

UML for Java Developers

Model Constraints & The Object Constraint Language

Jason Gorman



If you're reading this tutorial, you're probably either studying for or teaching a computing or software engineering academic qualification.

How do I know this?

Simple. In the real world, almost nobody uses OCL. And by "almost nobody", I mean maybe 1/100 software professionals may have learned it. And maybe 1/100 of them ever use it. Learning OCL very probably is not going to get you a job as anything other than someone who teaches OCL.

I believe it is a useful skill to have if you want to really get to grips with UML, but I can state categorically, with my hand on my heart, that I have in my entire career used OCL in anger maybe twice.

Knowing OCL will not make you a better software developer, and you are unlikely to work with other software developers who know OCL, rendering it useless as a communication tool.

You may have been told about Model-driven Architecture. Back in 2000, it was going to be the next big thing. It wasn't.

On 99.9% of professional software projects, we still type code in a third-generation language into a text editor. Occasionally we draw UML diagrams on whiteboards when we want to visualise a design or analysis concept. You will find that some UML notations are still in widespread use - especially class and sequence diagrams, and activity diagrams for workflow analysis.

Consider OCL as being a classical language, like Latin or Ancient Greek.

It's useful to know, as it can give you some general background when applying things like Design By Contract, or even for functional programming. But it is, too all intents and purposes, a dead language.

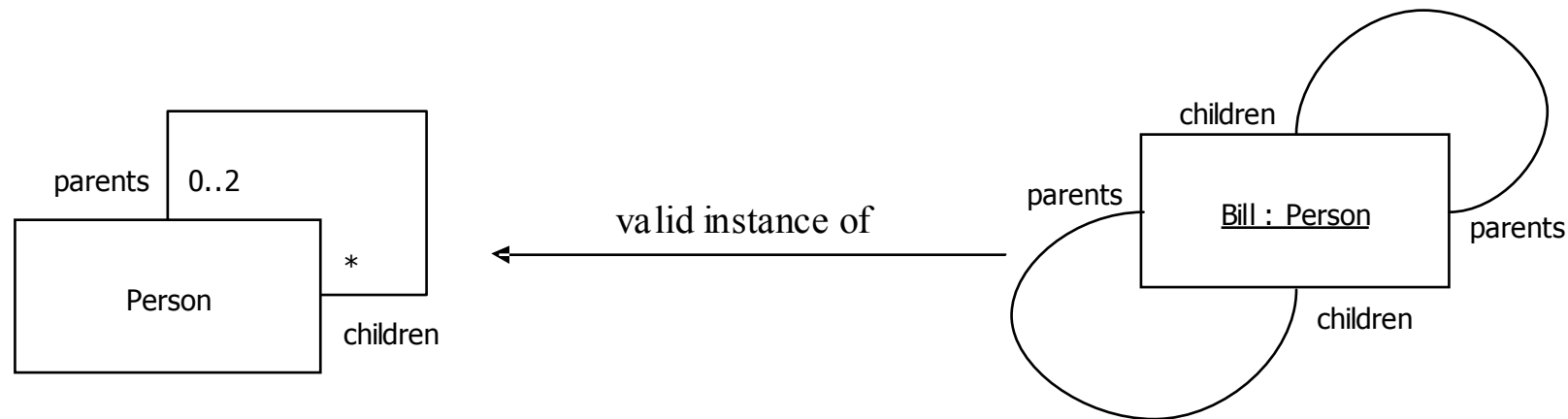
Trust me, almost nobody out here speaks it.

Having said that, I hope you find this tutorial useful in passing your exams. And I look forward to maybe teaching you some useful skills - like test-driven development or refactoring - when you graduate and join the community of professional software developers.

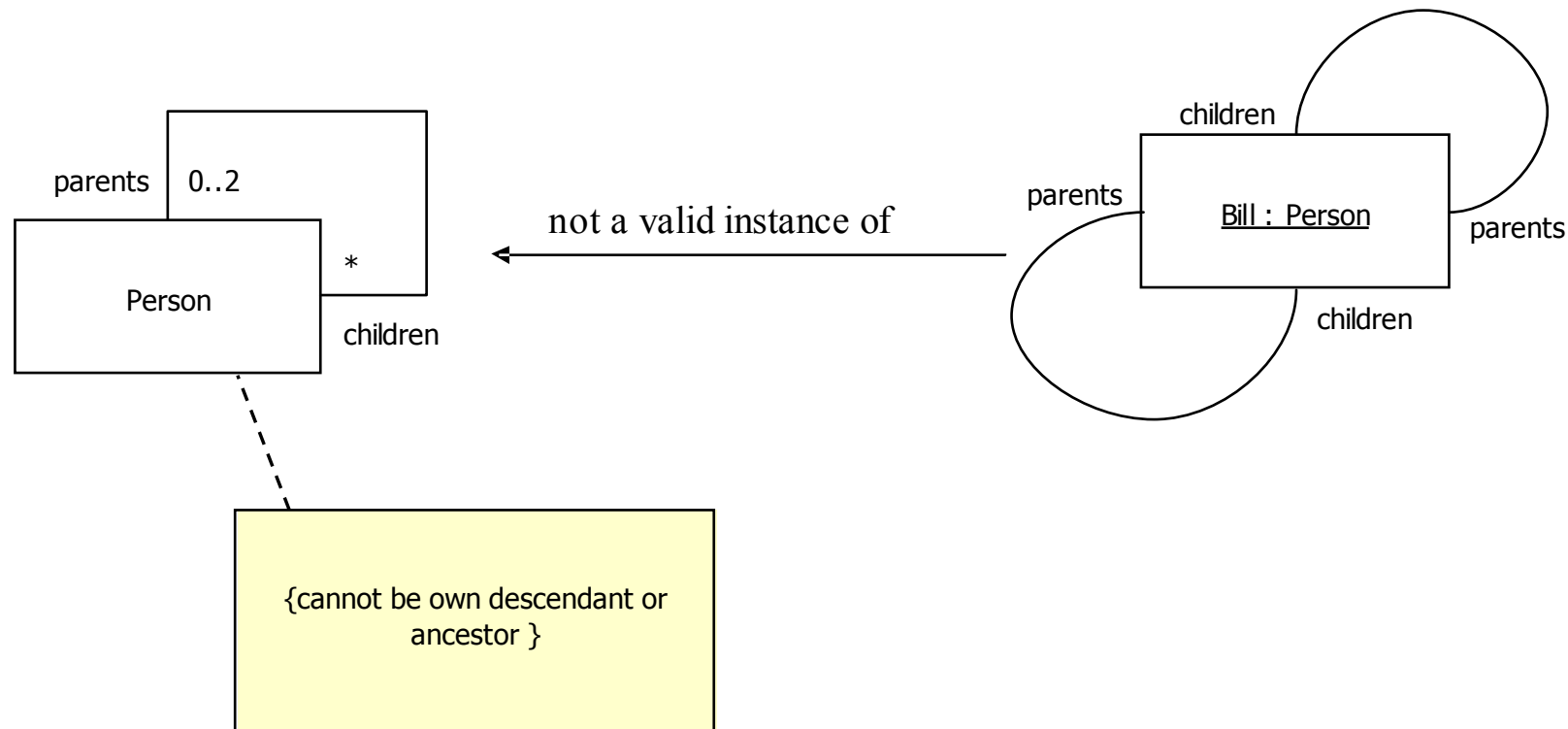
Best wishes,

Jason Gorman

UML Diagrams Don't Tell Us Everything



Constraints Make Models More Precise



What is the Object Constraint Language?

- A language for expressing necessary extra information about a model
- A precise and unambiguous language that can be read and understood by developers and customers

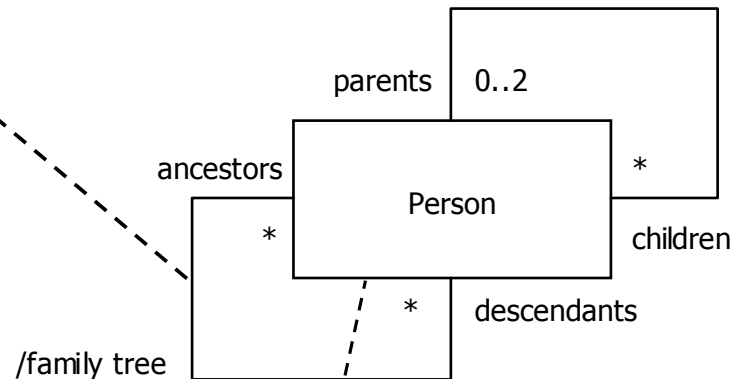
A language that is purely declarative ie, it has no side-effects (in other words it describes what rather than how)

What is an OCL Constraint?

- An OCL constraint is an OCL expression that evaluates to true or false (a Boolean OCL expression, in other words)

OCL Makes Constraints Unambiguous

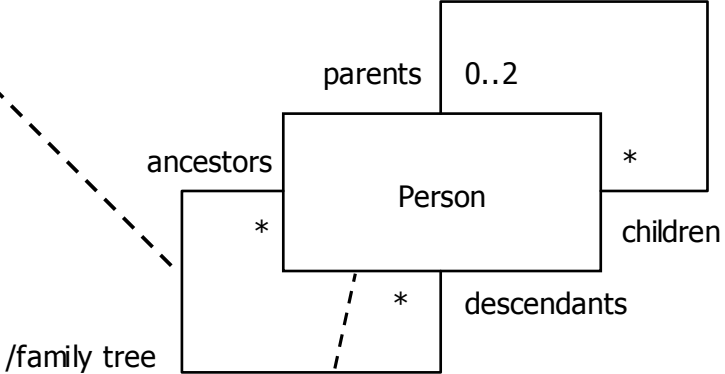
```
{ancestors = parents->union(parents.ancestors->asSet())}  
{descendants = children->union(children.descendants->asSet())}
```



```
{ancestors->excludes(self) and descendants->excludes(self) }
```

Introducing OCL Constraints & Contexts

```
{ancestors = parents->union(parents.ancestors->asSet())}
{descendants = children->union(children.descendants->asSet())}
```



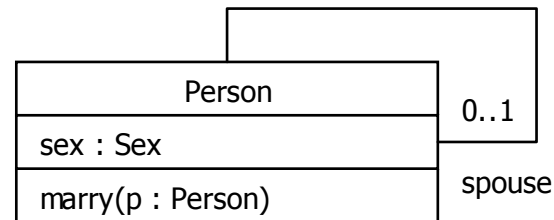
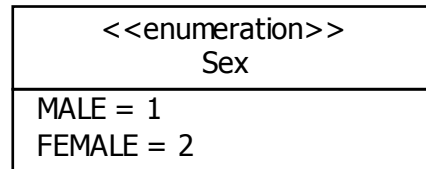
Q: To what which type this constraint apply?
 A: Person

```
context Person
inv: ancestors->excludes(self) and descendants->excludes(self)
```

Q: When does this constraint apply?
 A: inv = invariant = always

```
{ancestors->excludes(self) and descendants->excludes(self) }
```

Operations, Pre & Post-conditions



optional constraint name

applies to the marry() operation of the type Person

```
context Person::marry(p : Person)
pre cannot_marry_self: not (p= self)
pre not_same_sex: not (p.sex = self.sex)
-- neither person can be married already
pre not_already_married: self.spouse->size() = 0 and p.spouse->size() = 0
```

comments start with --

Design By Contract :assert

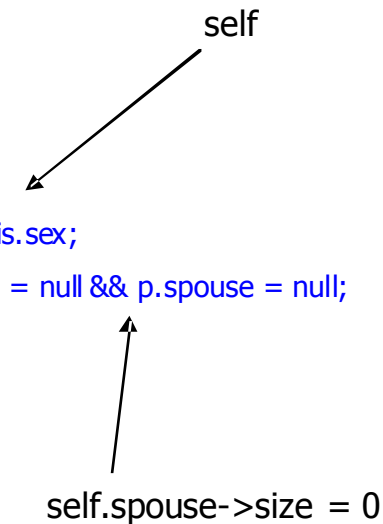
```
class Sex
{
    static final int MALE = 1;
    static final int FEMALE = 2;
}

class Person
{
    public int sex;
    public Person spouse;

    public void marry(Person p)
    {
        assert p != this;
        assert p.sex != this.sex;
        assert this.spouse = null && p.spouse = null;

        this.spouse = p;
        p.spouse = this;

        self.spouse->size = 0
    }
}
```



```
context Person::marry(p : Person)
pre cannot_marry_self: not (p = self)
pre not_same_sex: not (p.sex = self.sex)
-- neither person can be married already
pre not_already_married: self.spouse->size() = 0 and p.spouse->size() = 0
post : self.spouse = p and p.spouse = self
```

Defensive Programming : Throwing Exceptions

```
class Person
{
    public int sex;
    public Person spouse;

    public void marry(Person p) throws ArgumentException {
        if(p == this) {
            throw new ArgumentException("cannot marry self");
        }
        if(p.sex == this.sex) {
            throw new ArgumentException("spouse is same sex");
        }
        if((p.spouse != null || this.spouse != null) {
            throw new ArgumentException("already married");
        }

        this.spouse = p;
        p.spouse = this;
    }
}
```

Referring to previous values and operation return values

Account
balance : Real = 0
deposit(amount : Real) withdraw(amount : Real) getBalance() : Real

balance before execution of operation

```
context Account::withdraw (amount : Real)
pre: amount <= balance
post: balance = balance@pre - amount

context Account::getBalance() : Real
post : result = balance
```

return value of operation

@pre and result in Java

```
context Account::withdraw(amount : Real)
pre: amount <= balance
post: balance = balance@pre - amount

context Account::getBalance() : Real
post : result = balance
```

```
class Account
{
    private float balance = 0;

    public void withdraw(float amount) {
        assert amount <= balance;

        balance = balance - amount;
    }

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

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

```
public void testWithdrawWithSufficientFunds() {
    Account account = new Account();

    account.deposit(500);

    float balanceAtPre = account.getBalance();

    float amount = 250;

    account.withdraw(amount);

    assertTrue(account.getBalance() == balanceAtPre - amount);
}
```

balance = balance@pre - amount

OCL Basic Value Types

Account
balance : Real = 0 name : String id : Integer isActive : Boolean
deposit(amount : Real) withdraw(amount : Real)

- Integer : A whole number of any size
- Real : A decimal number of any size
- String : A string of characters
- Boolean : True/False

id : Integer

int id;

long id;

byte id;

short id;

balance : Real = 0

double balance = 0;

name : String

string name;

isActive : Boolean

boolean isActive;

Operations on Real and Integer Types

Operation	Notation	Result type
equals	$a = b$	Boolean
not equals	$a \neq b$	Boolean
less	$a < b$	Boolean
more	$a > b$	Boolean
less or equal	$a \leq b$	Boolean
more or equal	$a \geq b$	Boolean
plus	$a + b$	Integer or Real
minus	$a - b$	Integer or Real
multiply	$a * b$	Integer or Real
divide	a / b	Real
modulus	$a.\text{mod}(b)$	Integer
integer division	$a.\text{div}(b)$	Integer
absolute value	$a.\text{abs}$	Integer or Real
maximum	$a.\text{max}(b)$	Integer or Real
minimum	$a.\text{min}(b)$	Integer or Real
round	$a.\text{round}$	Integer
floor	$a.\text{floor}$	Integer

Eg, $6.7.\text{floor}() = 6$

Operations on String Type

Operation	Expression	Result type
concatenation	s.concat(string)	String
size	s.size	Integer
to lower case	s.toLowerCase	String
to upper case	s.toUpperCase	String
substring	s.substring(int, int)	String
equals	s1 = s2	Boolean
not equals	s1 <> s2	Boolean

Eg, jason.concat(gorman) = jason gorman

Eg, jason.substring(1, 2) = ja

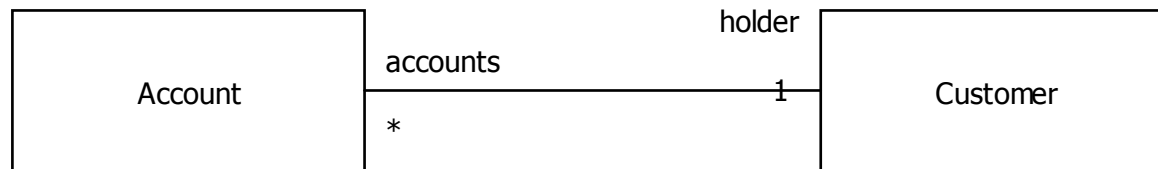
Operations on Boolean Type

Operation	Notation	Result type
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implication	a implies b	Boolean
if then else	if a then b1 else b2 endif	type of b

Eg, true or false = true

Eg, true and false = false

Navigating in OCL Expressions



In OCL:

`account.holder`

Evaluates to a customer object who is in the role holder for that association

And:

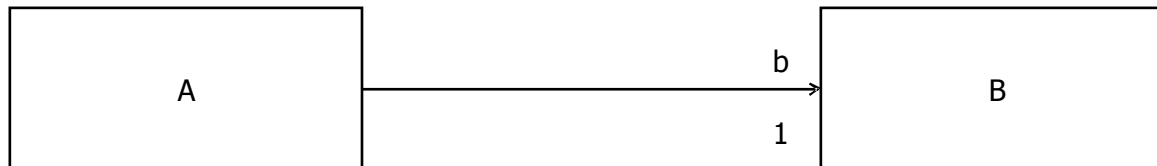
`customer.accounts`

Evaluates to a collection of Account objects in the role accounts for that association

```
Account account = new Account();
Customer customer = new Customer();

customer.accounts = new Account[] {account};
account.holder = customer;
```

Navigability in OCL Expressions



a.b is allowed

b.a is not allowed it is not navigable

```
class A
{
    public B b;
}

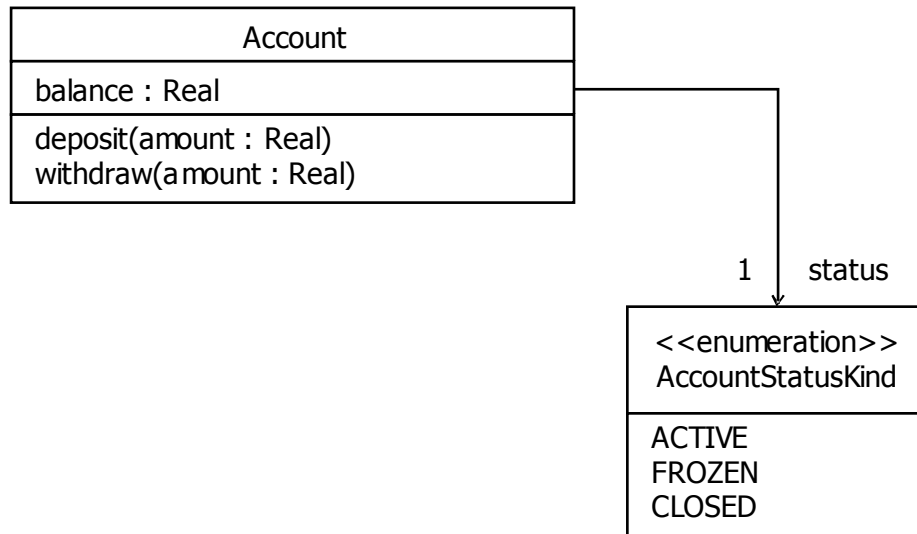
class B
{
}
```

Calling class features

Account
id : Integer status : enum{active, frozen, closed} balance : Real <u>nextId : Integer</u>
deposit(amount : Real) withdraw(amount : Real) <u>fetch(id : Integer) : Account</u>

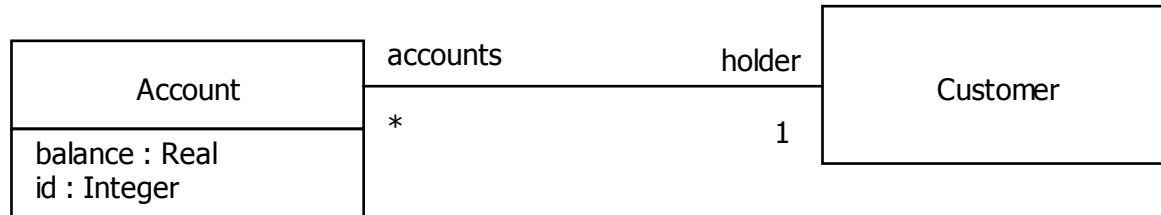
```
context Account::createNew() : Account
post: result.oclIsNew() and
      result.id = Account.nextId@pre and
      Account.nextId = result.id + 1
```

Enumerations in OCL



`context Account::withdraw(amount : Real)`
pre: amount <= balance
pre: status = AccountStatusKind.ACTIVE
post: balance = balance@pre - amount

Collections in OCL



`customer.accounts.balance = 0` is not allowed

`customer.accounts->select(id = 2324).balance = 0` is allowed

Collections in Java

```
class Account
{
    public double balance;
    public int id;
}

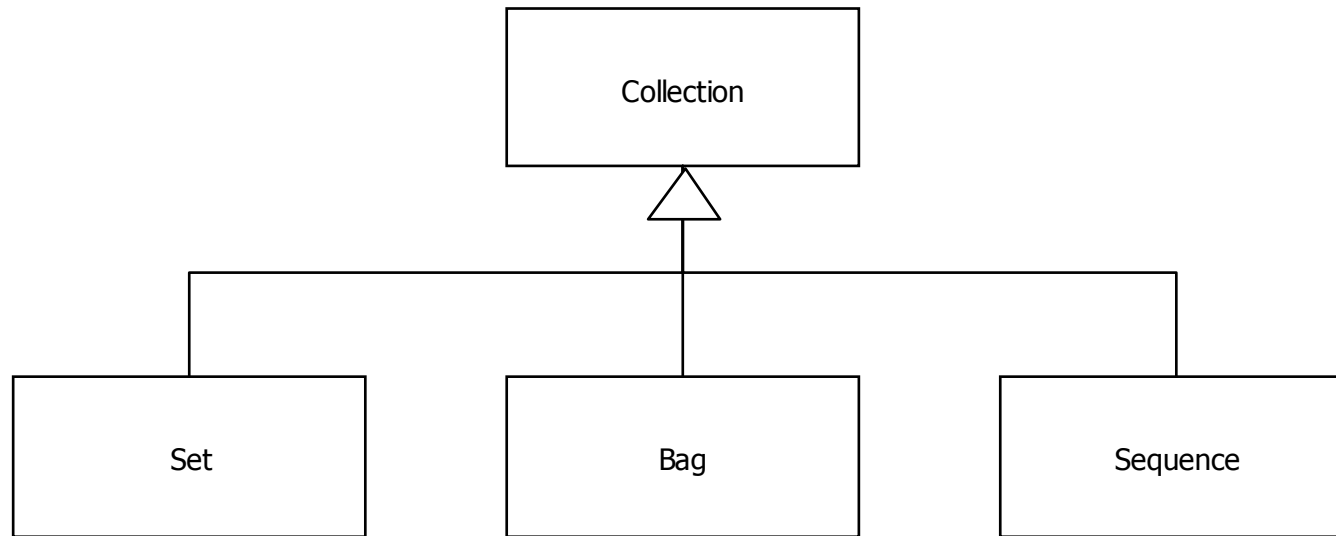
class Customer
{
    Account[] accounts;

    public Account SelectAccount(int id)
    {
        Account selected = null;

        for(int i = 0; i < accounts.length; i++)
        {
            Account account = accounts[i];
            if(account.id == id)
            {
                selected = account;
            }
        }

        return selected;
    }
}
```

The OCL Collection Hierarchy



Elements can be included only once, and in no specific order

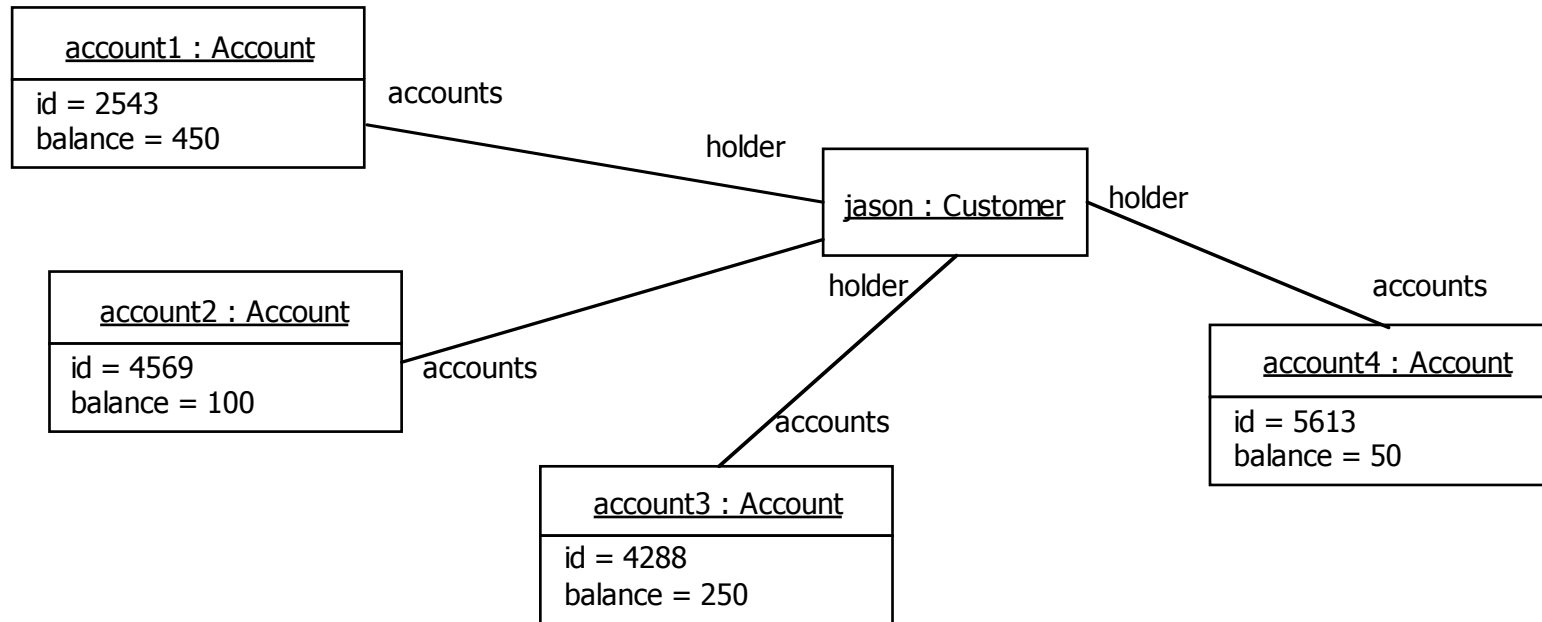
Elements can be included more than once, in no specific order

Elements can be included more than once, but in a specific order

Operations on All Collections

Operation	Description
size	The number of elements in the collection
count(object)	The number of occurrences of object in the collection.
includes(object)	True if the object is an element of the collection.
includesAll(collection)	True if all elements of the parameter collection are present in the current collection.
isEmpty	True if the collection contains no elements.
notEmpty	True if the collection contains one or more elements.
iterate(expression)	Expression is evaluated for every element in the collection.
sum(collection)	The addition of all elements in the collection.
exists(expression)	True if expression is true for at least one element in the collection.
forAll(expression)	True if expression is true for all elements.
select(expression)	Returns the subset of elements that satisfy the expression
reject(expression)	Returns the subset of elements that do not satisfy the expression
collect(expression)	Collects all of the elements given by expression into a new collection
one(expression)	Returns true if exactly one element satisfies the expression
sortedBy(expression)	Returns a Sequence of all the elements in the collection in the order specified (expression must contain the < operator)

Examples of Collection Operations



`jason.accounts->forAll(a : Account | a.balance > 0) = true`

`jason.accounts->select(balance > 100) = {account1, account3}`

`jason.accounts->includes(account4) = true`

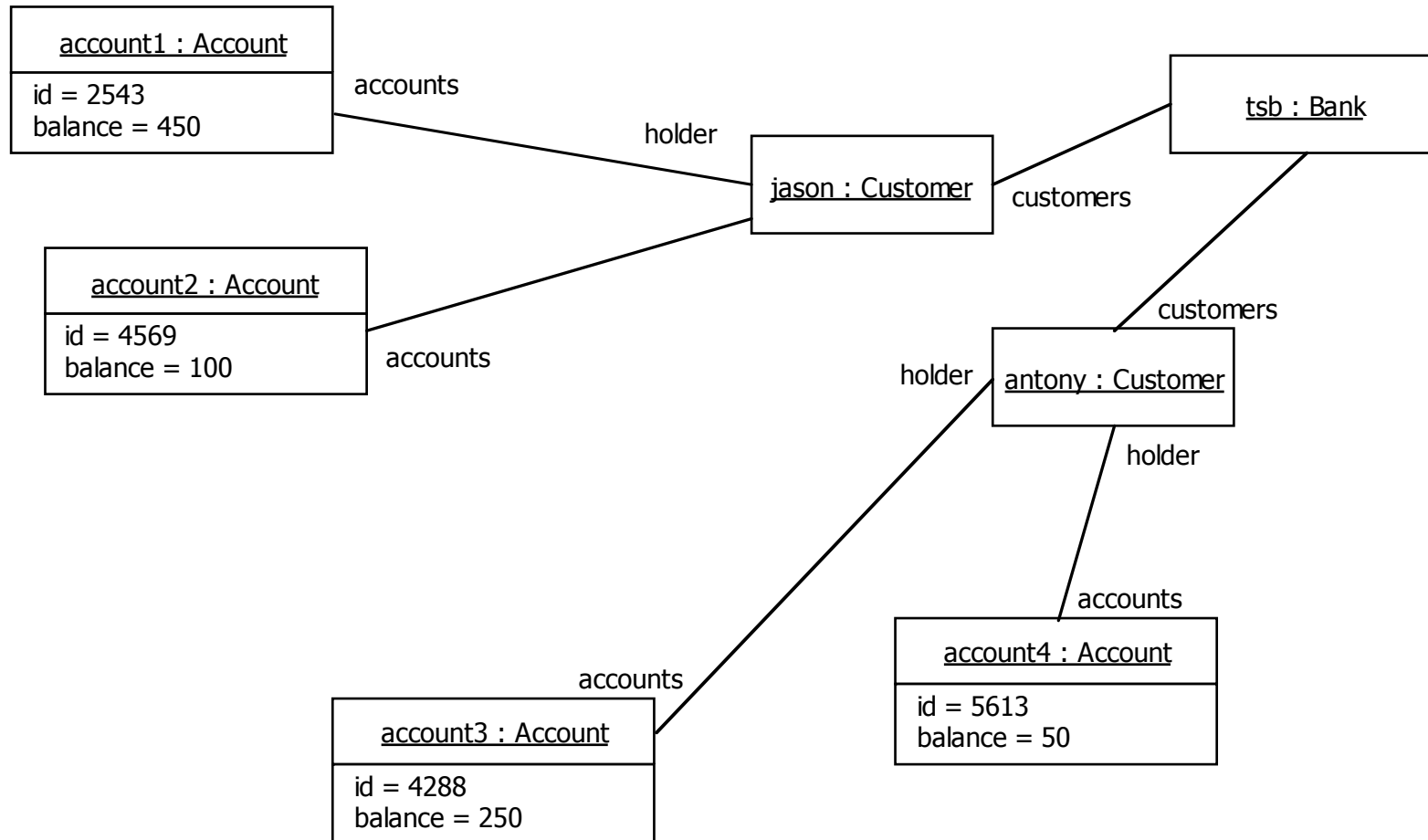
`jason.accounts->exists(a : account | a.id = 333) = false`

`jason.accounts->includesAll({account1, account2}) = true`

`jason.accounts.balance->sum() = 850`

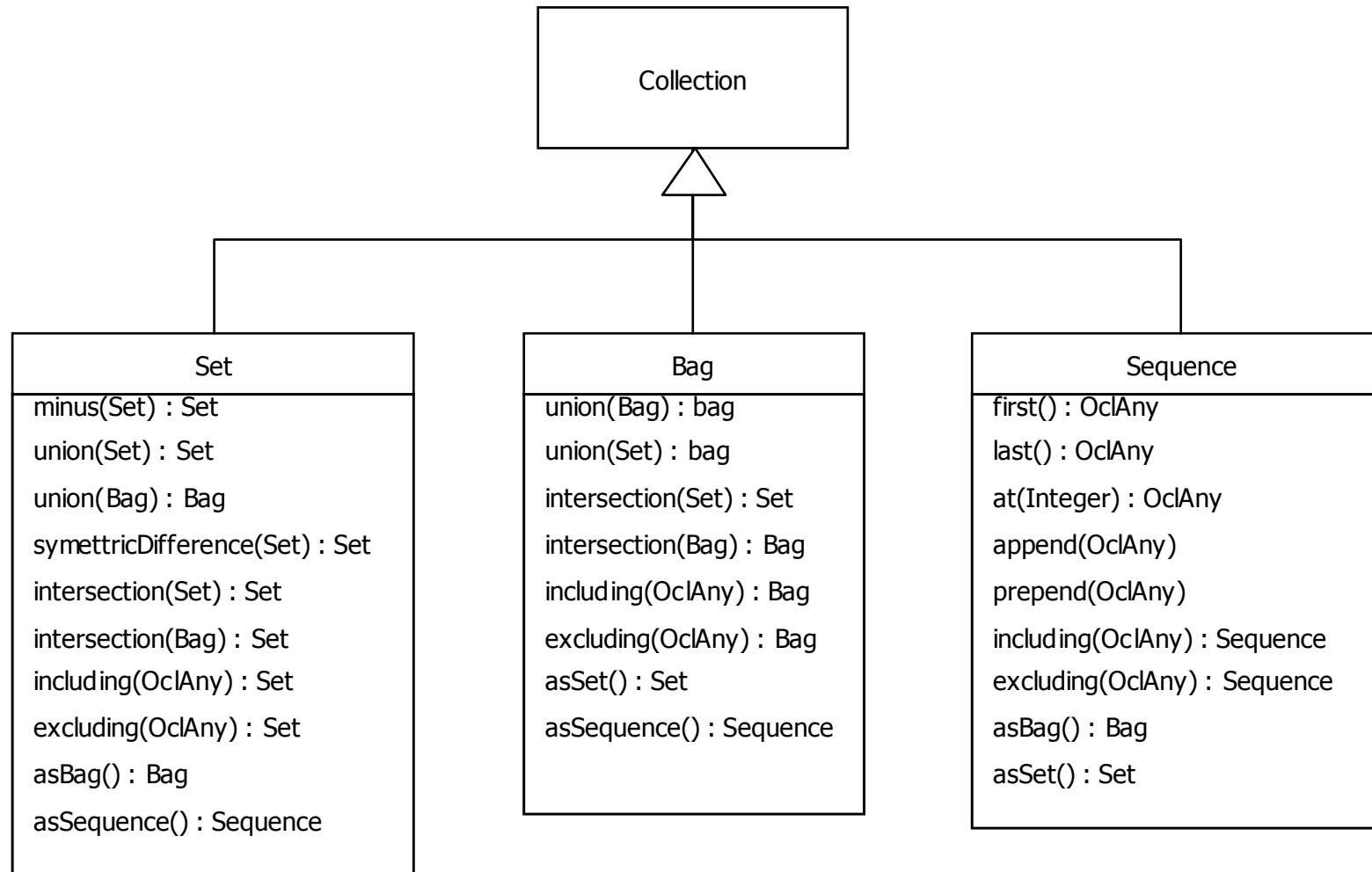
```
bool forAll = true;
foreach(Account a in accounts)
{
    if(!(a.balance > 0))
    {
        forAll = forAll && (a.balance > 0);
    }
}
```

Navigating Across & Flattening Collections



```
tsb.customers.accounts = {account1, account2, account3, account4}  
tsb.customers.accounts.balance = {450, 100, 250, 50}
```

Specialized Collection Operations

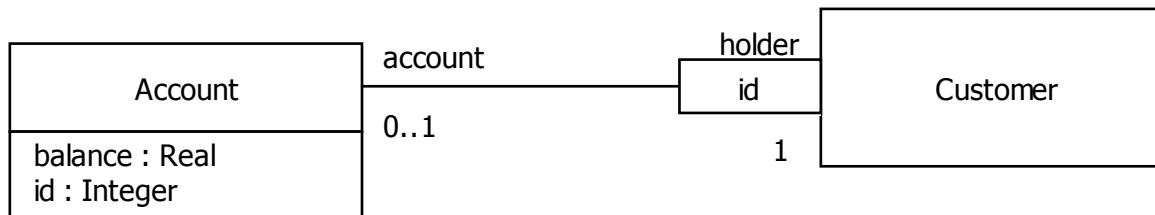


Eg, $\text{Set}\{4, 2, 3, 1\}.\text{minus}(\text{Set}\{2, 3\}) = \text{Set}\{4, 1\}$

Eg, $\text{Bag}\{1, 2, 3, 5\}.\text{including}(6) = \text{Bag}\{1, 2, 3, 5, 6\}$

Eg, $\text{Sequence}\{1, 2, 3, 4\}.\text{append}(5) = \text{Sequence}\{1, 2, 3, 4, 5\}$

Navigating across Qualified Associations

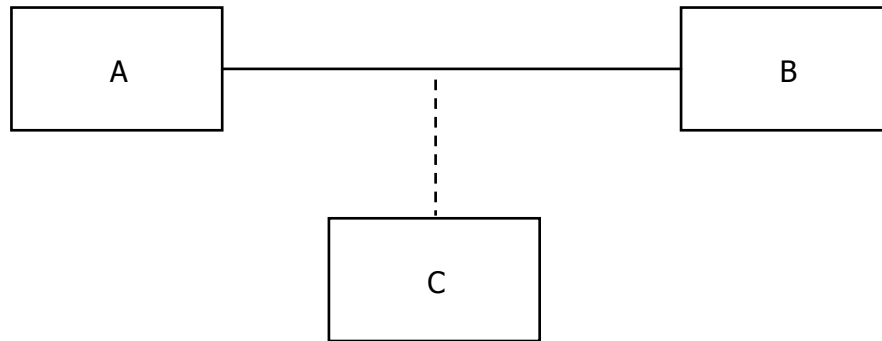


`customer.account[3435]`

Or

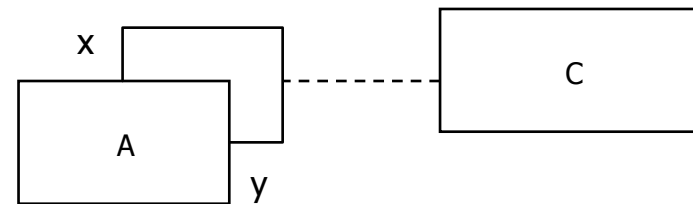
`customer.account[id = 3435]`

Navigating to Association Classes



context A inv: self.c

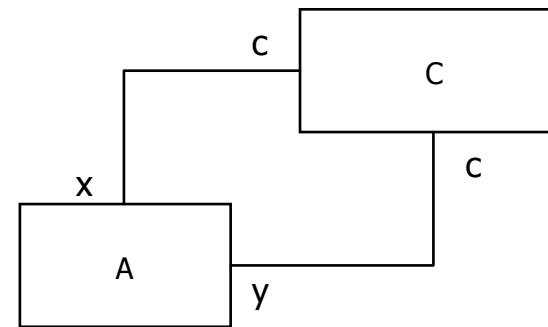
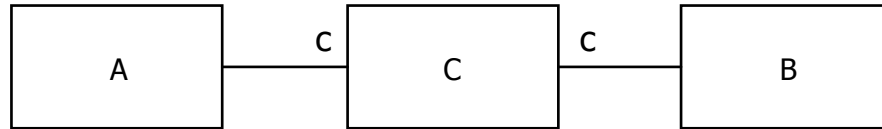
context B inv: self.c



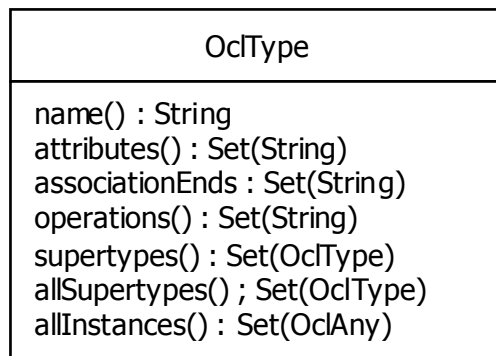
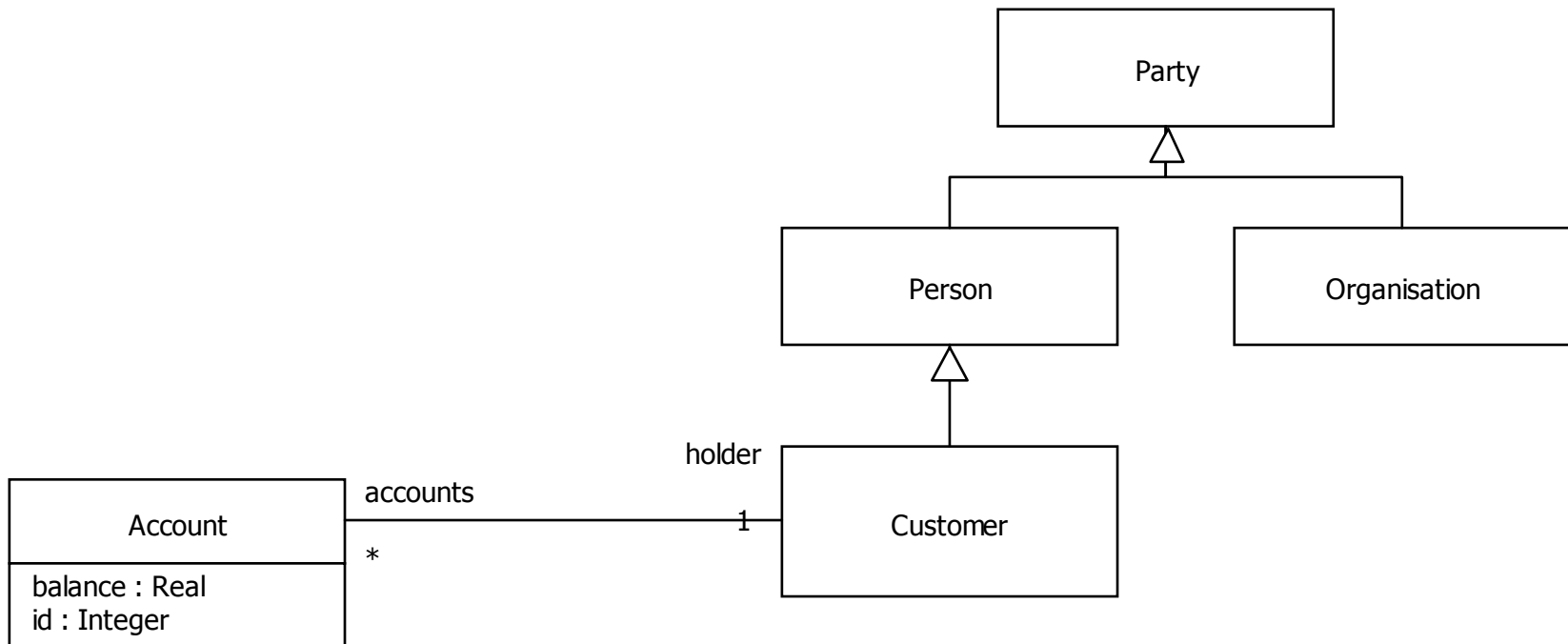
context A inv: self.c[x]

context A inv: self.c[y]

Equivalents to Association Classes



Built-in OCL Types : OclType



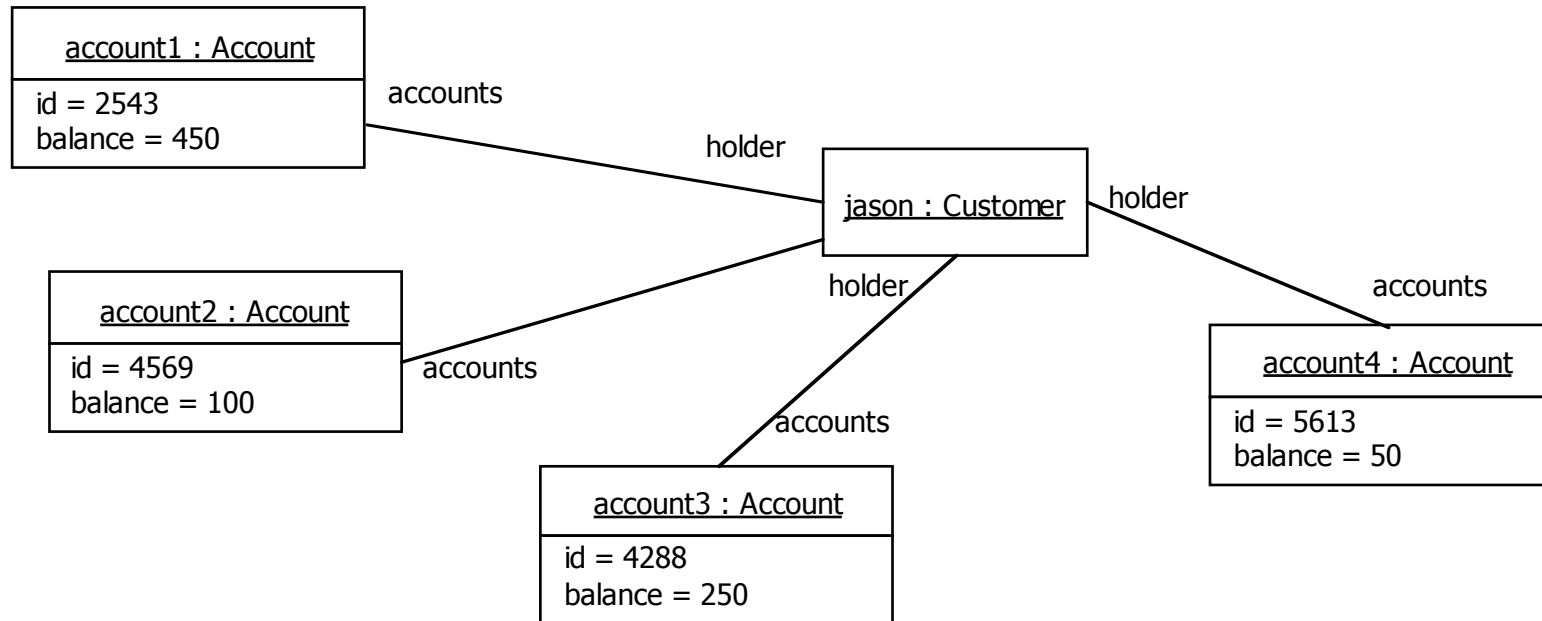
Eg, Account.name() = Account

Eg, Account.attributes() = Set{ balance , id }

Eg, Customer.supertypes() = Set{Person}

Eg, Customer.allSupertypes() = Set{Person, Party}

Built-in OCL Types : OclAny



OclAny
oclIsKindOf(OclType) : Boolean
oclIsTypeOf(OclType) : Boolean
oclAsType(OclType) : OclAny
oclInState(OclState) : Boolean
oclIsNew() : Boolean
oclType() : OclType

Eg, `jason.oclType() = Customer`

Eg, `jason.oclIsKindOf(Person) = true`

Eg, `jason.oclIsTypeOf(Person) = false`

Eg, `Account.allInstances() = Set{account1, account2, account3, account4}`

More on OCL

- [OCL 1.5 Language Specification](#)
- [OCL Evaluator](#) a tool for editing, syntax checking & evaluating OCL
- [Octopus OCL 2.0 Plug-in for Eclipse](#)

www.parlezuml.com