Queen Mary
University of London

ECS505U
SOFTWARE ENGINEERING

MUSTAFA BOZKURT & LORENZO JAMONE

LECTURER IN SOFTWARE ENGINEERING

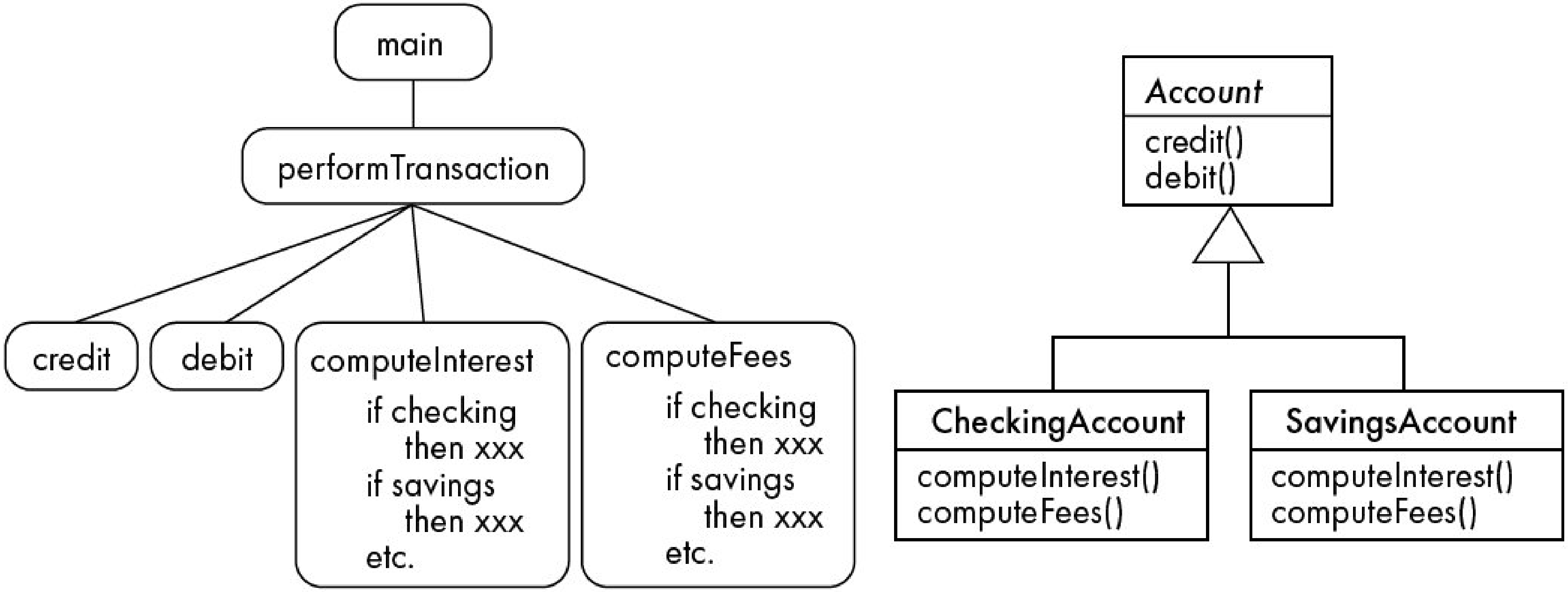
Week 3

Object-Oriented Modelling

LESSON OBJECTIVES

- Understand the concept of objects and classes applied to real world problems and programming
- Be able to draw ‘classes’
- Understand the rationale for OO design and UML
- Be aware of the basic types of UML diagrams

OO VS PROCEDURAL



OBJECT-ORIENTED METHODS

Based on identifying:

- Objects
- Classes
- Attributes
- Members
- Relationships between objects

OO technology applies to specification, design and programming

CONCEPTS AND PHENOMENA

Smartphone

https://static-secure.guim.co.uk/sys-images/Guardian/Pix/pictures/2013/12/2/1386002179310/Mobile-phone-bills-008.jpg

OBJECT-ORIENTED TERMINOLOGY

Smartphone

https://static-secure.guim.co.uk/sys-images/Guardian/Pix/pictures/2013/12/2/1386002179310/Mobile-phone-bills-008.jpg

WHAT IS AN OBJECT?

Objects represent entities, either physical, conceptual, or software

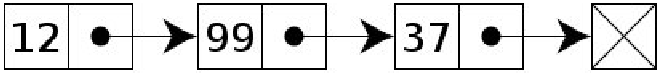
- Physical entity

http://media.caranddriver.com/images/media/51/dissected-lotus-based-infiniti-emerg-e-sports-car-concept-top-image-photo-451994-s-original.jpg

- Conceptual entity

http://www.financialgazette.co.zw/wp-content/uploads/pension-fund15.jpg

- Software entity



WHAT IS A CLASS?

http://edsource.org/wp-content/uploads/Fresno_class-size1.jpg

WHAT IS A CLASS?

http://www.toberight.com/wp-content/uploads/2010/10/middleclass1.jpg

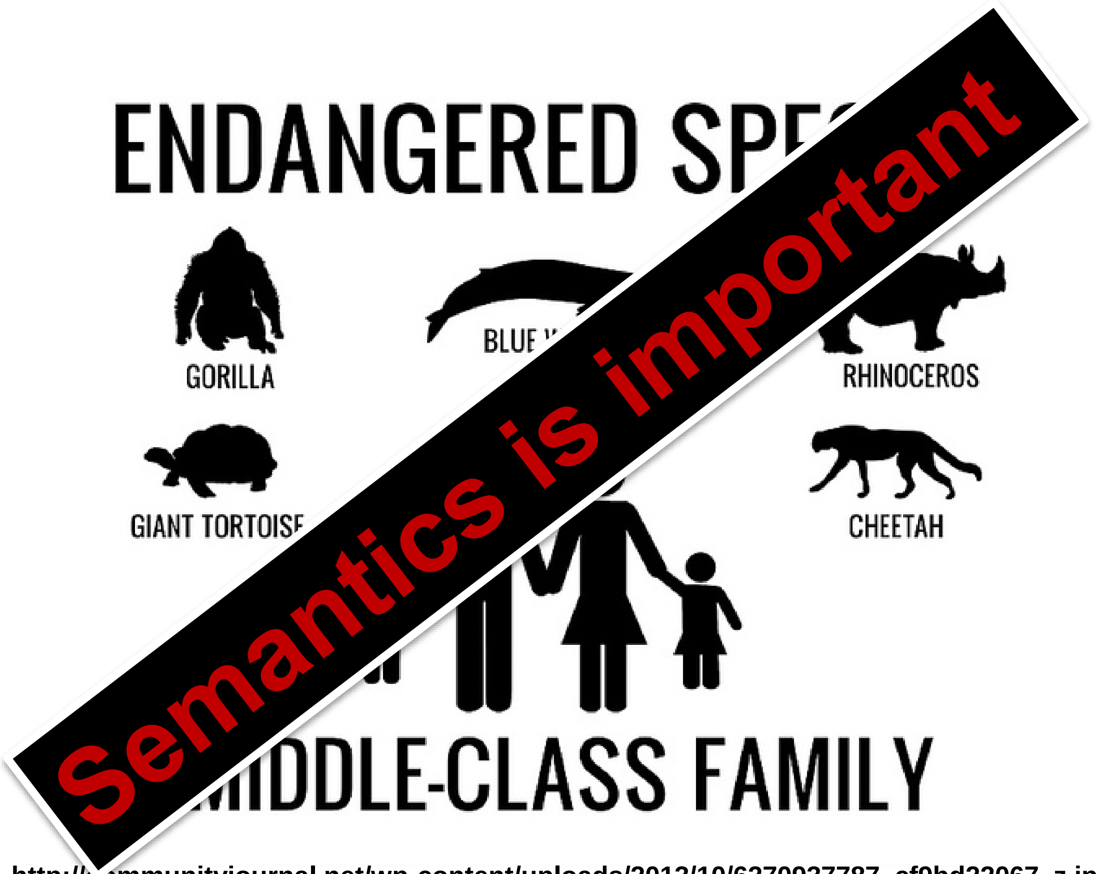
WHAT IS A CLASS?

http://static.guim.co.uk/sys-images/Guardian/Pix/pictures/2010/7/21/1279722913566/Middle-class-family-006.jpg

WHAT IS A CLASS?

http://www.eeo.com.cn/ens/uploadfiles/observations/20100915013008625.jpg

WHAT IS A CLASS?



Semantics is important

http://communityjournal.net/wp-content/uploads/2013/10/6270937787_cf9bd22067_z.jpg

WHAT IS A CLASS?

A class is a description of a group of entities with common

- Attributes (properties)
- Behavior (methods)
- Relationships
- Semantics

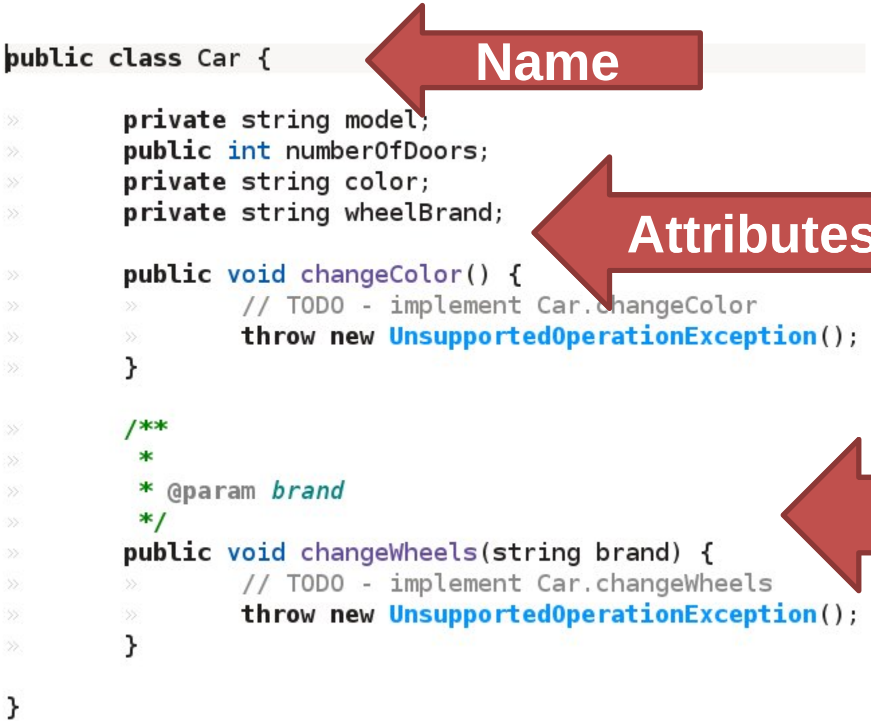
WHAT IS A CLASS?

A class is an abstraction in that it:

- Emphasizes relevant characteristics
- Suppresses other characteristics

OO Principle: Abstraction

WHAT IS A CLASS?



```
public class Car {

        private string model;
        public int numberOfDoors;
        private string color;
        private string wheelBrand;

        public void changeColor() {
                // TODO - implement Car.changeColor
                throw new UnsupportedOperationException();
        }

        /**
         *
         * @param brand
         */
        public void changeWheels(string brand) {
                // TODO - implement Car.changeWheels
                throw new UnsupportedOperationException();
        }

}
```

Name

Attributes

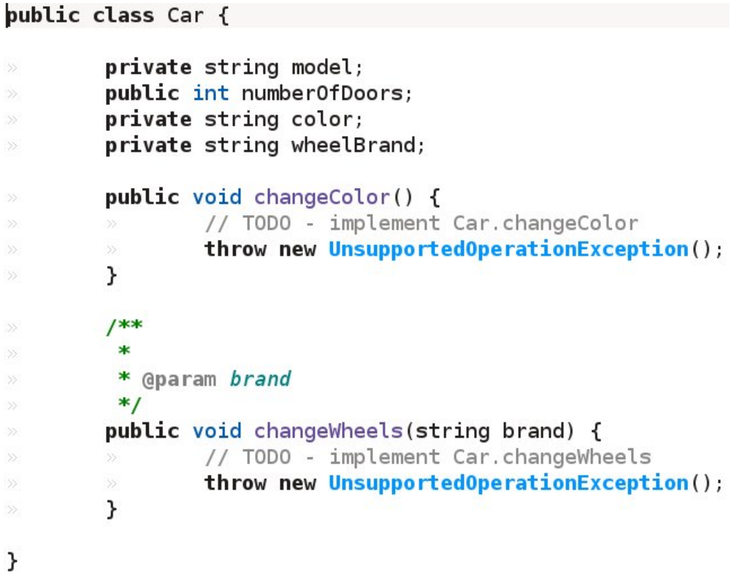
Methods

An abstraction in the context of object-oriented languages

Encapsulates both state (attributes) and behavior (methods)

WHAT DO WE USE TO MODEL A CONCEPT?

UML



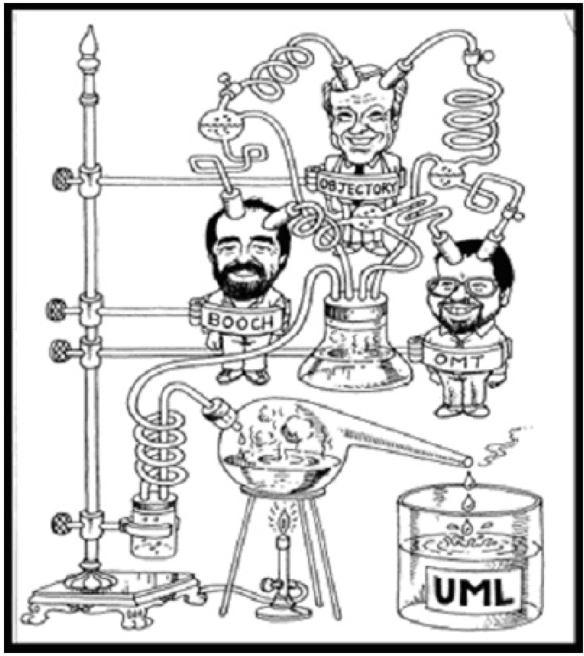
```
public class Car {

        private string model;
        public int numberOfDoors;
        private string color;
        private string wheelBrand;

        public void changeColor() {
                // TODO - implement Car.changeColor
                throw new UnsupportedOperationException();
        }

        /**
         *
         * @param brand
         */
        public void changeWheels(string brand) {
                // TODO - implement Car.changeWheels
                throw new UnsupportedOperationException();
        }

}
```

WHAT IS UML?

Unified Modeling Language

Convergence of three leading OO methods:

- OMT (James Rumbaugh)
- OOSE (Ivar Jacobson)
- Booch (Grady Booch)



http://www.ibm.com/developerworks/rational/library/998.html

Reference: “The Unified Modeling Language User Guide”, Addison Wesley, 1999**.**

Supported by several CASE tools (e.g Visual Paradigm and Rational Rose)

UML AND THIS COURSE

You can model 80% of most problems

by using about 30% UML

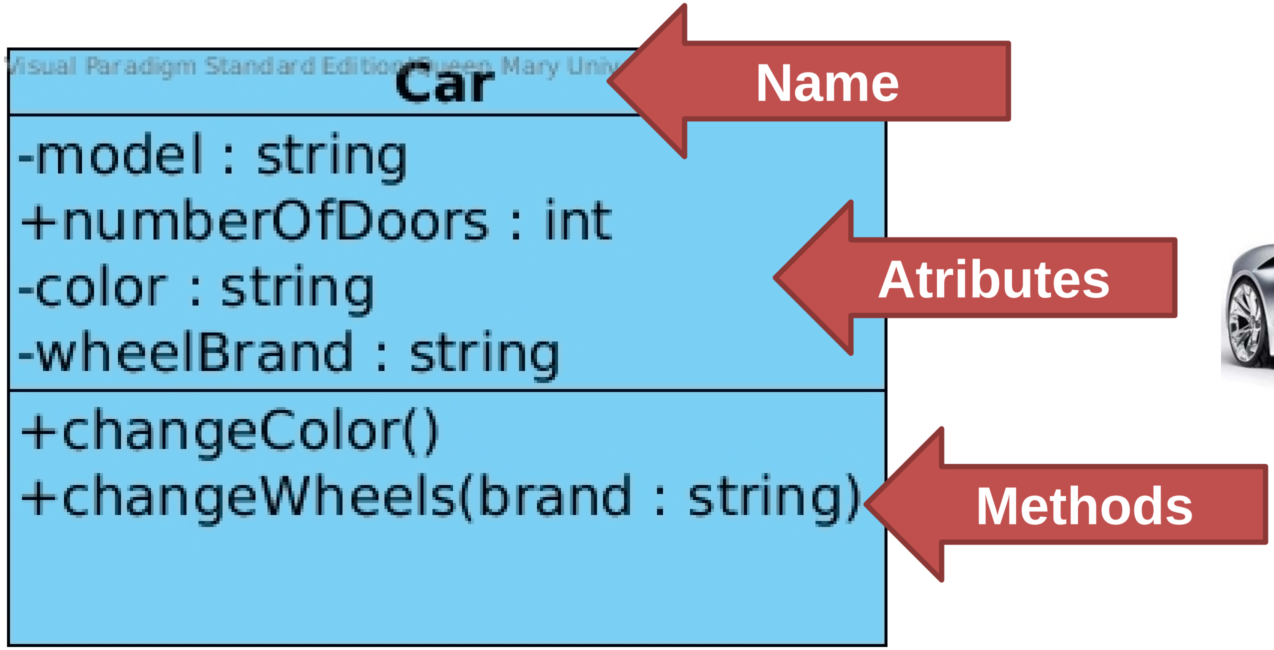
In this course, we teach you those 30%

WHY UML?

- The industry standard method for software engineering (design and documentation)
- When applied properly using the tool support it makes software engineering possible (‘round-trip engineering’)
- All design/documentation and implementation can really be integrated

WHAT IS A UML CLASS?

Design of a class



Car

-model : string
+numberOfDoors : int
-color : string
-wheelBrand : string

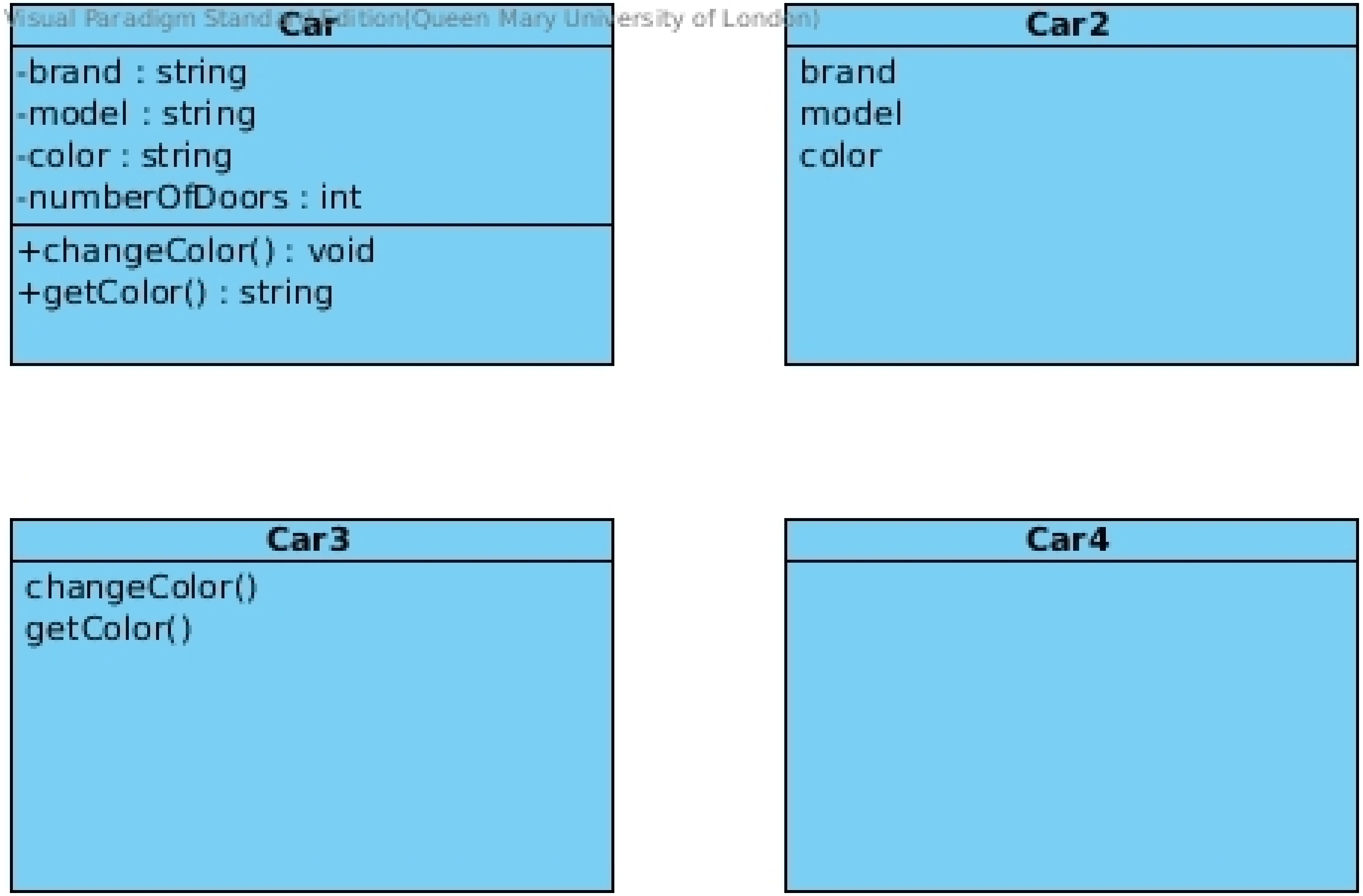
+changeColor()
+changeWheels(brand : string)

Name

Atributes

Methods

UML CLASS: DIFFERENT LEVELS OF DETAIL



Car

-brand : string
-model : string
-color : string
-numberOfDoors : int

+changeColor() : void
+getColor() : string

Car2

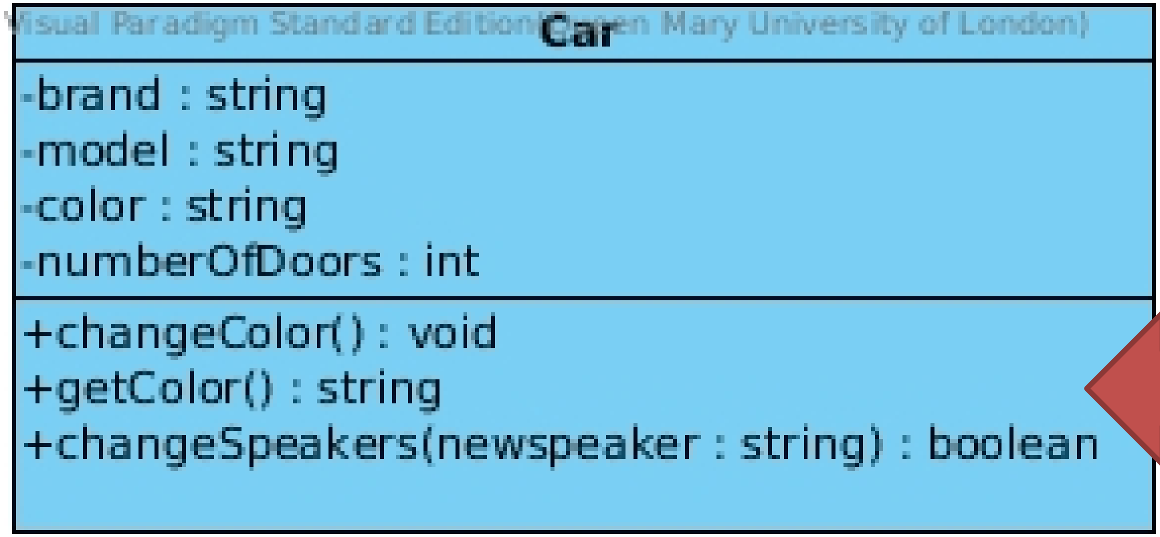
brand
model
color

Car3

changeColor()
getColor()

Car4

WHAT IS SIGNATURE ? (OF A METHOD)



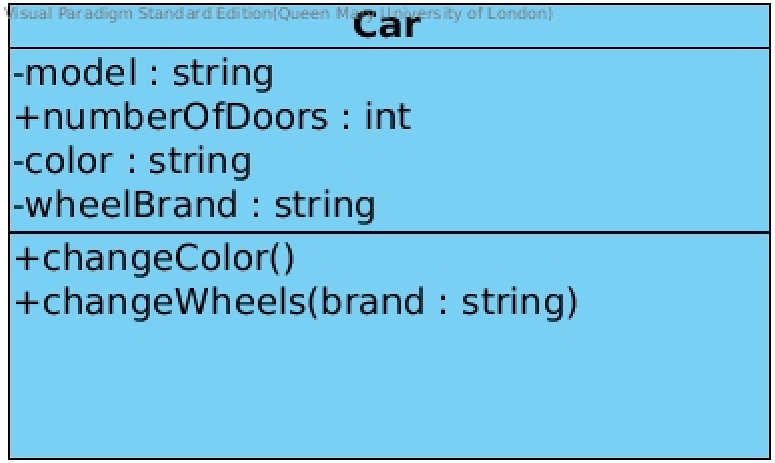
```
Car
-brand : string
-model : string
-color : string
-numberOfDoors : int
+changeColor() : void
+getColor() : string
+changeSpeakers(newspeaker : string) : boolean
```

Methods

UML specification lists an operation signature as follows:

visibility <<stereotype>> operation-name (parameter-list) : return-value

UML CLASS AND IMPLEMENTATION

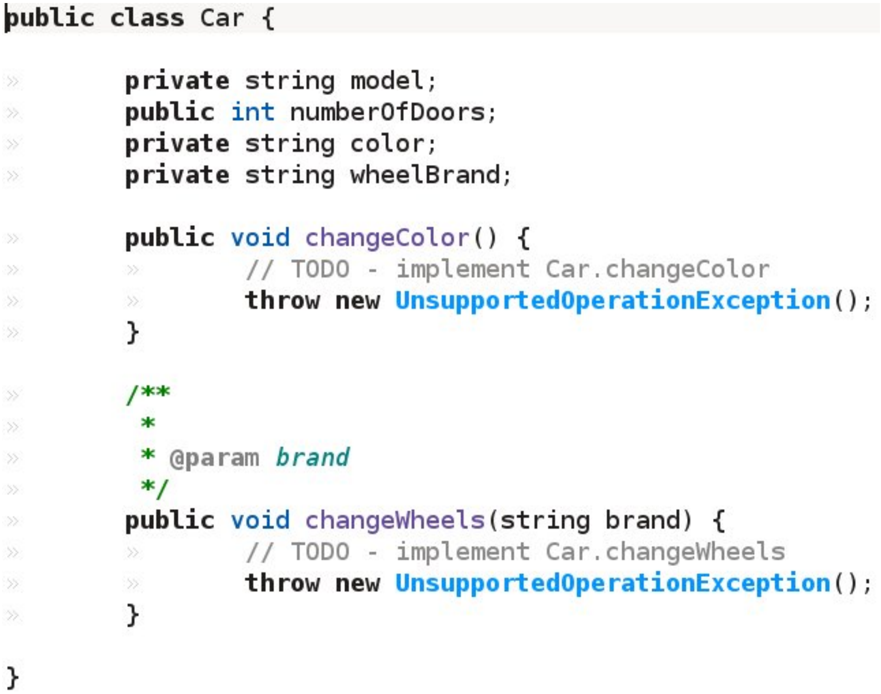


Car

-model : string
+numberOfDoors : int
-color : string
-wheelBrand : string

+changeColor()
+changeWheels(brand : string)

?



```
public class Car {

        private string model;
        public int numberOfDoors;
        private string color;
        private string wheelBrand;

        public void changeColor() {
                // TODO - implement Car.changeColor
                throw new UnsupportedOperationException();
        }

        /**
         *
         * @param brand
         */
        public void changeWheels(string brand) {
                // TODO - implement Car.changeWheels
                throw new UnsupportedOperationException();
        }

}
```

THE RELATIONSHIP BETWEEN CLASSES AND OBJECTS

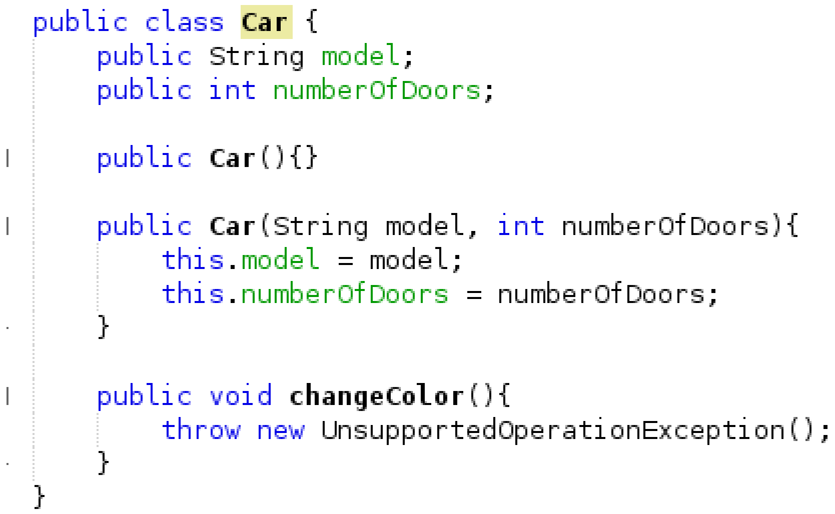
A class is an abstract definition of an object (programming view)

- It defines the structure and behavior of each object in the class
- It serves as a template for creating objects

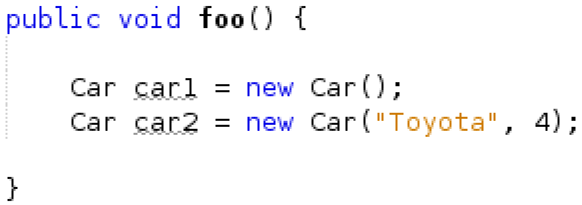
THE RELATIONSHIP BETWEEN CLASSES AND OBJECTS

An object is an instance of a class.

How do we create an instance of a class?



```java
public class Car {
    public String model;
    public int numberOfDoors;

    public Car(){}

    public Car(String model, int numberOfDoors){
        this.model = model;
        this.numberOfDoors = numberOfDoors;
    }

    public void changeColor(){
        throw new UnsupportedOperationException();
    }
}
```

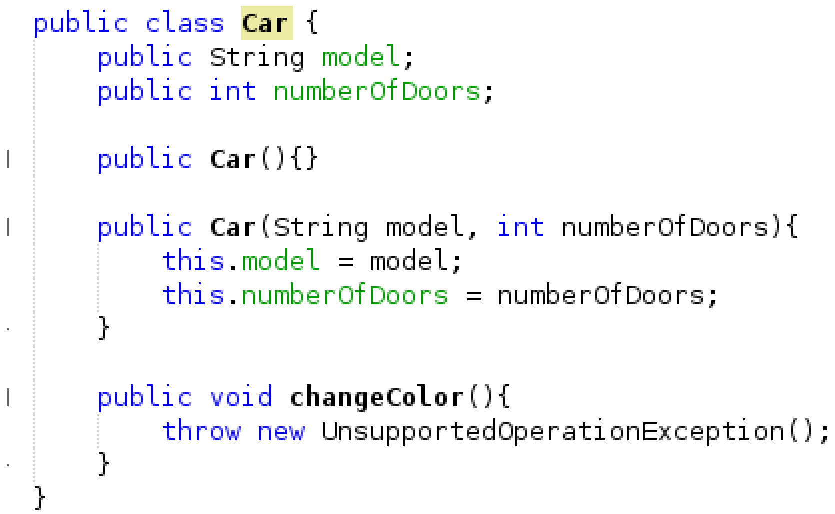


```java
public void foo() {

    Car car1 = new Car();
    Car car2 = new Car("Toyota", 4);

}
```

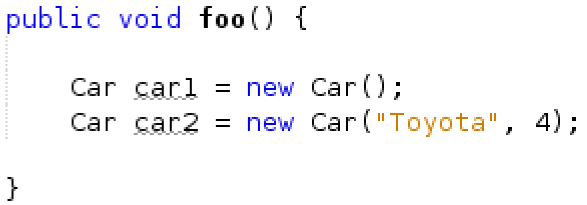
THE RELATIONSHIP BETWEEN CLASSES AND OBJECTS

An object is an instance of a class.

How do we create an instance of a class?



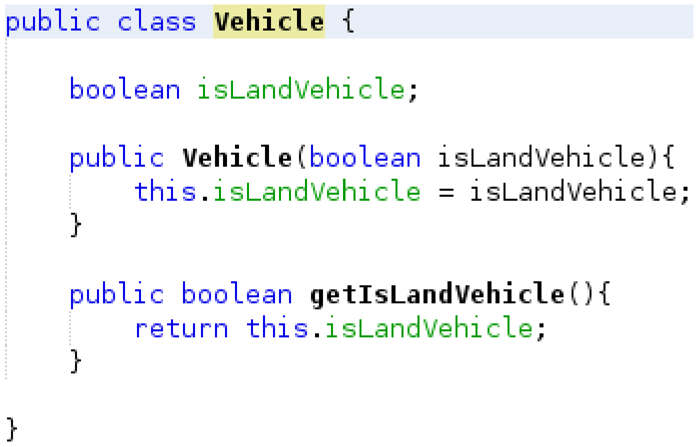
```
public class Car {
    public String model;
    public int numberOfDoors;

    public Car(){}

    public Car(String model, int numberOfDoors){
        this.model = model;
        this.numberOfDoors = numberOfDoors;
    }

    public void changeColor(){
        throw new UnsupportedOperationException();
    }
}
```



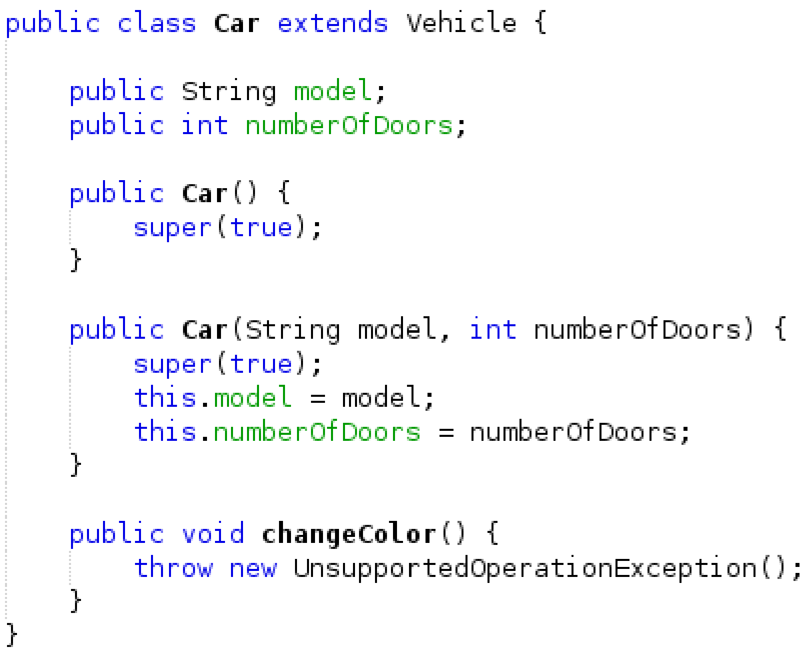
```
public void foo() {

    Car car1 = new Car();
    Car car2 = new Car("Toyota", 4);

}
```

THE RELATIONSHIP BETWEEN CLASSES AND OBJECTS

How do we create an instance of a class?

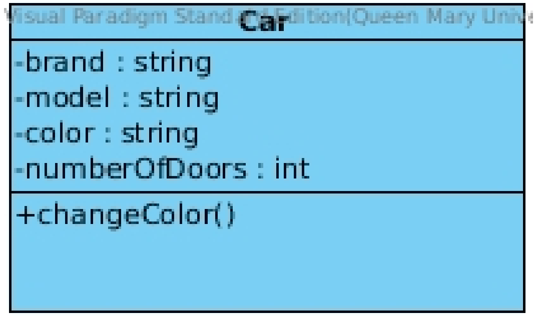


```java
public class Vehicle {

    boolean isLandVehicle;

    public Vehicle(boolean isLandVehicle){
        this.isLandVehicle = isLandVehicle;
    }

    public boolean getIsLandVehicle(){
        return this.isLandVehicle;
    }

}
```



```java
public class Car extends Vehicle {

    public String model;
    public int numberOfDoors;

    public Car() {
        super(true);
    }

    public Car(String model, int numberOfDoors) {
        super(true);
        this.model = model;
        this.numberOfDoors = numberOfDoors;
    }

    public void changeColor() {
        throw new UnsupportedOperationException();
    }
}
```

REPRESENTING OBJECTS



Car

-brand : string
-model : string
-color : string
-numberOfDoors : int

+changeColor()

Class

Car Mustafa's Car:

brand = “Ferrari”
model = “458 Spider”
color = “red”
numberOfDoors = 2

Class and Object Name

Adam's Car:

brand = “Ford”
model = “Festiva”
color = “blue”
numberOfDoors = 2

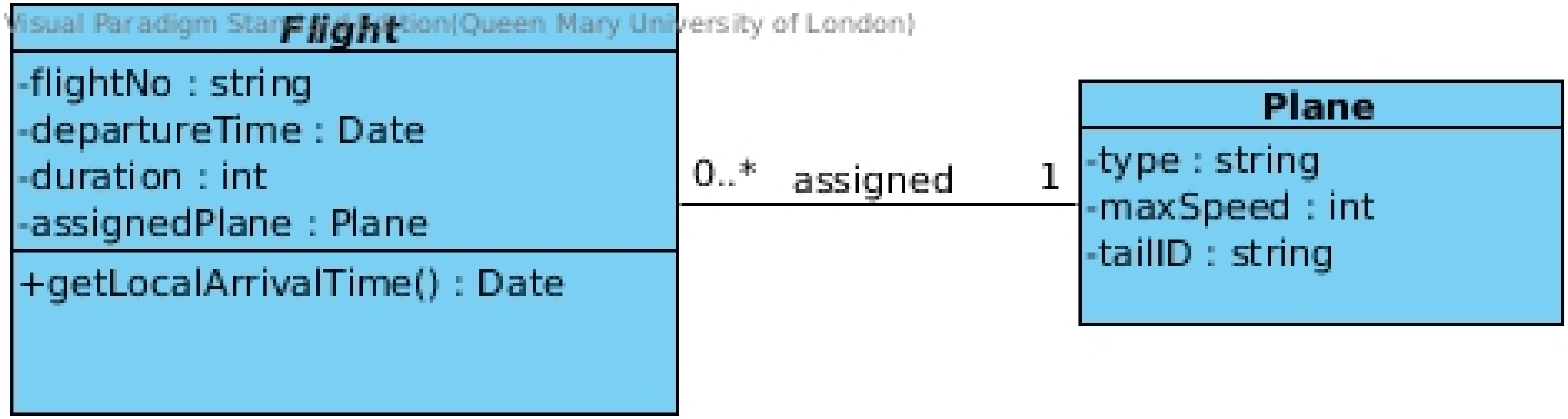
Object Name Only

auto.ferrari.com

cargurus.com

An object is represented as rectangles with underlined names

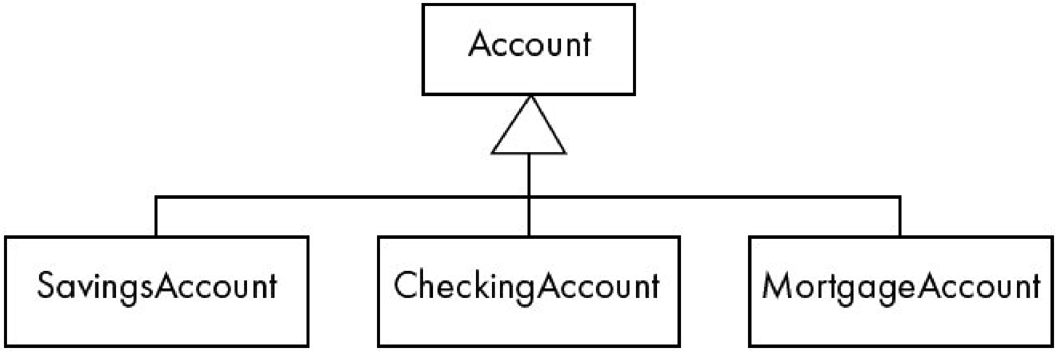
ATTRIBUTE VS ASSOCIATION



Two types of variables:

- Attributes
- Associations

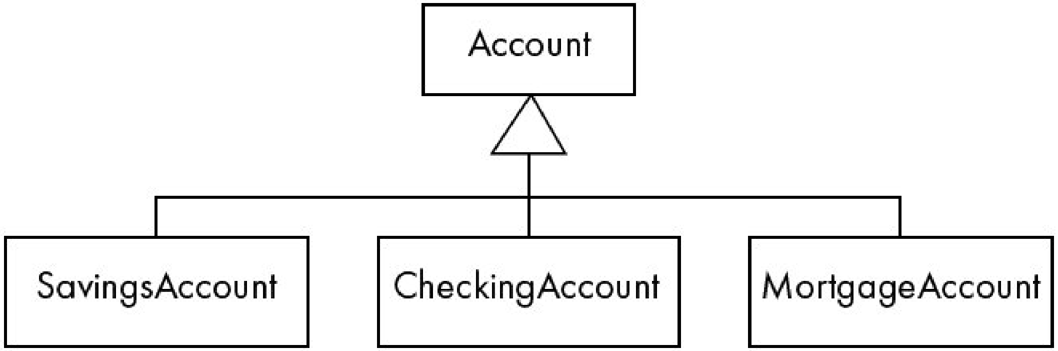
CLASS HIERARCHY



Two types of classes

- **Superclass** (common attributes, associations and operations)
- **Subclass** (specialised)

CLASS HIERARCHY



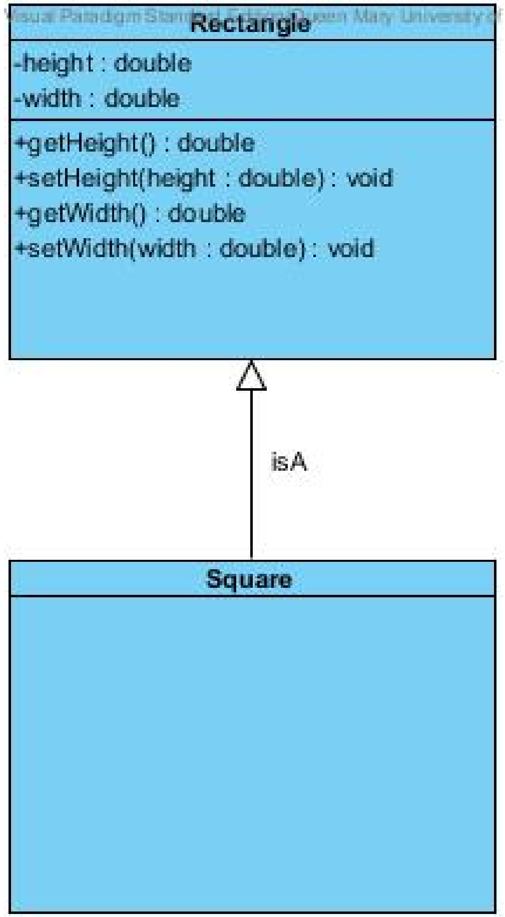
Two types of relations

- **Generalisation:** Relationship between subclass and immediate superclass
- **Specialisation** is the subclass

A hierarchy with one ore more generalisations called

inheritance hierarchy

CLASS HIERARCHY



Rectangle

-height : double
-width : double

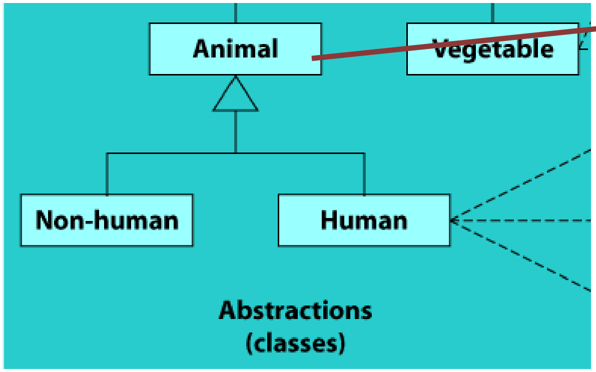
+getHeight() : double
+setHeight(height : double) : void
+getWidth() : double
+setWidth(width : double) : void

isA

Square

INHERITANCE

- Don’t the subclass or superclass ambiguous names (will lead to bad generalisations).
- A subclass must retain its distinctiveness throughout its life.
- All the inherited features must make sense in each subclass.



Run behaviour

INHERITANCE BENEFITS

Generalisations and their resulting inheritance help to

- Avoid duplication
- Improve reuse

Beware *poorly designed generalisations* can actually cause *more problems* than they solve.

HIERARCHY PRINCIPLES

1. **Open/Close principle** – Classes should be open for extensions and close for changes
2. **Liskov principle** – Subclasses should not require more, and not deliver less
3. **Dependency inversion principle** – Classes should only depend on abstractions

OPEN/CLOSE PRINCIPLE

According to Bertrand Meyer:

A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.

A module will be said to be closed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

OPEN/CLOSE PRINCIPLE

1. A class should be open for extension, but closed for changes
2. Achieved via inheritance and dynamic binding

OPEN/CLOSE PRINCIPLE

Suppose you were asked to create an application for a library which display book info to screen and print to paper for customers to read.

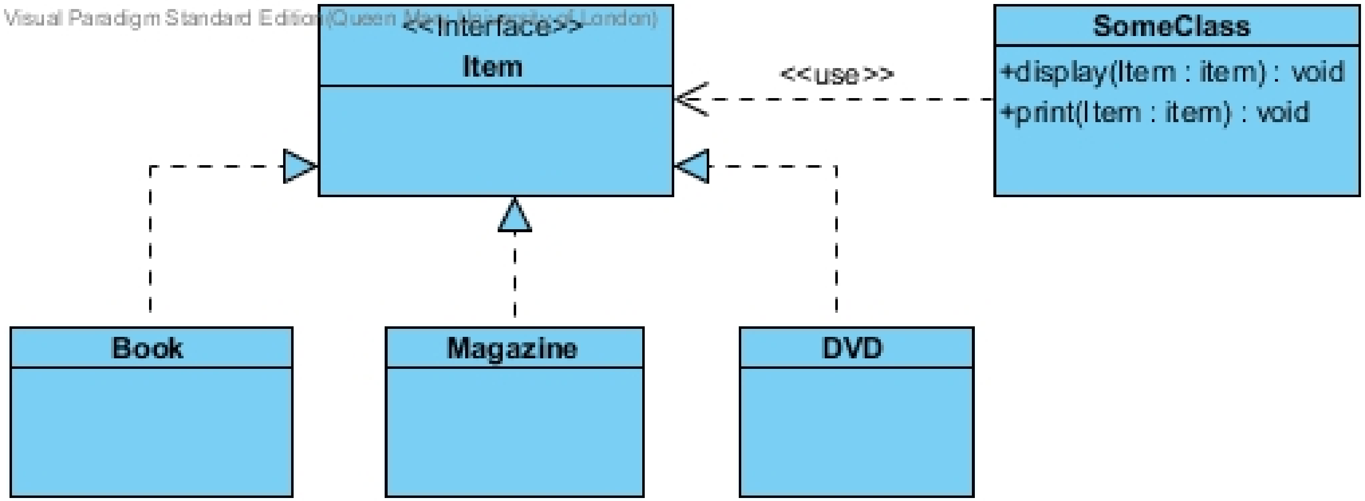
```
public class Book {
        // book details
}
```

```
public class SomeClass {
        public void display(Book book){
        //display book
        }
        public void print(Book book){
        //print book
        }
}
```

Your customer liked the code and everyone is happy!

OPEN/CLOSE PRINCIPLE

Few days later your customer said he wants the code to print other items in the library magazines and DVDs.

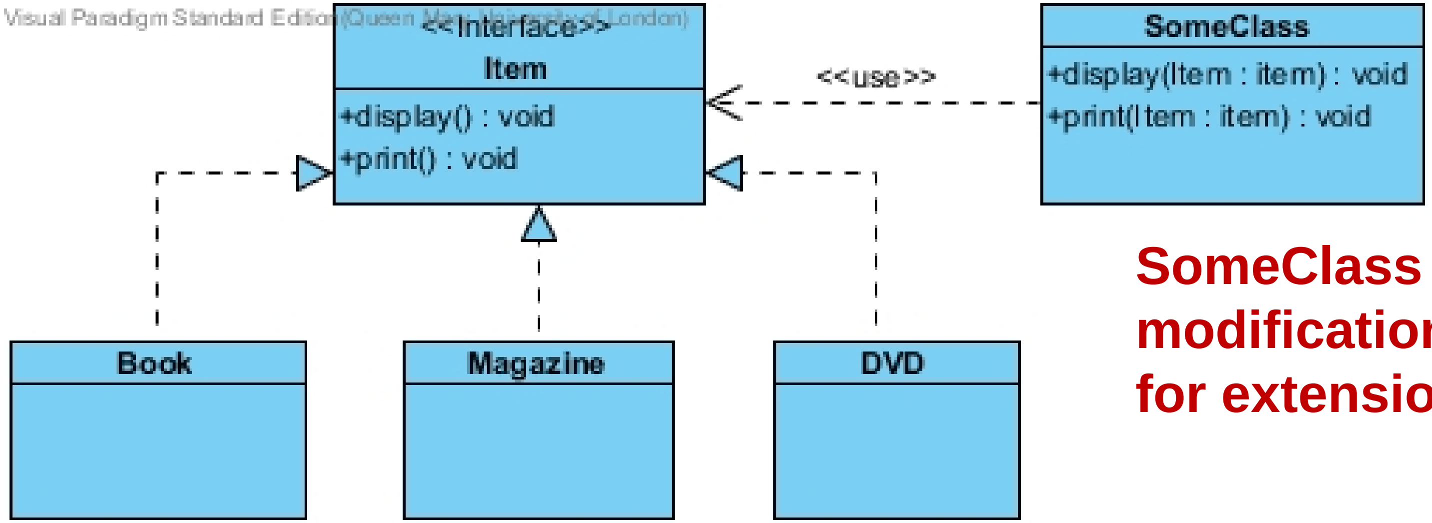

Visual Paradigm Standard Edition(Queen Mary University of London)
<<Interface>>
Item
<<use>>
SomeClass
+display(Item : item) : void
+print(Item : item) : void
Book
Magazine
DVD

OPEN/CLOSE PRINCIPLE

```
public class SomeClass {
            public void display(Item item){
             if (item instanceof Book){
             //display book
            }
            if (item instanceof Magazine){
             //display Magazine
            }
            if (item instanceof DVD){
             //display DVD
            }
}
```

It’s bad because every time you add a new item type you need to modify SomeClass

OPEN/CLOSE PRINCIPLE


Visual Paradigm Standard Edition(Queen Mary University of London)
<<Interface>>
Item
+display() : void
+print() : void
<<use>>
SomeClass
+display(Item : item) : void
+print(Item : item) : void
Book
Magazine
DVD

SomeClass is closed for modification but open for extension

```
public class SomeClass {
        public void display(Item item){
                item.display();
        }
        public void print(Item item){
                item.print();
        }
}
```

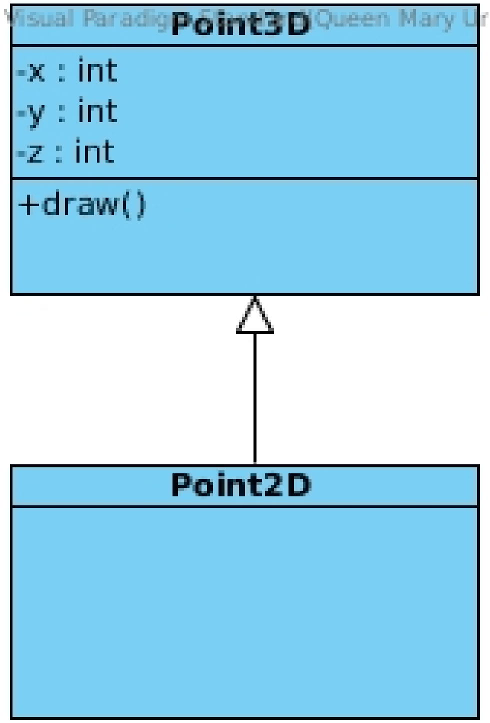
INHERITANCE: LISKOV PRINCIPLE

The Liskov Substitution Principle:

If you have a variable whose type is a superclass, then the program should work properly if you place an instance of that superclass or any of its subclasses in the variable.

The program using the variable should not be able to tell which class is being used, and should not care.

INHERITANCE: LISKOV PRINCIPLE

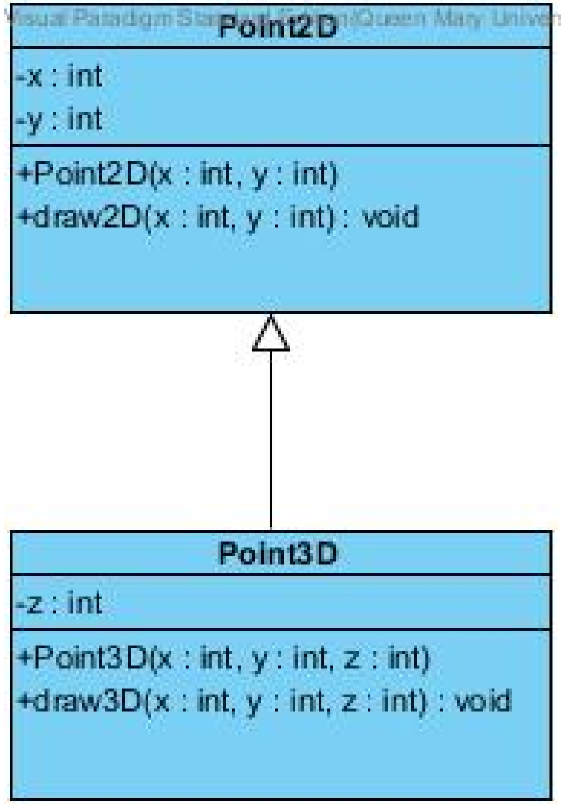


Point3D

-x : int
-y : int
-z : int

+draw()

Point2D



Point2D

-x : int
-y : int

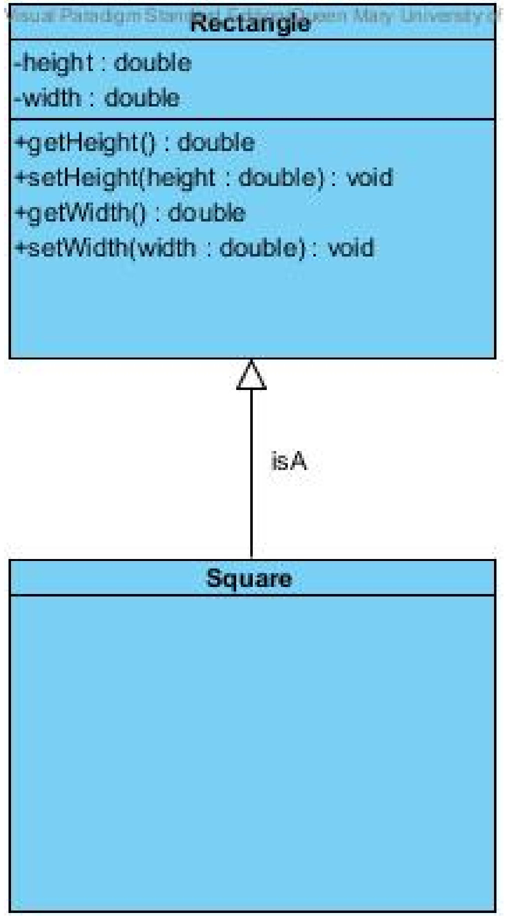
+Point2D(x : int, y : int)
+draw2D(x : int, y : int) : void

Point3D

-z : int

+Point3D(x : int, y : int, z : int)
+draw3D(x : int, y : int, z : int) : void

INHERITANCE: LISKOV PRINCIPLE



Rectangle

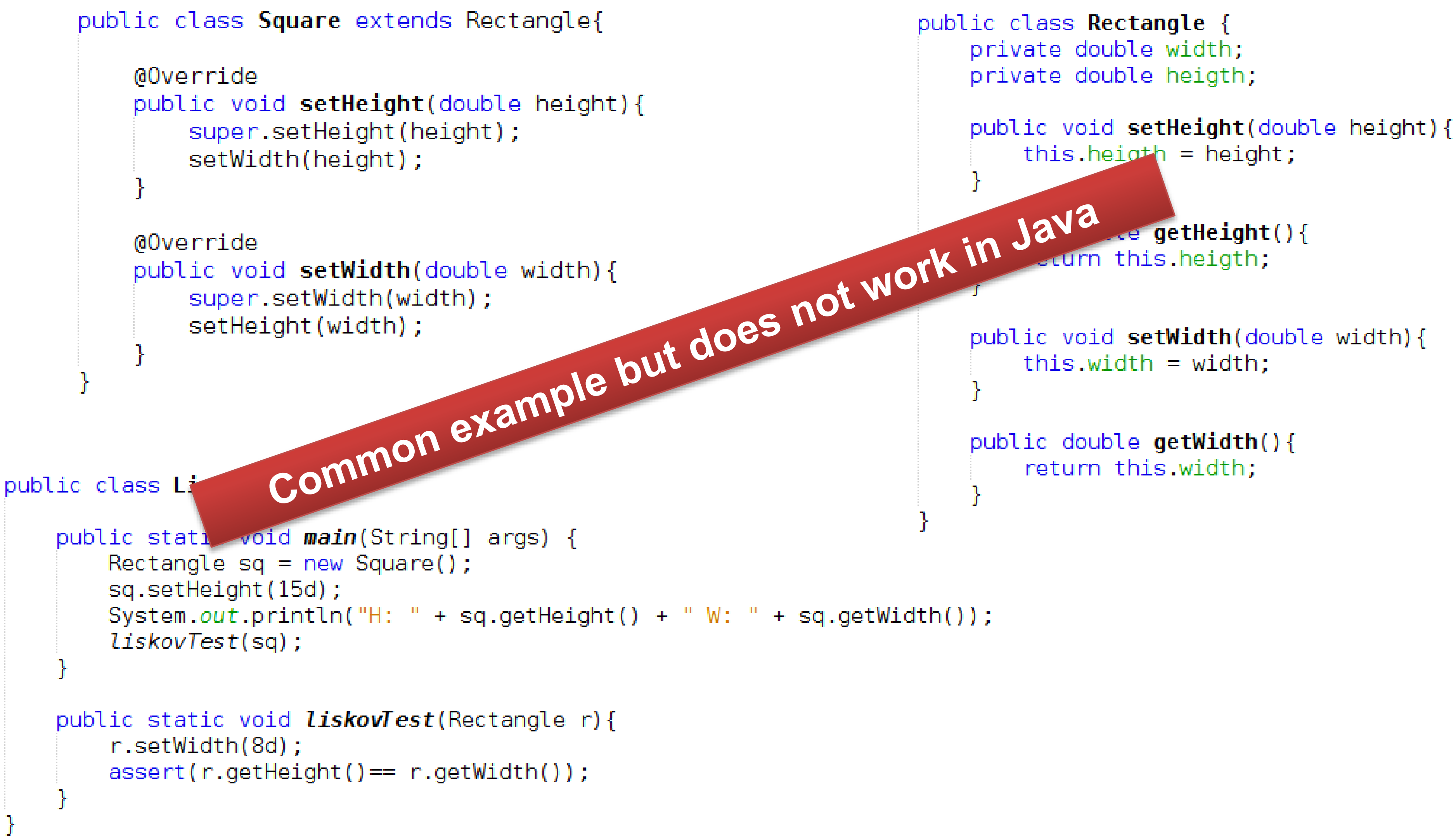
-height : double
-width : double

+getHeight() : double
+setHeight(height : double) : void
+getWidth() : double
+setWidth(width : double) : void

isA

Square

INHERITANCE: LISKOV PRINCIPLE



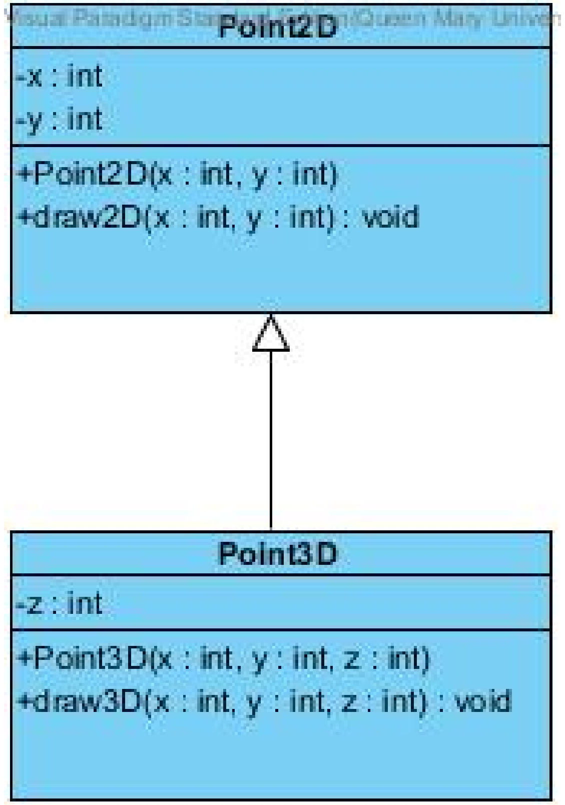
```
public class Square extends Rectangle{

    @Override
    public void setHeight(double height){
        super.setHeight(height);
        setWidth(height);
    }

    @Override
    public void setWidth(double width){
        super.setWidth(width);
        setHeight(width);
    }
}
```

```
public class Rectangle {
    private double width;
    private double heigth;

    public void setHeight(double height){
        this.heigth = height;
    }

            e getHeight(){
        turn this.heigth;

    public void setWidth(double width){
        this.width = width;
    }

    public double getWidth(){
        return this.width;
    }
}
```

```
public class Li

    public stati     void main(String[] args) {
        Rectangle sq = new Square();
        sq.setHeight(15d);
        System.out.println("H: " + sq.getHeight() + " W: " + sq.getWidth());
        liskovTest(sq);
    }

    public static void liskovTest(Rectangle r){
        r.setWidth(8d);
        assert(r.getHeight()== r.getWidth());
    }
}
```

Common example but does not work in Java

INHERITANCE: LISKOV PRINCIPLE



Point2D

-x : int
-y : int

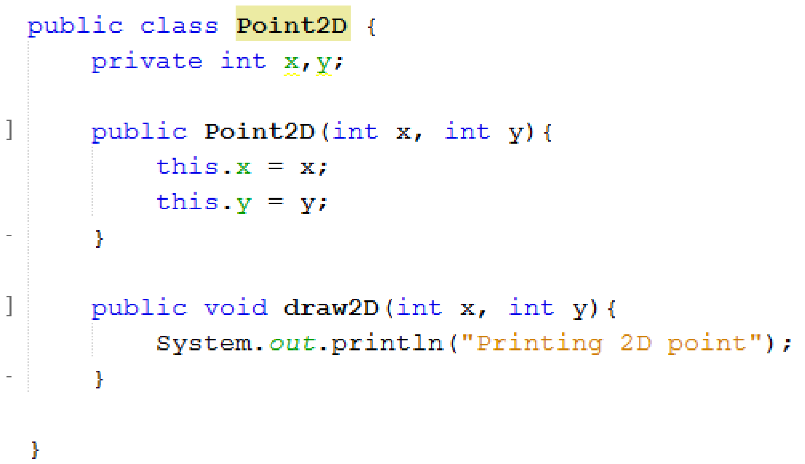
+Point2D(x : int, y : int)
+draw2D(x : int, y : int) : void

Point3D

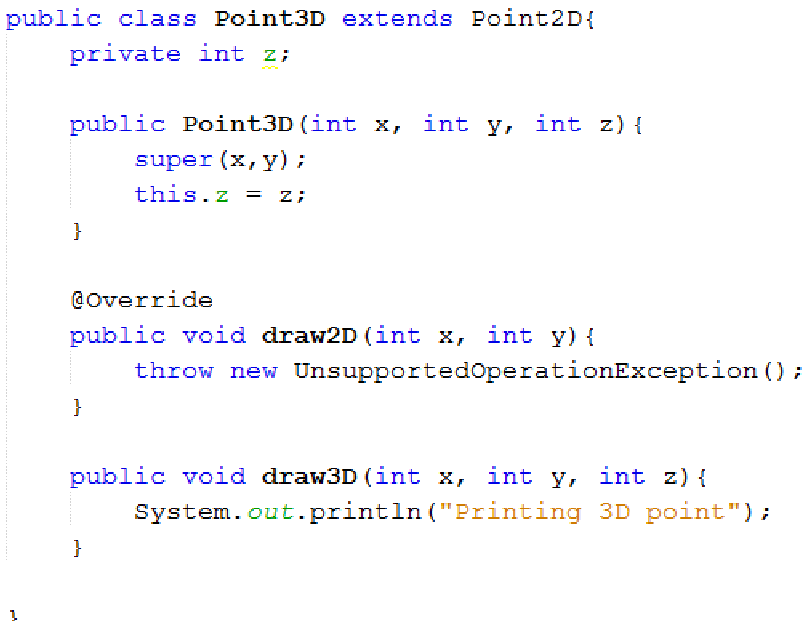
-z : int

+Point3D(x : int, y : int, z : int)
+draw3D(x : int, y : int, z : int) : void

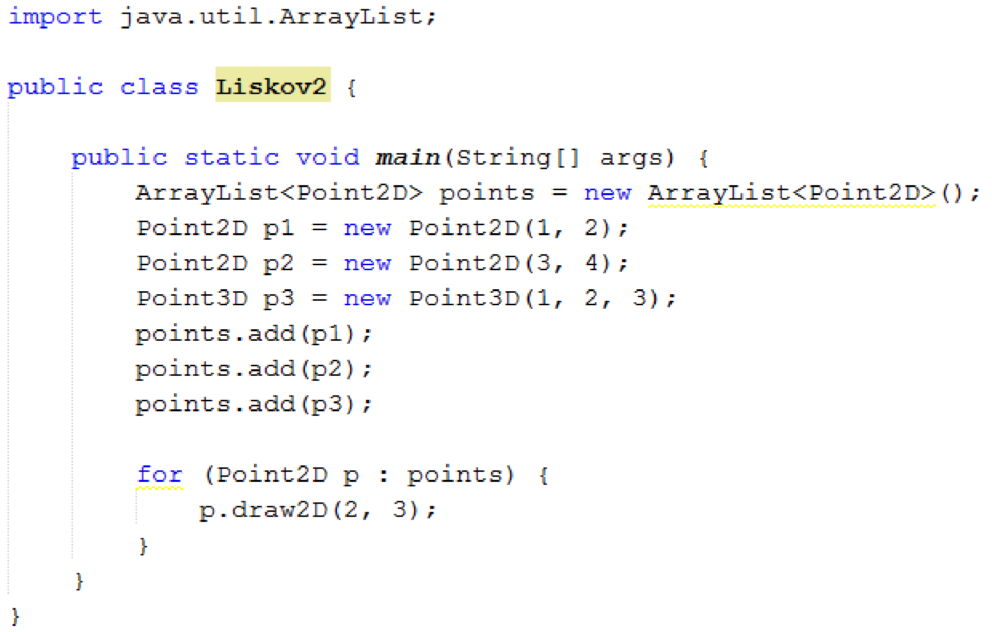
INHERITANCE: LISKOV PRINCIPLE



```java
public class Point2D {
    private int x,y;

    public Point2D(int x, int y){
        this.x = x;
        this.y = y;
    }

    public void draw2D(int x, int y){
        System.out.println("Printing 2D point");
    }

}
```

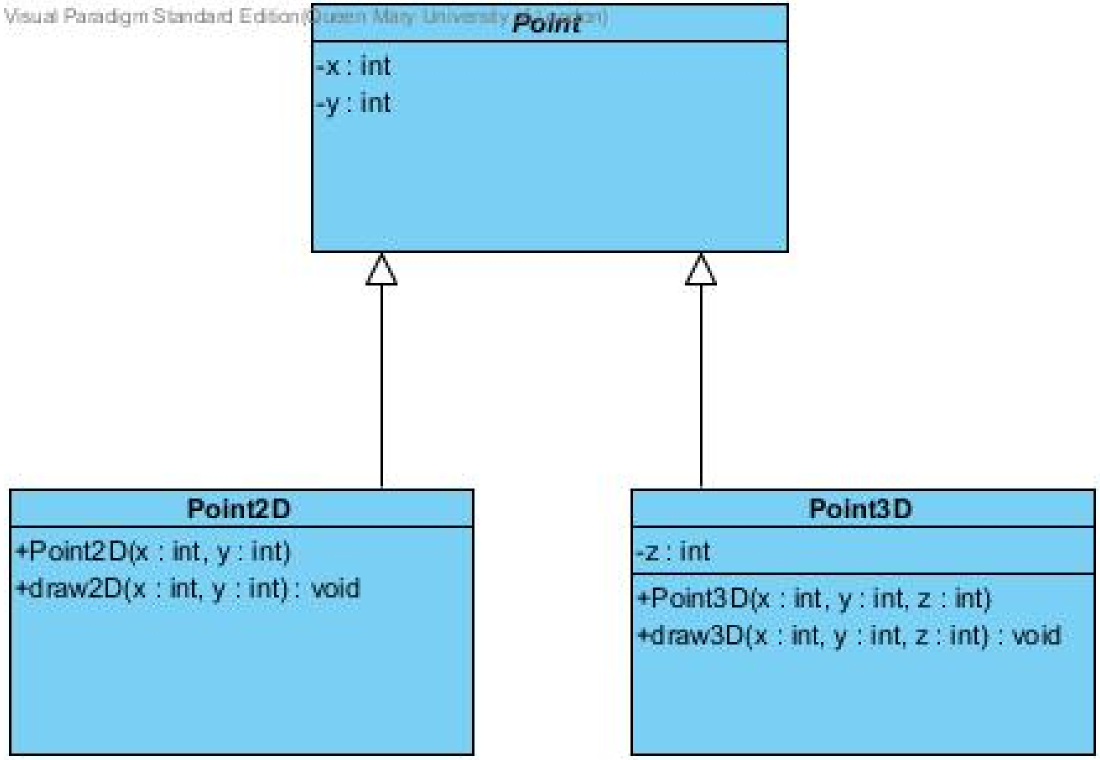


```java
public class Point3D extends Point2D{
    private int z;

    public Point3D(int x, int y, int z){
        super(x,y);
        this.z = z;
    }

    @Override
    public void draw2D(int x, int y){
        throw new UnsupportedOperationException();
    }

    public void draw3D(int x, int y, int z){
        System.out.println("Printing 3D point");
    }
}
```



```java
import java.util.ArrayList;

public class Liskov2 {

    public static void main(String[] args) {
        ArrayList<Point2D> points = new ArrayList<Point2D>();
        Point2D p1 = new Point2D(1, 2);
        Point2D p2 = new Point2D(3, 4);
        Point3D p3 = new Point3D(1, 2, 3);
        points.add(p1);
        points.add(p2);
        points.add(p3);

        for (Point2D p : points) {
            p.draw2D(2, 3);
        }
    }
}
```

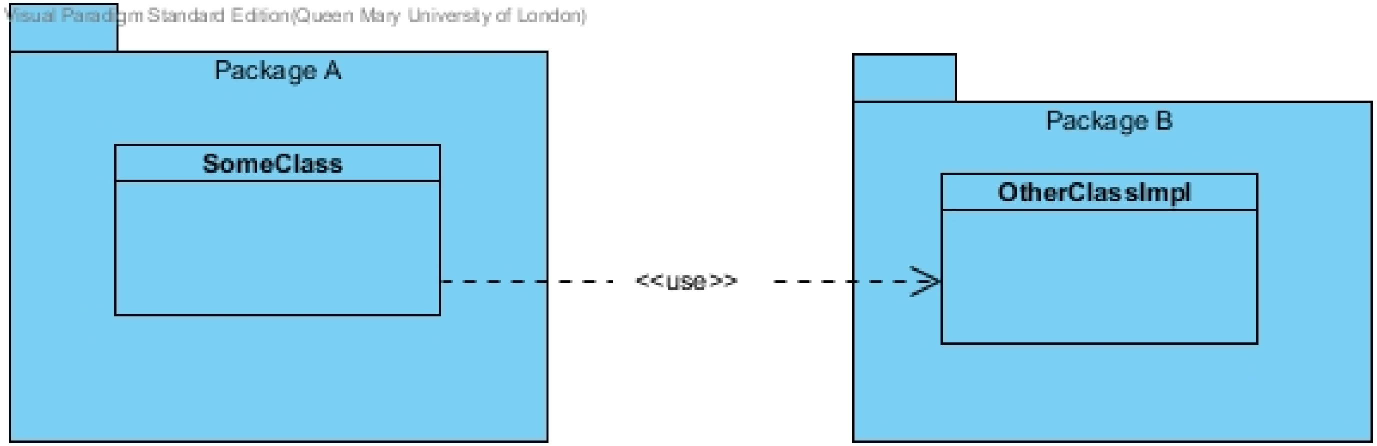
INHERITANCE: LISKOV PRINCIPLE


Visual Paradigm Standard Edition(Queen Mary University of London)
Point
-x : int
-y : int
Point2D
+Point2D(x : int, y : int)
+draw2D(x : int, y : int) : void
Point3D
-z : int
+Point3D(x : int, y : int, z : int)
+draw3D(x : int, y : int, z : int) : void

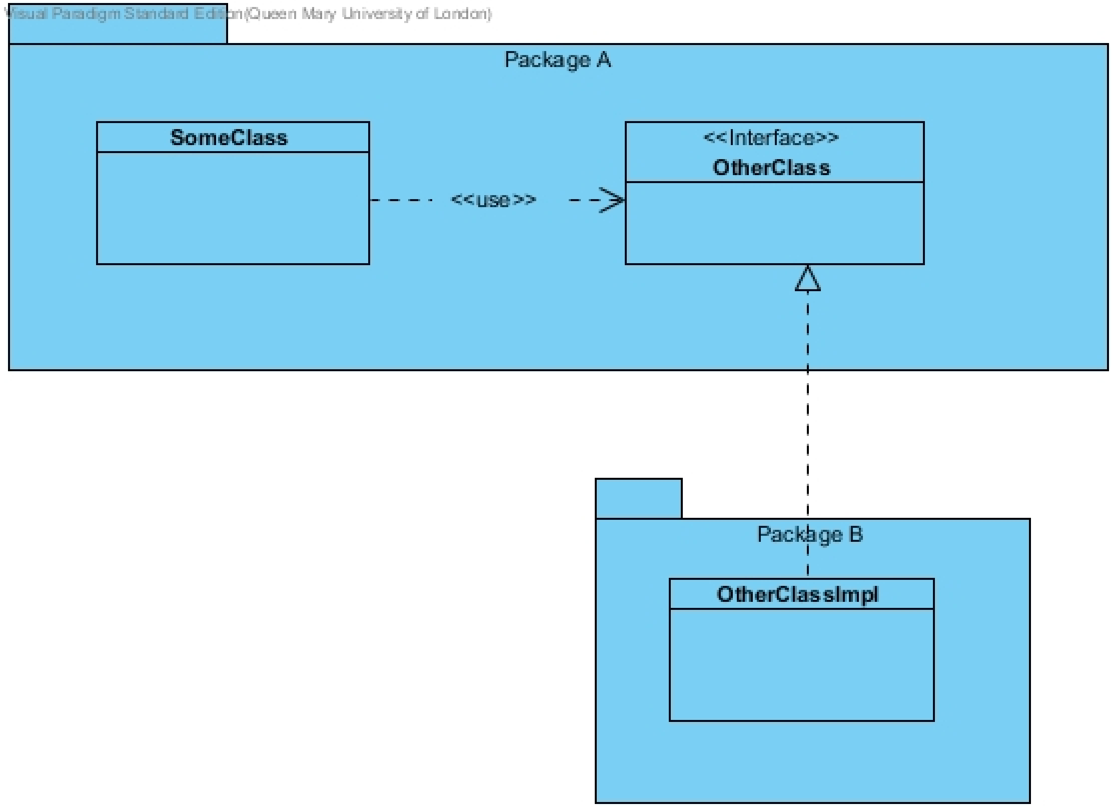
DEPENDENCY INVERSION PRINCIPLE

1. High-level modules should not depend on low-level modules. Both should depend on abstractions. – never on concrete subclasses
2. Abstractions should not depend on details. Details should depend on abstractions.

DEPENDENCY INVERSION PRINCIPLE


Visual Paradigm Standard Edition(Queen Mary University of London)
Package A
SomeClass
<<use>>
Package B
OtherClassImpl

DEPENDENCY INVERSION PRINCIPLE

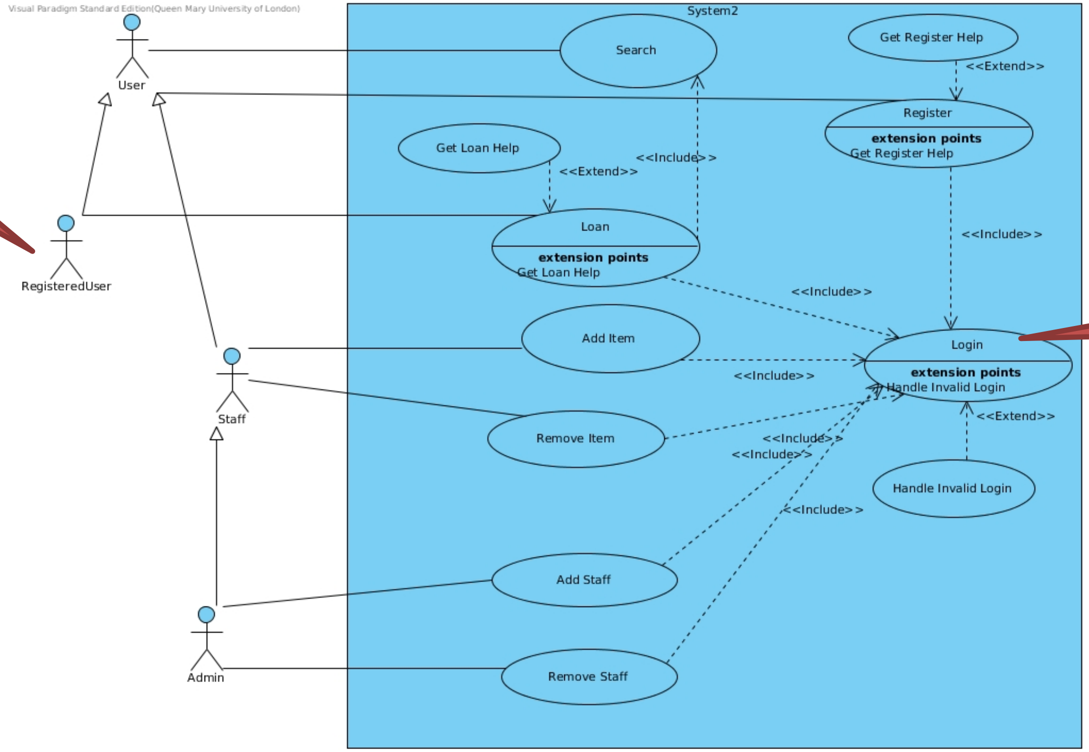


This principle can be used to break dependencies

UML: THE BASIC DIAGRAMS

- Class diagrams
- Use case diagrams
- Sequence diagrams
- Statechart diagrams

UML: USE CASE DIAGRAM

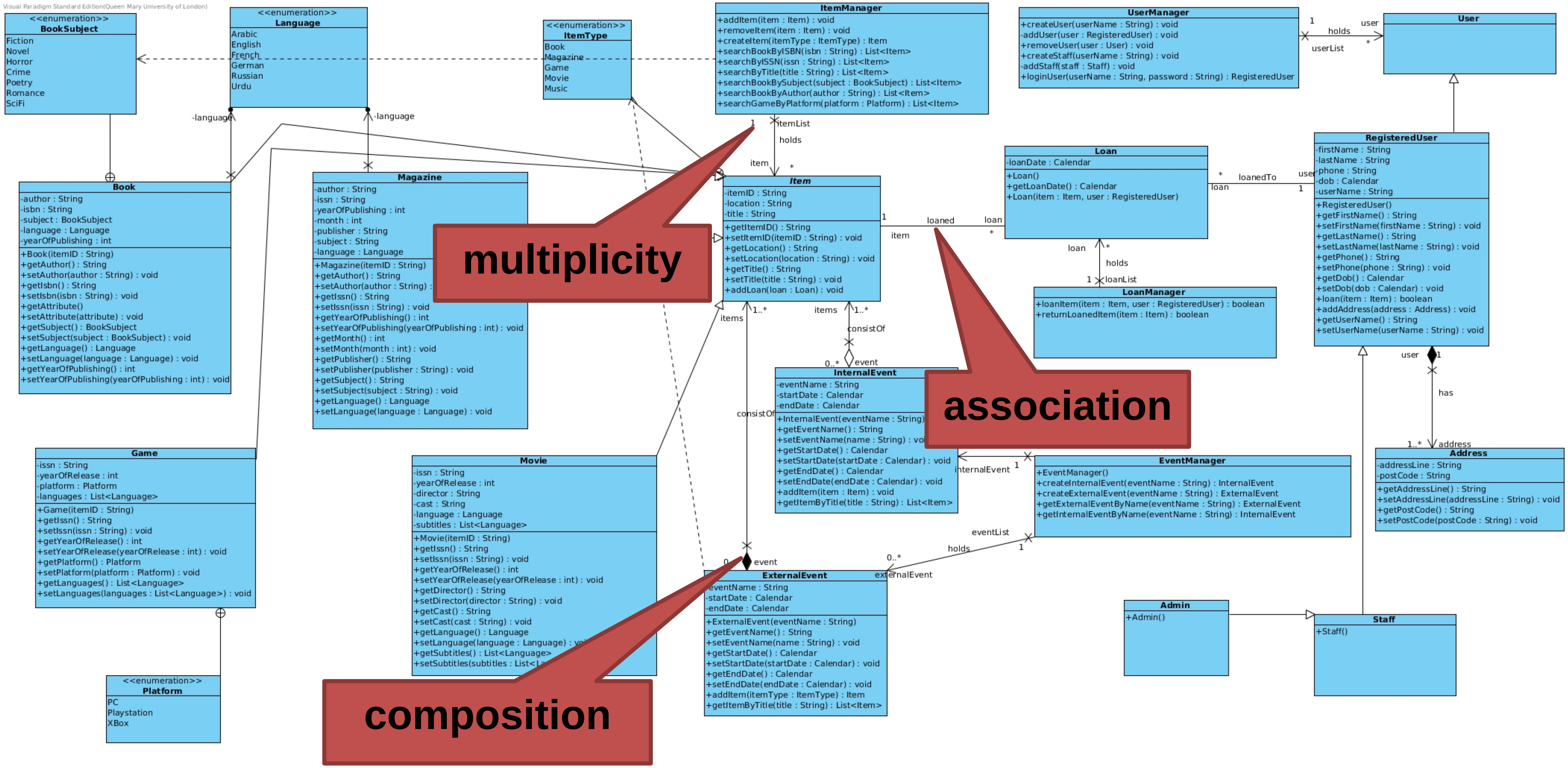


actor

use case

Use case diagrams represent the functionality of the system from user's point of view

UML: CLASS DIAGRAM



```
<<enumeration>>
BookSubject
Fiction
Novel
Horror
Crime
Poetry
Romance
SciFi

<<enumeration>>
Language
Arabic
English
French
German
Russian
Urdu

<<enumeration>>
ItemType
Book
Magazine
Game
Movie
Music

ItemManager
+addItem(item : Item) : void
+removeItem(item : Item) : void
+createItem(itemType : ItemType) : Item
+searchBookByISBN(isbn : String) : List<Item>
+searchByISSN(issn : String) : List<Item>
+searchByTitle(title : String) : List<Item>
+searchBookBySubject(subject : BookSubject) : List<Item>
+searchBookByAuthor(author : String) : List<Item>
+searchGameByPlatform(platform : Platform) : List<Item>

UserManager
+createUser(userName : String) : void
-addUser(user : RegisteredUser) : void
+removeUser(user : User) : void
+createStaff(userName : String) : void
-addStaff(staff : Staff) : void
+loginUser(userName : String, password : String) : RegisteredUser

User

Book
-author : String
-isbn : String
-subject : BookSubject
-language : Language
-yearOfPublishing : int
+Book(itemID : String)
+getAuthor() : String
+setAuthor(author : String) : void
+getIsbn() : String
+setIsbn(isbn : String) : void
+getAttribute()
+setAttribute(attribute) : void
+getSubject() : BookSubject
+setSubject(subject : BookSubject) : void
+getLanguage() : Language
+setLanguage(language : Language) : void
+getYearOfPublishing() : int
+setYearOfPublishing(yearOfPublishing : int) : void

Magazine
-author : String
-issn : String
-yearOfPublishing : int
-month : int
-publisher : String
-subject : String
-language : Language
+Magazine(itemID : String)
+getAuthor() : String
+setAuthor(author : String) :
+getIssn() : String
+setIssn(issn : String) : void
+getYearOfPublishing() : int
+setYearOfPublishing(yearOfPublishing : int) : void
+getMonth() : int
+setMonth(month : int) : void
+getPublisher() : String
+setPublisher(publisher : String) : void
+getSubject() : String
+setSubject(subject : String) : void
+getLanguage() : Language
+setLanguage(language : Language) : void

Item
-itemID : String
-location : String
-title : String
+getItemID() : String
+setItemID(itemID : String) : void
+getLocation() : String
+setLocation(location : String) : void
+getTitle() : String
+setTitle(title : String) : void
+addLoan(loan : Loan) : void

Loan
-loanDate : Calendar
+Loan()
+getLoanDate() : Calendar
+Loan(item : Item, user : RegisteredUser)

LoanManager
+loanItem(item : Item, user : RegisteredUser) : boolean
+returnLoanedItem(item : Item) : boolean

RegisteredUser
-firstName : String
-lastName : String
-phone : String
-dob : Calendar
-userName : String
+RegisteredUser()
+getFirstName() : String
+setFirstName(firstName : String) : void
+getLastName() : String
+setLastName(lastName : String) : void
+getPhone() : String
+setPhone(phone : String) : void
+getDob() : Calendar
+setDob(dob : Calendar) : void
+loan(item : Item) : boolean
+addAddress(address : Address) : void
+getUserName() : String
+setUserName(userName : String) : void

InternalEvent
-eventName : String
-startDate : Calendar
-endDate : Calendar
+InternalEvent(eventName : String)
+getEventName() : String
+setEventName(name : String) : voi
+getStartDate() : Calendar
+setStartDate(startDate : Calendar) : void
+getEndDate() : Calendar
+setEndDate(endDate : Calendar) : void
+addItem(item : Item) : void
+getItemByTitle(title : String) : List<Item>

Game
-issn : String
-yearOfRelease : int
-platform : Platform
-languages : List<Language>
+Game(itemID : String)
+getIssn() : String
+setIssn(issn : String) : void
+getYearOfRelease() : int
+setYearOfRelease(yearOfRelease : int) : void
+getPlatform() : Platform
+setPlatform(platform : Platform) : void
+getLanguages() : List<Language>
+setLanguages(languages : List<Language>) : void

Movie
-issn : String
-yearOfRelease : int
-director : String
-cast : String
-language : Language
-subtitles : List<Language>
+Movie(itemID : String)
+getIssn() : String
+setIssn(issn : String) : void
+getYearOfRelease() : int
+setYearOfRelease(yearOfRelease : int) : void
+getDirector() : String
+setDirector(director : String) : void
+getCast() : String
+setCast(cast : String) : void
+getLanguage() : Language
+setLanguage(language : Language) : vo
+getSubtitles() : List<Language>
+setSubtitles(subtitles : List<La

EventManager
+EventManager()
+createInternalEvent(eventName : String) : InternalEvent
+createExternalEvent(eventName : String) : ExternalEvent
+getExternalEventByName(eventName : String) : ExternalEvent
+getInternalEventByName(eventName : String) : InternalEvent

Address
-addressLine : String
-postCode : String
+getAddressLine() : String
+setAddressLine(addressLine : String) : void
+getPostCode() : String
+setPostCode(postCode : String) : void

ExternalEvent
-eventName : String
-startDate : Calendar
-endDate : Calendar
+ExternalEvent(eventName : String)
+getEventName() : String
+setEventName(name : String) : void
+getStartDate() : Calendar
+setStartDate(startDate : Calendar) : void
+getEndDate() : Calendar
+setEndDate(endDate : Calendar) : void
+addItem(itemType : ItemType) : Item
+getItemByTitle(title : String) : List<Item>

Admin
+Admin()

Staff
+Staff()

<<enumeration>>
Platform
PC
Playstation
XBox
```

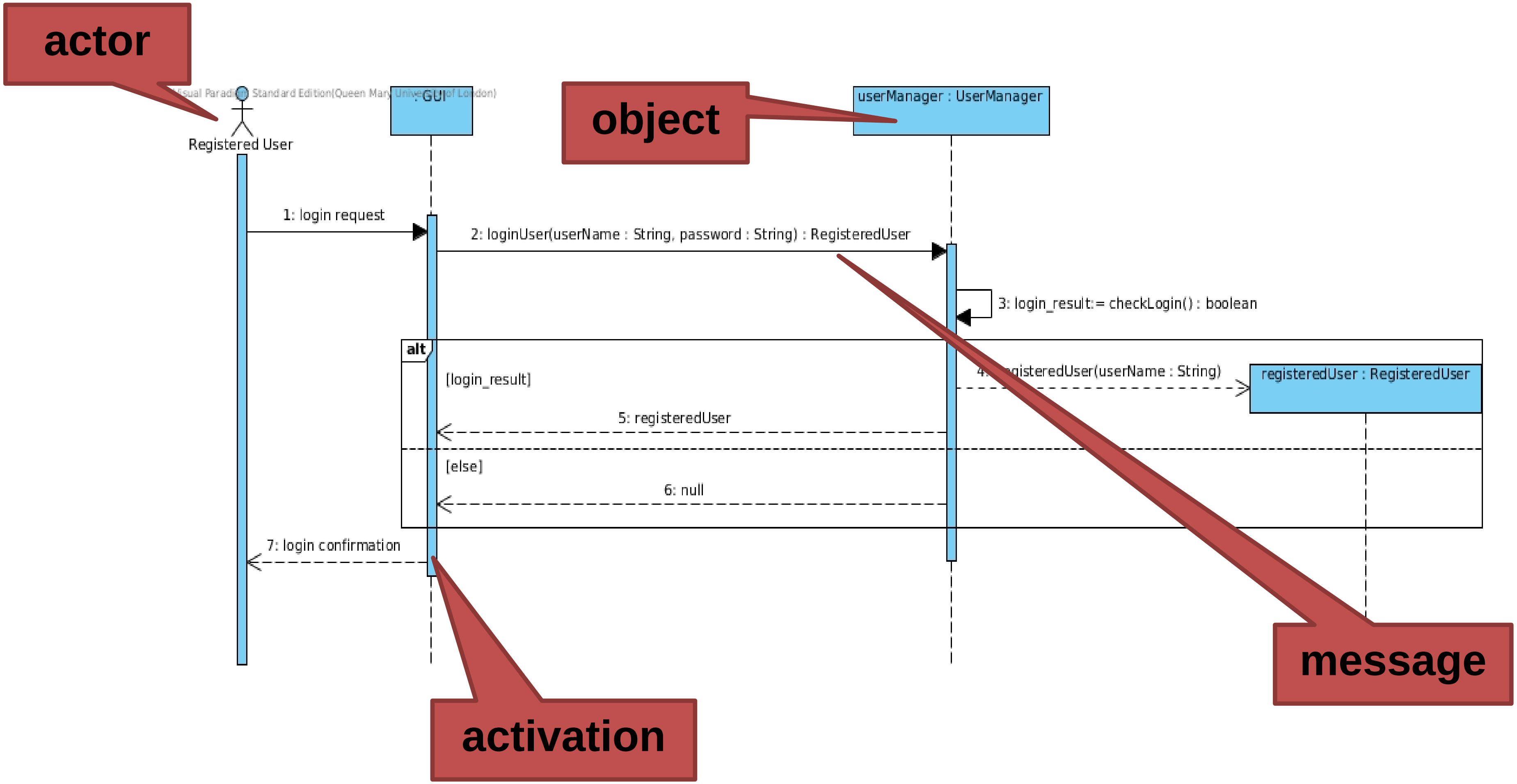
multiplicity

association

composition

Class diagrams represent the structure of the system

UML: SEQUENCE DIAGRAM


actor

object

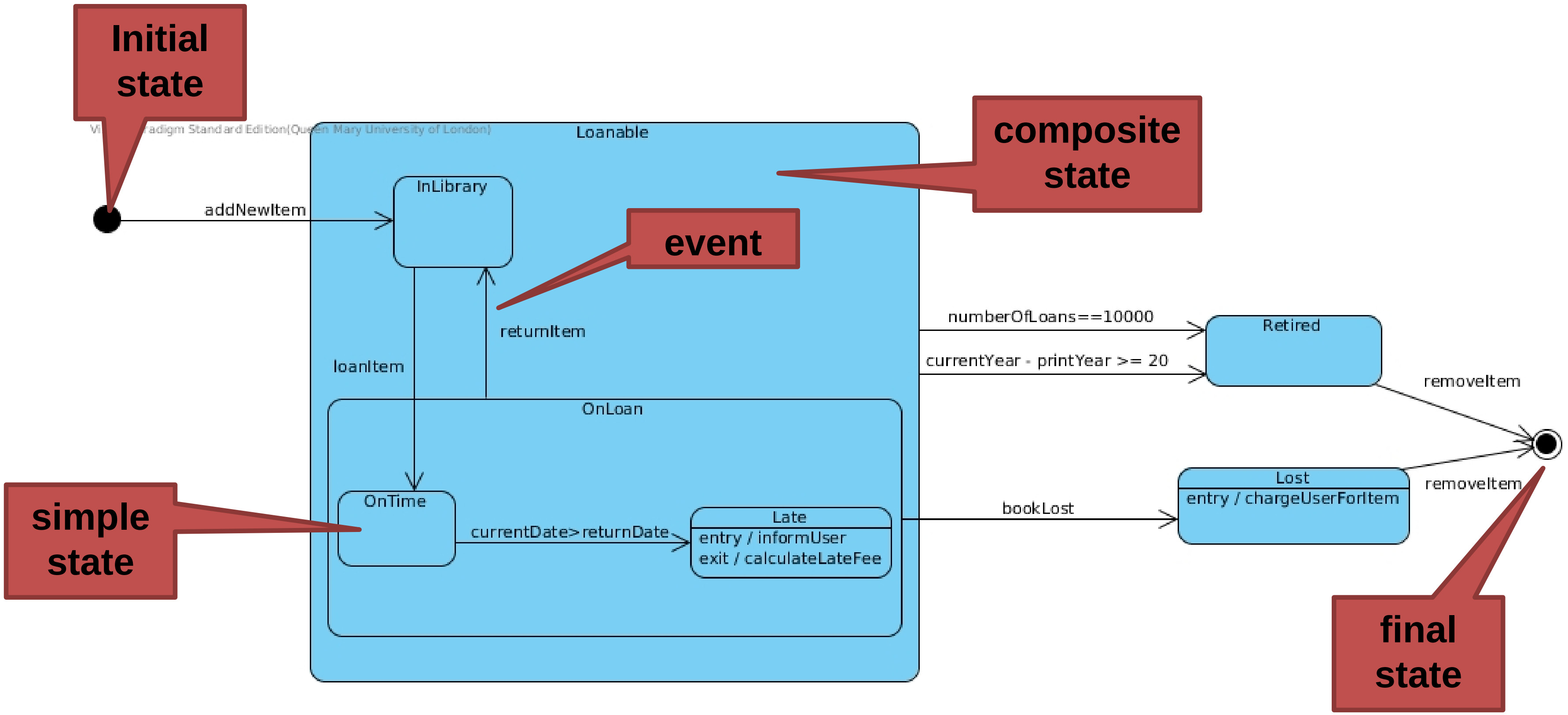
message

activation

Visual Paradigm Standard Edition(Queen Mary University of London)
Registered User
: GUI
userManager : UserManager
1: login request
2: loginUser(userName : String, password : String) : RegisteredUser
3: login_result:= checkLogin() : boolean
alt
[login_result]
4: registeredUser(userName : String)
registeredUser : RegisteredUser
5: registeredUser
[else]
6: null
7: login confirmation

Sequence diagrams represent the behavior as interactions

STATECHART DIAGRAMS



Describe dynamic behaviour of an individual object as a finite state machine

OTHER UML NOTATIONS

- Activity diagrams
- Implementation diagrams
  - Component diagrams
  - Deployment diagrams
- Object Constraint Language (OCL)

We will not be using any of the above on this course

LESSON SUMMARY

- Classes are abstractions of objects
- Objects within a class have common attributes and operations
- Objected-oriented software design is all about identifying appropriate classes
- UML provides a wide variety of notations for representing many aspects of software development
- We can concentrate only on a subset of the UML notations