, money: number): boolean {

}
}

/**
```

```
* Your Bank object will be instantiated and called as such:
* var obj = new Bank(balance)
* var param_1 = obj.transfer(account1,account2,money)
* var param_2 = obj.deposit(account,money)
* var param_3 = obj.withdraw(account,money)
*/
```

C#:

```
public class Bank {

public Bank(long[] balance) {

}

public bool Transfer(int account1, int account2, long money) {

}

public bool Deposit(int account, long money) {

}

public bool Withdraw(int account, long money) {

}
}

/**
* Your Bank object will be instantiated and called as such:
* Bank obj = new Bank(balance);
* bool param_1 = obj.Transfer(account1,account2,money);
* bool param_2 = obj.Deposit(account,money);
* bool param_3 = obj.Withdraw(account,money);
*/
```

C:

```
typedef struct {
```

```
} Bank;


Bank* bankCreate(long long* balance, int balanceSize) {

}

bool bankTransfer(Bank* obj, int account1, int account2, long long money) {

}

bool bankDeposit(Bank* obj, int account, long long money) {

}

bool bankWithdraw(Bank* obj, int account, long long money) {

}

void bankFree(Bank* obj) {

}

/**
* Your Bank struct will be instantiated and called as such:
* Bank* obj = bankCreate(balance, balanceSize);
* bool param_1 = bankTransfer(obj, account1, account2, money);

* bool param_2 = bankDeposit(obj, account, money);

* bool param_3 = bankWithdraw(obj, account, money);

* bankFree(obj);
*/
```

Go:

```
type Bank struct {

}
```

```
func Constructor(balance []int64) Bank {

}

func (this *Bank) Transfer(account1 int, account2 int, money int64) bool {

}

func (this *Bank) Deposit(account int, money int64) bool {

}

func (this *Bank) Withdraw(account int, money int64) bool {

}

/**
* Your Bank object will be instantiated and called as such:
* obj := Constructor(balance);
* param_1 := obj.Transfer(account1,account2,money);
* param_2 := obj.Deposit(account,money);
* param_3 := obj.Withdraw(account,money);
*/
```

Kotlin:

```
class Bank(balance: LongArray) {

fun transfer(account1: Int, account2: Int, money: Long): Boolean {

}

fun deposit(account: Int, money: Long): Boolean {

}

fun withdraw(account: Int, money: Long): Boolean {
```

```
}

}

/**
* Your Bank object will be instantiated and called as such:
* var obj = Bank(balance)
* var param_1 = obj.transfer(account1,account2,money)
* var param_2 = obj.deposit(account,money)
* var param_3 = obj.withdraw(account,money)
*/
```

Swift:

```
class Bank {

init(_ balance: [Int]) {

}

func transfer(_ account1: Int, _ account2: Int, _ money: Int) -> Bool {

}

func deposit(_ account: Int, _ money: Int) -> Bool {

}

func withdraw(_ account: Int, _ money: Int) -> Bool {

}
}

/**
* Your Bank object will be instantiated and called as such:
* let obj = Bank(balance)
* let ret_1: Bool = obj.transfer(account1, account2, money)
* let ret_2: Bool = obj.deposit(account, money)
* let ret_3: Bool = obj.withdraw(account, money)
*/
```

Rust:

```
struct Bank {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Bank {

fn new(balance: Vec<i64>) -> Self {

}

fn transfer(&self, account1: i32, account2: i32, money: i64) -> bool {

}

fn deposit(&self, account: i32, money: i64) -> bool {

}

fn withdraw(&self, account: i32, money: i64) -> bool {

}
}

/**
* Your Bank object will be instantiated and called as such:
* let obj = Bank::new(balance);
* let ret_1: bool = obj.transfer(account1, account2, money);
* let ret_2: bool = obj.deposit(account, money);
* let ret_3: bool = obj.withdraw(account, money);
*/
```

Ruby:

```
class Bank
```

```
=begin
:type balance: Integer[]
=end
def initialize(balance)

end


=begin
:type account1: Integer
:type account2: Integer
:type money: Integer
:rtype: Boolean
=end
def transfer(account1, account2, money)

end


=begin
:type account: Integer
:type money: Integer
:rtype: Boolean
=end
def deposit(account, money)

end


=begin
:type account: Integer
:type money: Integer
:rtype: Boolean
=end
def withdraw(account, money)

end



end

# Your Bank object will be instantiated and called as such:
```

```
# obj = Bank.new(balance)
# param_1 = obj.transfer(account1, account2, money)
# param_2 = obj.deposit(account, money)
# param_3 = obj.withdraw(account, money)
```

PHP:

```
class Bank {
/**
* @param Integer[] $balance
*/
function __construct($balance) {

}

/**
* @param Integer $account1
* @param Integer $account2
* @param Integer $money
* @return Boolean
*/
function transfer($account1, $account2, $money) {

}

/**
* @param Integer $account
* @param Integer $money
* @return Boolean
*/
function deposit($account, $money) {

}

/**
* @param Integer $account
* @param Integer $money
* @return Boolean
*/
function withdraw($account, $money) {

}
```

```
}

/**
* Your Bank object will be instantiated and called as such:
* $obj = Bank($balance);
* $ret_1 = $obj->transfer($account1, $account2, $money);
* $ret_2 = $obj->deposit($account, $money);
* $ret_3 = $obj->withdraw($account, $money);
*/
```

Dart:

```
class Bank {

Bank(List<int> balance) {

}

bool transfer(int account1, int account2, int money) {

}

bool deposit(int account, int money) {

}

bool withdraw(int account, int money) {

}
}

/**
* Your Bank object will be instantiated and called as such:
* Bank obj = Bank(balance);
* bool param1 = obj.transfer(account1,account2,money);
* bool param2 = obj.deposit(account,money);
* bool param3 = obj.withdraw(account,money);
*/
```

Scala:

```
class Bank(_balance: Array[Long]) {

def transfer(account1: Int, account2: Int, money: Long): Boolean = {

}

def deposit(account: Int, money: Long): Boolean = {

}

def withdraw(account: Int, money: Long): Boolean = {

}

}

/**
* Your Bank object will be instantiated and called as such:
* val obj = new Bank(balance)
* val param_1 = obj.transfer(account1,account2,money)
* val param_2 = obj.deposit(account,money)
* val param_3 = obj.withdraw(account,money)
*/
```

Elixir:

```
defmodule Bank do
@spec init_(balance :: [integer]) :: any
def init_(balance) do

end

@spec transfer(account1 :: integer, account2 :: integer, money :: integer) ::
boolean
def transfer(account1, account2, money) do

end

@spec deposit(account :: integer, money :: integer) :: boolean
def deposit(account, money) do

end
```

```
@spec withdraw(account :: integer, money :: integer) :: boolean
def withdraw(account, money) do

end
end

# Your functions will be called as such:
# Bank.init_(balance)
# param_1 = Bank.transfer(account1, account2, money)
# param_2 = Bank.deposit(account, money)
# param_3 = Bank.withdraw(account, money)

# Bank.init_ will be called before every test case, in which you can do some
necessary initializations.
```

Erlang:

```
-spec bank_init_(Balance :: [integer()]) -> any().
bank_init_(Balance) ->
.

-spec bank_transfer(Account1 :: integer(), Account2 :: integer(), Money ::
integer()) -> boolean().
bank_transfer(Account1, Account2, Money) ->
.

-spec bank_deposit(Account :: integer(), Money :: integer()) -> boolean().
bank_deposit(Account, Money) ->
.

-spec bank_withdraw(Account :: integer(), Money :: integer()) -> boolean().
bank_withdraw(Account, Money) ->
.


%% Your functions will be called as such:
%% bank_init_(Balance),
%% Param_1 = bank_transfer(Account1, Account2, Money),
%% Param_2 = bank_deposit(Account, Money),
%% Param_3 = bank_withdraw(Account, Money),

%% bank_init_ will be called before every test case, in which you can do some
```

```
necessary initializations.
```

Racket:

```
(define bank%
(class object%
(super-new)

; balance : (listof exact-integer?)
(init-field
balance)

; transfer : exact-integer? exact-integer? exact-integer? -> boolean?
(define/public (transfer account1 account2 money)
)
; deposit : exact-integer? exact-integer? -> boolean?
(define/public (deposit account money)
)
; withdraw : exact-integer? exact-integer? -> boolean?
(define/public (withdraw account money)
)))

;; Your bank% object will be instantiated and called as such:
;; (define obj (new bank% [balance balance]))
;; (define param_1 (send obj transfer account1 account2 money))
;; (define param_2 (send obj deposit account money))
;; (define param_3 (send obj withdraw account money))
```

Solutions

C++ Solution:

```
/*
* Problem: Simple Bank System
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/
```

```
class Bank {
public:
Bank(vector<long long>& balance) {

}

bool transfer(int account1, int account2, long long money) {

}

bool deposit(int account, long long money) {

}

bool withdraw(int account, long long money) {

}
};

/**
* Your Bank object will be instantiated and called as such:
* Bank* obj = new Bank(balance);
* bool param_1 = obj->transfer(account1,account2,money);
* bool param_2 = obj->deposit(account,money);
* bool param_3 = obj->withdraw(account,money);
*/
```

Java Solution:

```
/**
* Problem: Simple Bank System
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Bank {
```

```
public Bank(long[] balance) {

}

public boolean transfer(int account1, int account2, long money) {

}

public boolean deposit(int account, long money) {

}

public boolean withdraw(int account, long money) {

}
}

/**
* Your Bank object will be instantiated and called as such:
* Bank obj = new Bank(balance);
* boolean param_1 = obj.transfer(account1,account2,money);
* boolean param_2 = obj.deposit(account,money);
* boolean param_3 = obj.withdraw(account,money);
*/
```

Python3 Solution:

```
"""
Problem: Simple Bank System
Difficulty: Medium
Tags: array, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Bank:

def __init__(self, balance: List[int]):
```

```
def transfer(self, account1: int, account2: int, money: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Bank(object):

def __init__(self, balance):
"""
:type balance: List[int]
"""


def transfer(self, account1, account2, money):
"""
:type account1: int
:type account2: int
:type money: int
:rtype: bool
"""


def deposit(self, account, money):
"""
:type account: int
:type money: int
:rtype: bool
"""


def withdraw(self, account, money):
"""
:type account: int
:type money: int
:rtype: bool
"""
```

```
# Your Bank object will be instantiated and called as such:
# obj = Bank(balance)
# param_1 = obj.transfer(account1,account2,money)
# param_2 = obj.deposit(account,money)
# param_3 = obj.withdraw(account,money)
```

JavaScript Solution:

```
/**
* Problem: Simple Bank System
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number[]} balance
*/
var Bank = function(balance) {

};

/**
* @param {number} account1
* @param {number} account2
* @param {number} money
* @return {boolean}
*/
Bank.prototype.transfer = function(account1, account2, money) {

};

/**
* @param {number} account
* @param {number} money
* @return {boolean}
*/
```

```
Bank.prototype.deposit = function(account, money) {

};

/**
* @param {number} account
* @param {number} money
* @return {boolean}
*/
Bank.prototype.withdraw = function(account, money) {

};

/**
* Your Bank object will be instantiated and called as such:
* var obj = new Bank(balance)
* var param_1 = obj.transfer(account1,account2,money)
* var param_2 = obj.deposit(account,money)
* var param_3 = obj.withdraw(account,money)
*/
```

TypeScript Solution:

```
/**
* Problem: Simple Bank System
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Bank {
constructor(balance: number[]) {

}

transfer(account1: number, account2: number, money: number): boolean {

}
```

```
deposit(account: number, money: number): boolean {

}

withdraw(account: number, money: number): boolean {

}
}

/**
* Your Bank object will be instantiated and called as such:
* var obj = new Bank(balance)
* var param_1 = obj.transfer(account1,account2,money)
* var param_2 = obj.deposit(account,money)
* var param_3 = obj.withdraw(account,money)
*/
```

C# Solution:

```
/*
* Problem: Simple Bank System
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Bank {

public Bank(long[] balance) {

}

public bool Transfer(int account1, int account2, long money) {

}

public bool Deposit(int account, long money) {
```

```
}

public bool Withdraw(int account, long money) {

}
}

/**
* Your Bank object will be instantiated and called as such:
* Bank obj = new Bank(balance);
* bool param_1 = obj.Transfer(account1,account2,money);
* bool param_2 = obj.Deposit(account,money);
* bool param_3 = obj.Withdraw(account,money);
*/
```

C Solution:

```
/*
* Problem: Simple Bank System
* Difficulty: Medium
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/




typedef struct {

} Bank;


Bank* bankCreate(long long* balance, int balanceSize) {

}

bool bankTransfer(Bank* obj, int account1, int account2, long long money) {
```

```
}

bool bankDeposit(Bank* obj, int account, long long money) {

}

bool bankWithdraw(Bank* obj, int account, long long money) {

}

void bankFree(Bank* obj) {

}

/**
* Your Bank struct will be instantiated and called as such:
* Bank* obj = bankCreate(balance, balanceSize);
* bool param_1 = bankTransfer(obj, account1, account2, money);

* bool param_2 = bankDeposit(obj, account, money);

* bool param_3 = bankWithdraw(obj, account, money);

* bankFree(obj);
*/
```

Go Solution:

```
// Problem: Simple Bank System
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type Bank struct {

}
```

```
func Constructor(balance []int64) Bank {

}

func (this *Bank) Transfer(account1 int, account2 int, money int64) bool {

}

func (this *Bank) Deposit(account int, money int64) bool {

}

func (this *Bank) Withdraw(account int, money int64) bool {

}

/**
* Your Bank object will be instantiated and called as such:
* obj := Constructor(balance);
* param_1 := obj.Transfer(account1,account2,money);
* param_2 := obj.Deposit(account,money);
* param_3 := obj.Withdraw(account,money);
*/
```

Kotlin Solution:

```
class Bank(balance: LongArray) {

fun transfer(account1: Int, account2: Int, money: Long): Boolean {

}

fun deposit(account: Int, money: Long): Boolean {

}
```

```
fun withdraw(account: Int, money: Long): Boolean {

}

}

/**
* Your Bank object will be instantiated and called as such:
* var obj = Bank(balance)
* var param_1 = obj.transfer(account1,account2,money)
* var param_2 = obj.deposit(account,money)
* var param_3 = obj.withdraw(account,money)
*/
```

Swift Solution:

```
class Bank {

init(_ balance: [Int]) {

}

func transfer(_ account1: Int, _ account2: Int, _ money: Int) -> Bool {

}

func deposit(_ account: Int, _ money: Int) -> Bool {

}

func withdraw(_ account: Int, _ money: Int) -> Bool {

}
}

/**
* Your Bank object will be instantiated and called as such:
* let obj = Bank(balance)
* let ret_1: Bool = obj.transfer(account1, account2, money)
* let ret_2: Bool = obj.deposit(account, money)
```

```
* let ret_3: Bool = obj.withdraw(account, money)
*/
```

Rust Solution:

```
// Problem: Simple Bank System
// Difficulty: Medium
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct Bank {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Bank {

fn new(balance: Vec<i64>) -> Self {

}

fn transfer(&self, account1: i32, account2: i32, money: i64) -> bool {

}

fn deposit(&self, account: i32, money: i64) -> bool {

}

fn withdraw(&self, account: i32, money: i64) -> bool {

}
}
```

```
/**
* Your Bank object will be instantiated and called as such:
* let obj = Bank::new(balance);
* let ret_1: bool = obj.transfer(account1, account2, money);
* let ret_2: bool = obj.deposit(account, money);
* let ret_3: bool = obj.withdraw(account, money);
*/
```

Ruby Solution:

```
class Bank

=begin
:type balance: Integer[]
=end
def initialize(balance)

end


=begin
:type account1: Integer
:type account2: Integer
:type money: Integer
:rtype: Boolean
=end
def transfer(account1, account2, money)

end


=begin
:type account: Integer
:type money: Integer
:rtype: Boolean
=end
def deposit(account, money)

end
```

```
=begin
:type account: Integer
:type money: Integer
:rtype: Boolean
=end
def withdraw(account, money)

end



end

# Your Bank object will be instantiated and called as such:
# obj = Bank.new(balance)
# param_1 = obj.transfer(account1, account2, money)
# param_2 = obj.deposit(account, money)
# param_3 = obj.withdraw(account, money)
```

PHP Solution:

```
class Bank {
/**
* @param Integer[] $balance
*/
function __construct($balance) {

}

/**
* @param Integer $account1
* @param Integer $account2
* @param Integer $money
* @return Boolean
*/
function transfer($account1, $account2, $money) {

}

/**
* @param Integer $account
* @param Integer $money
```

```
* @return Boolean
*/
function deposit($account, $money) {

}

/**
* @param Integer $account
* @param Integer $money
* @return Boolean
*/
function withdraw($account, $money) {

}
}

/**
* Your Bank object will be instantiated and called as such:
* $obj = Bank($balance);
* $ret_1 = $obj->transfer($account1, $account2, $money);
* $ret_2 = $obj->deposit($account, $money);
* $ret_3 = $obj->withdraw($account, $money);
*/
```

Dart Solution:

```
class Bank {

Bank(List<int> balance) {

}

bool transfer(int account1, int account2, int money) {

}

bool deposit(int account, int money) {

}

bool withdraw(int account, int money) {
```

```
}
}

/**
* Your Bank object will be instantiated and called as such:
* Bank obj = Bank(balance);
* bool param1 = obj.transfer(account1,account2,money);
* bool param2 = obj.deposit(account,money);
* bool param3 = obj.withdraw(account,money);
*/
```

Scala Solution:

```
class Bank(_balance: Array[Long]) {

def transfer(account1: Int, account2: Int, money: Long): Boolean = {

}

def deposit(account: Int, money: Long): Boolean = {

}

def withdraw(account: Int, money: Long): Boolean = {

}

}

/**
* Your Bank object will be instantiated and called as such:
* val obj = new Bank(balance)
* val param_1 = obj.transfer(account1,account2,money)
* val param_2 = obj.deposit(account,money)
* val param_3 = obj.withdraw(account,money)
*/
```

Elixir Solution:

```
defmodule Bank do
@spec init_(balance :: [integer]) :: any
def init_(balance) do

end

@spec transfer(account1 :: integer, account2 :: integer, money :: integer) ::
boolean
def transfer(account1, account2, money) do

end

@spec deposit(account :: integer, money :: integer) :: boolean
def deposit(account, money) do

end

@spec withdraw(account :: integer, money :: integer) :: boolean
def withdraw(account, money) do

end
end

# Your functions will be called as such:
# Bank.init_(balance)
# param_1 = Bank.transfer(account1, account2, money)
# param_2 = Bank.deposit(account, money)
# param_3 = Bank.withdraw(account, money)

# Bank.init_ will be called before every test case, in which you can do some
necessary initializations.
```

Erlang Solution:

```
-spec bank_init_(Balance :: [integer()]) -> any().
bank_init_(Balance) ->
.

-spec bank_transfer(Account1 :: integer(), Account2 :: integer(), Money ::
integer()) -> boolean().
bank_transfer(Account1, Account2, Money) ->
.
```

```
-spec bank_deposit(Account :: integer(), Money :: integer()) -> boolean().
bank_deposit(Account, Money) ->
.

-spec bank_withdraw(Account :: integer(), Money :: integer()) -> boolean().
bank_withdraw(Account, Money) ->
.


%% Your functions will be called as such:
%% bank_init_(Balance),
%% Param_1 = bank_transfer(Account1, Account2, Money),
%% Param_2 = bank_deposit(Account, Money),
%% Param_3 = bank_withdraw(Account, Money),

%% bank_init_ will be called before every test case, in which you can do some
necessary initializations.
```

Racket Solution:

```
(define bank%
(class object%
(super-new)

; balance : (listof exact-integer?)
(init-field
balance)

; transfer : exact-integer? exact-integer? exact-integer? -> boolean?
(define/public (transfer account1 account2 money)
)
; deposit : exact-integer? exact-integer? -> boolean?
(define/public (deposit account money)
)
; withdraw : exact-integer? exact-integer? -> boolean?
(define/public (withdraw account money)
)))

;; Your bank% object will be instantiated and called as such:
;; (define obj (new bank% [balance balance]))
```

```
;; (define param_1 (send obj transfer account1 account2 money))
;; (define param_2 (send obj deposit account money))
;; (define param_3 (send obj withdraw account money))
```