

UML for Java Developers

Class Diagrams



"I am currently working on a team which is [in] the process of adopting RUP and UML standards and practices. After one of our design sessions, I needed to lookup some information and came across your site. Absolutely great! Most [of] the information I've had questions about is contained within your tutorials and then some."

"Really great site... I have been trying to grasp UML since the day I saw Visual Modeler. I knew a few things but there were gaps. Thanks to your site they have shrunk considerably."

"I went on a UML training course three months ago, and came out with a big folder full of hand-outs and no real understanding of UML and how to use it on my project. I spent a day reading the UML for .NET tutorials and it was a much easier way to learn. 'Here's the diagram. Now here's the code.' Simple."



UML for Java Developers (5 Days)

Since Autumn 2003, over 100,000 Java and .NET developers have learned the Unified Modeling Language from **Parlez UML** (<http://www.parlezuml.com>), making it one of the most popular UML training resources on the Internet.

UML for Java Developers is designed to accelerate the learning process by explaining UML in a language Java developers can understand – Java!

From Requirements to a Working System

Many UML courses focus on analysis and high-level design, falling short of explaining how you get from there to a working system. **UML for Java Developers** takes you all the way from system requirements to the finished code because that, after all, is why we model in the first place.

Learning By Doing

UML modeling is a practical skill, like driving a car or flying a plane. Just as we don't learn to drive just by looking at PowerPoint presentations, you cannot properly learn UML without getting plenty of practice at it.

Your skills will be developed by designing and building a working piece of software, giving you a genuine understanding of how UML can be applied throughout the development lifecycle.

www.parlezuml.com/training.htm

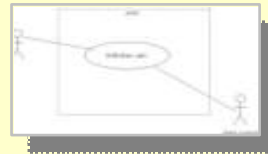
advertisement

© Jason Gorman 2005. All rights reserved.



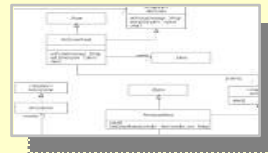


UML for Java Developers covers the most useful aspects of the UML standard, applying each notation within the context of an iterative, object oriented development process



Use Case Diagrams

Model the users of the system and the goals they can achieve by using it



Class Diagrams

Model types of objects and the relationships between them.



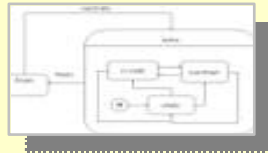
Sequence Diagrams

Model how objects interact to achieve functional goals



Activity Diagrams

Model the flow of use cases and single and multi-threaded code



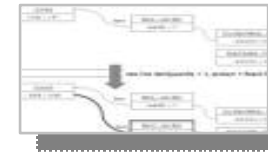
Statechart Diagrams

Model the behaviour of objects and event-driven applications



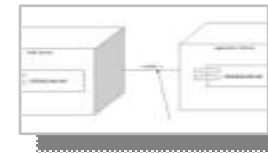
Design Principles

Create well-designed software that's easier to change and reuse



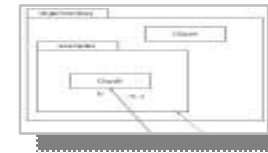
Object Diagrams & Filmstrips

Model snapshots of the running system and show how actions change object state



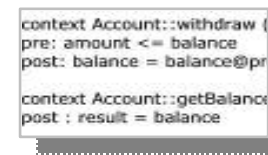
Implementation Diagrams

Model the physical components of a system and their deployment architecture



Packages & Model Management

Organise your logical and physical models with packages



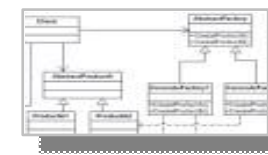
Object Constraint Language

Model business rules and create unambiguous specifications



User Experience Modeling

Design user-centred systems with UML



Design Patterns

Apply proven solutions to common OO design problems

www.parlezuml.com/training.htm

© Jason Gorman 2005. All rights reserved.

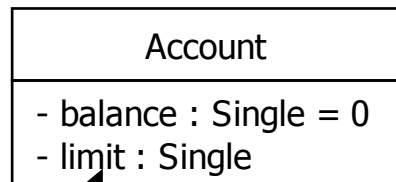


Classes

Account

```
class Account  
{  
}
```

Attributes



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

[visibility] [/] attribute_name[multiplicity] [: type [= default_value]]

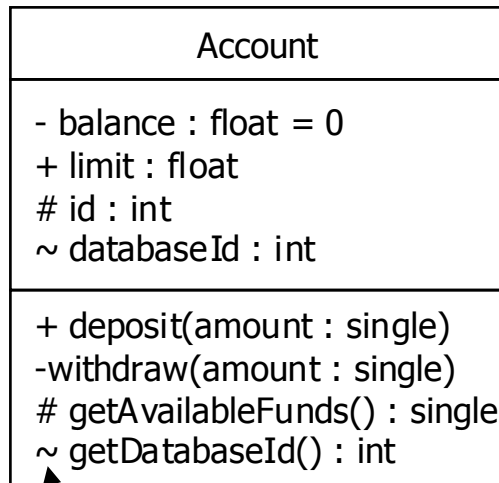
Operations

Account
- balance : Single = 0 - limit : Single
+ deposit(amount : Single) ← + withdraw(amount : Single)

[visibility] op_name([[in|out] parameter : type[, more params]])[: return_type]

```
class Account
{
    private float balance = 0;
    private float limit;
    public void deposit(float amount)
    {
        balance = balance + amount;
    }

    public void withdraw(float amount)
    {
        balance = balance - amount;
    }
}
```



+ = public
 - = private
 # = protected
 ~ = package

Visibility

class Account

```

{

    private float balance = 0;
    public float limit;
    protected int id;
    int databaseId;

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

    private void withdraw(float amount)
    {
        balance = balance - amount;
    }

    protected int getId()
    {
        return id;
    }

    int getDatabaseId()
    {
        return databaseId;
    }

}
  
```

```
int noOfPeople = Person.getNumberOfPeople();
Person p = Person.createPerson("Jason Gorman");
```

Person
- <u>numberOfPeople</u> : int - name : string
+ <u>createPerson</u> (name : string) : Person + <u>getName</u> () : string + <u>getNumberOfPeople</u> () : int - Person(name : string)

Class & Instance Scope

```
class Person
{
    private static int numberOfPeople = 0;
    private String name;

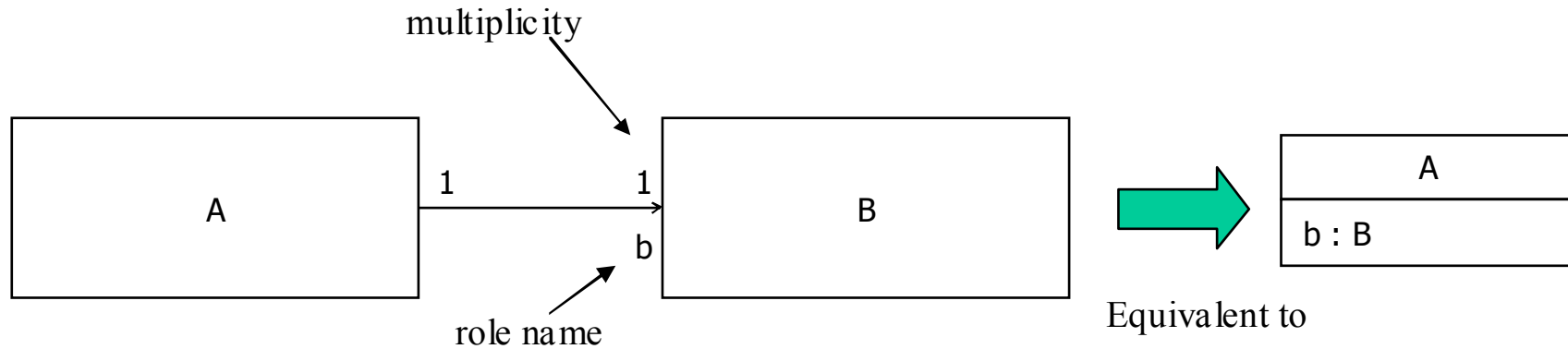
    private Person(string name)
    {
        this.name = name;
        numberOfPeople++;
    }

    public static Person createPerson(string name)
    {
        return new Person(name);
    }

    public string getName()
    {
        return this.name;
    }

    public static int getNumberOfPeople()
    {
        return numberOfPeople;
    }
}
```

Associations

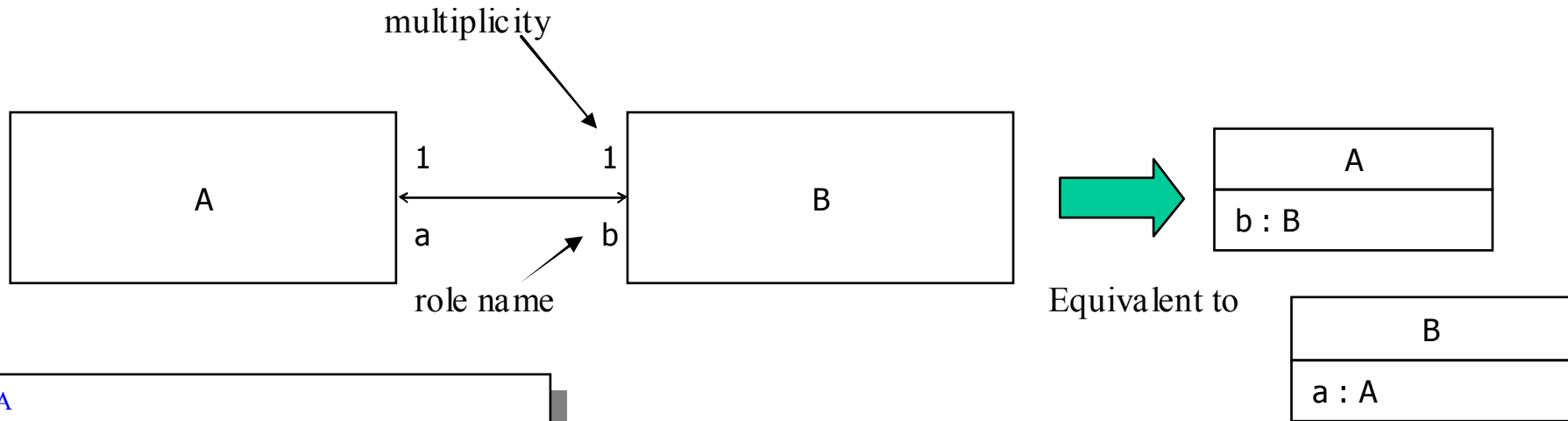


```
class A
{
    public B b = new B();
}

class B
{
}
```

```
A a = new A();
B b = a.b;
```

Bi-directional Associations



```
class A
{
    public B b;
    public A()
    {
        b = new B(this);
    }
}

class B
{
    public A a;
    public B(A a)
    {
        this.a = a;
    }
}
```

```
A a = new A();
B b = a.b;
A a1 = b.a;
assert a == a1;
```

Association names & role defaults

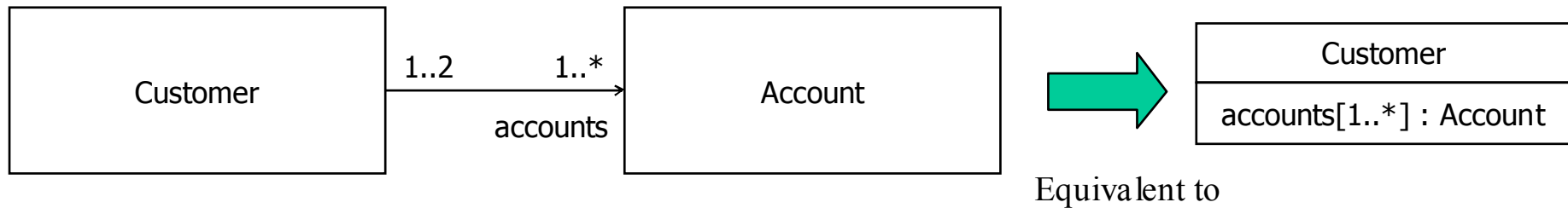


Default role name = address
Default multiplicity = 1

```
class Person
{
    // association: Lives at
    public Address address;

    public Person(Address address)
    {
        this.address = address;
    }
}
```

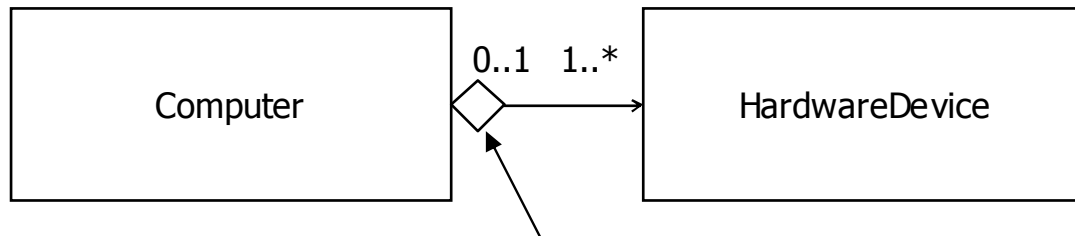
Multiplicity & Collections



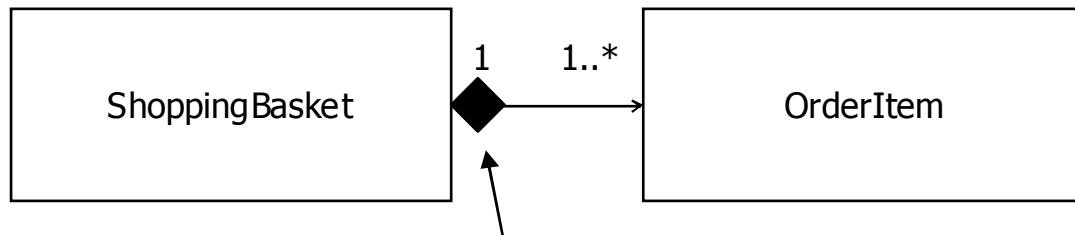
```
class Customer
{
    // accounts[1..*] : Account
    ArrayList accounts = new ArrayList();

    public Customer()
    {
        Account defaultAccount = new Account();
        accounts.add(defaultAccount);
    }
}
```

Aggregation & Composition

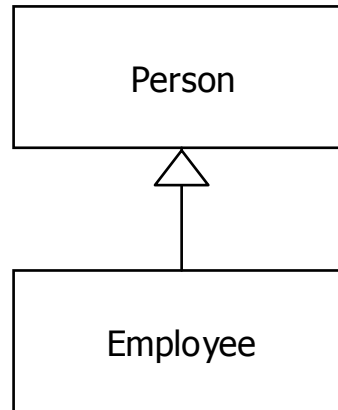


Aggregation – is made up of objects that can be shared or exchanged



Composition – is composed of objects that cannot be shared or exchanged and live only as long as the composite object

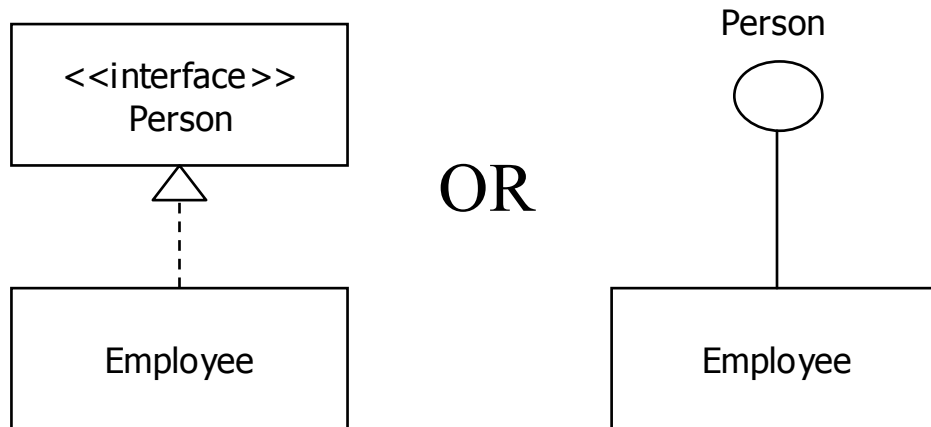
Generalization



```
class Person
{
}

class Employee extends Person
{
}
```

Realization



```
interface Person
{
}

class Employee implements Person
{
}
```

www.parlezuml.com