

INTRODUCTION

Programming Principles II

Topics

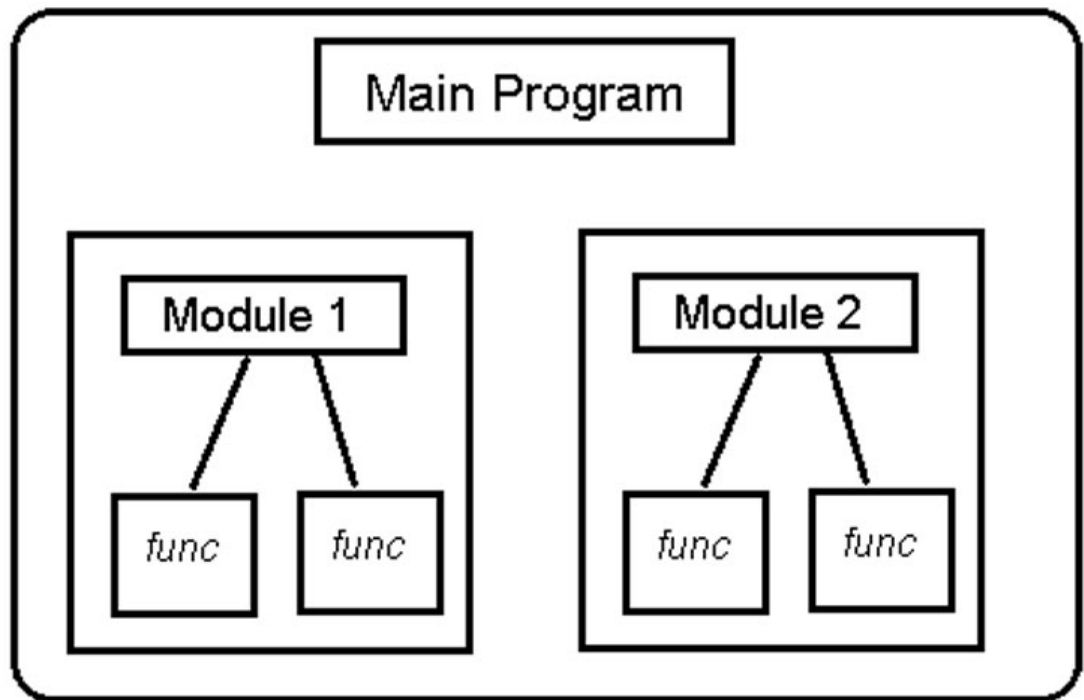
Classes and Objects

Modular Programming

In a *Programming Principles I* course we mentioned that the functions can be declared and implemented in different files and included into the main code.

This ***modular programming*** approach is widely used in ***procedural programming languages***.

Modular programming is a software design technique that emphasizes separating the functionality of a ***program*** into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality ([Wiki](#)).



Object Oriented Programming

In Object-Oriented Programming, the application is designed around Objects that have properties and behavior.

OOP is a programming paradigm based on the concept of “objects”, which may contain data and code;

- *data in the form of fields (a.k.a. fields/attributes)*
- *code in the form of procedures (a.k.a. methods) ([Wiki](#))*

Before explaining it in detail, let's see what an object itself is.

Consider this task...

You want to write a program that keeps the student and course data, and provides functionality that performs some operation:

- Prints the list of students enrolled to a particular course
- Get the courses of a student that he/she has enrolled to
- Calculate the current GPA of the student

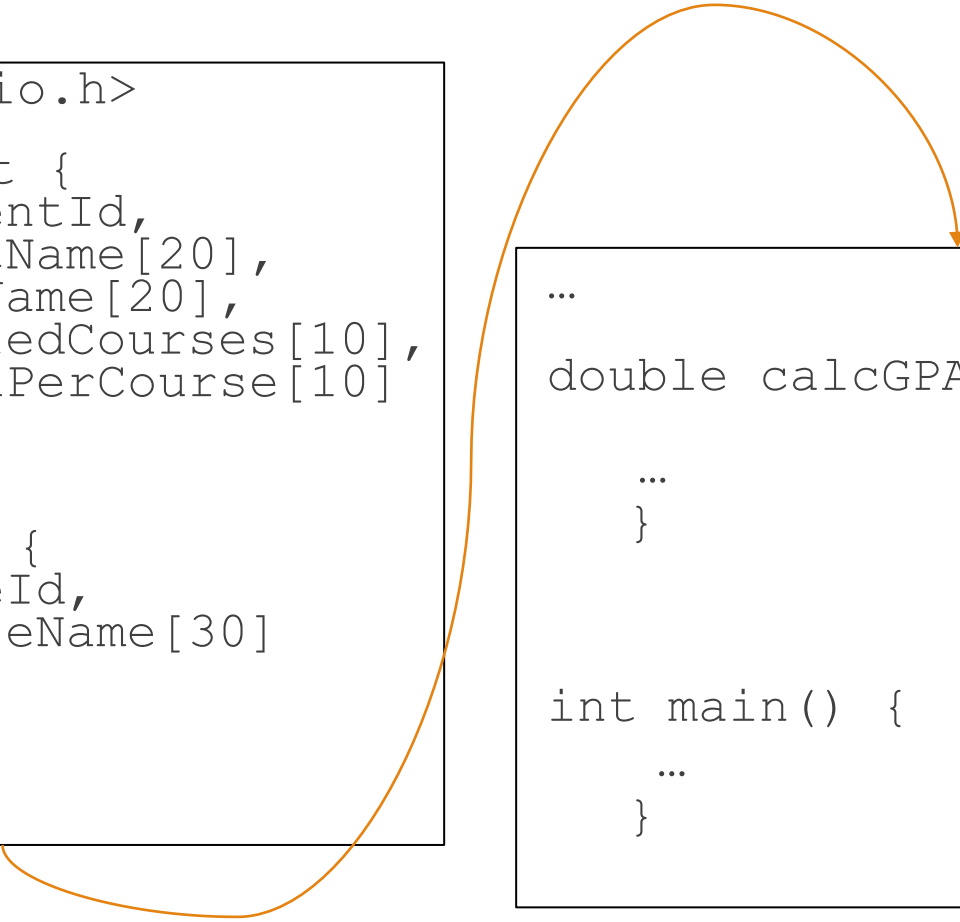
Traditional C implementation

```
#include <stdio.h>

struct Student {
    int studentId,
    char firstName[20],
    char lastName[20],
    int enrolledCourses[10],
    double gpaPerCourse[10]
};

struct Course {
    int courseId,
    char courseName[30]
};

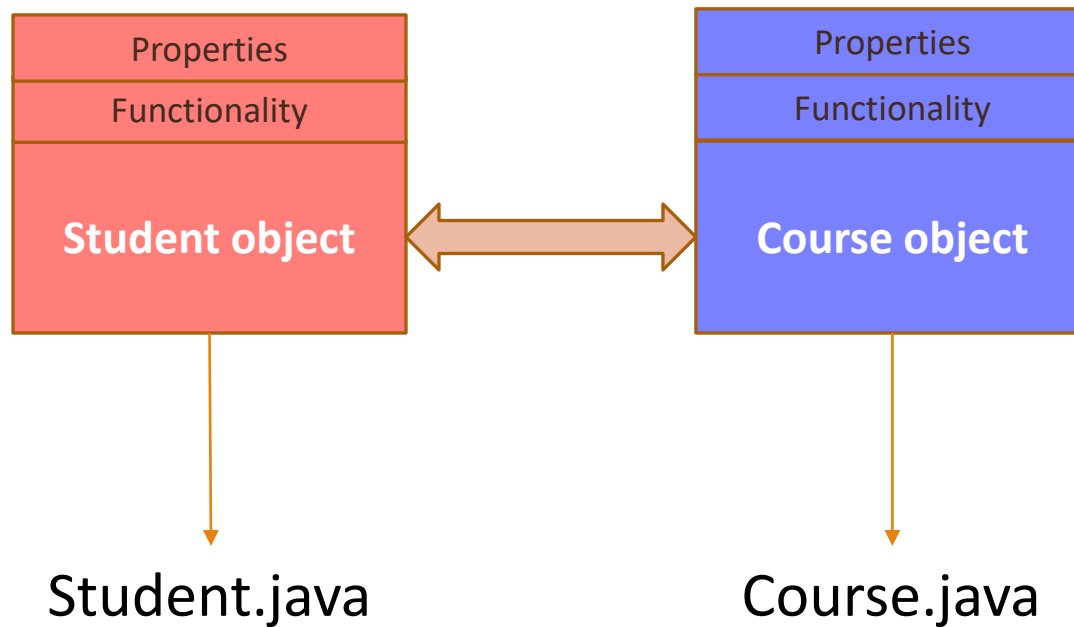
...
```



```
...
double calcGPA(int stId) {
    ...
}

int main() {
    ...
}
```

Object Oriented Approach



The Wolf as an object

Let's list the *properties* and *functionality* of the Wolf



The Wolf

Properties

- Eyes
- Nose
- Ears
- Teeth
- Legs
- Paws
- Tail
- Hair (color)
-

Functions

- Eats
- Drinks
- Bites
- Makes sound (howl)
- Walks
- Runs
- Jumps
-

Java Classes (anatomy)

```
class ClassName {
```

```
    type property1;
```

```
    type property2;
```

```
    ...
```

```
    type propertyN;
```



Properties / members / field / **state**

```
    type function1(...) { ... }
```

```
    type function2(...) { ... }
```

```
    ...
```

```
    type functionM(...) { ... }
```

```
}
```



Functionality / methods / **behavior**

The Wolf as a Java Class

```
class Wolf {  
    int eyes = 2;  
    int nose = 1;  
    int ears = 2;  
    int teeth = 42;  
    int legs = 4;  
    int paws = 4;  
    int tail = 1;  
    String hairColor = "gray";
```

Properties / members / field / **state**

```
    void eat(Meat someMeat) { ... }  
    void drink(Water someWater) { ... }  
    void bite(Object someObject) { ... }  
    void makeSound(Sound someSound) { ... }  
    void walk(int x, int y) { ... }  
    void run(int x, int y) { ... }  
    void jump(int x, int y, int height) { ... }
```

Functionality / methods / **behavior**

```
}
```

Class names

- If class is declared as ***public***, it should be saved in a **file with the *same name***. If you have several classes in one file, then make sure that **exactly one (optional: with the `main()` method) is declared as public**.
- In class names words start with capital letter, like *MyFirstClass* or *DataOrganizer*. Do not use UPPERCASE, underscore (“_”) in class names.

What do we mean by WORK?

Objects

Class defines a **new type**. In the provided example, we created a `Wolf` type. To make it **work**, we need to create an **object**:

```
Wolf rocky = new Wolf();
```

We just created a new ***Wolf object*** called ***rocky***. If we run the below code, we'll get the number of its teeth:

```
System.out.println("Rocky has "+rocky.teeth+" teeth");
```

Objects

The properties of the Class are called *members*:

rocky.teeth, rocky.ears, rocky.hairColor

The functions of the Class are called *methods*:

rocky.walk(5,3), rocky.run(35,3), rocky.makeSound(hawl)

Questions



Topics

Methods

Method overloading

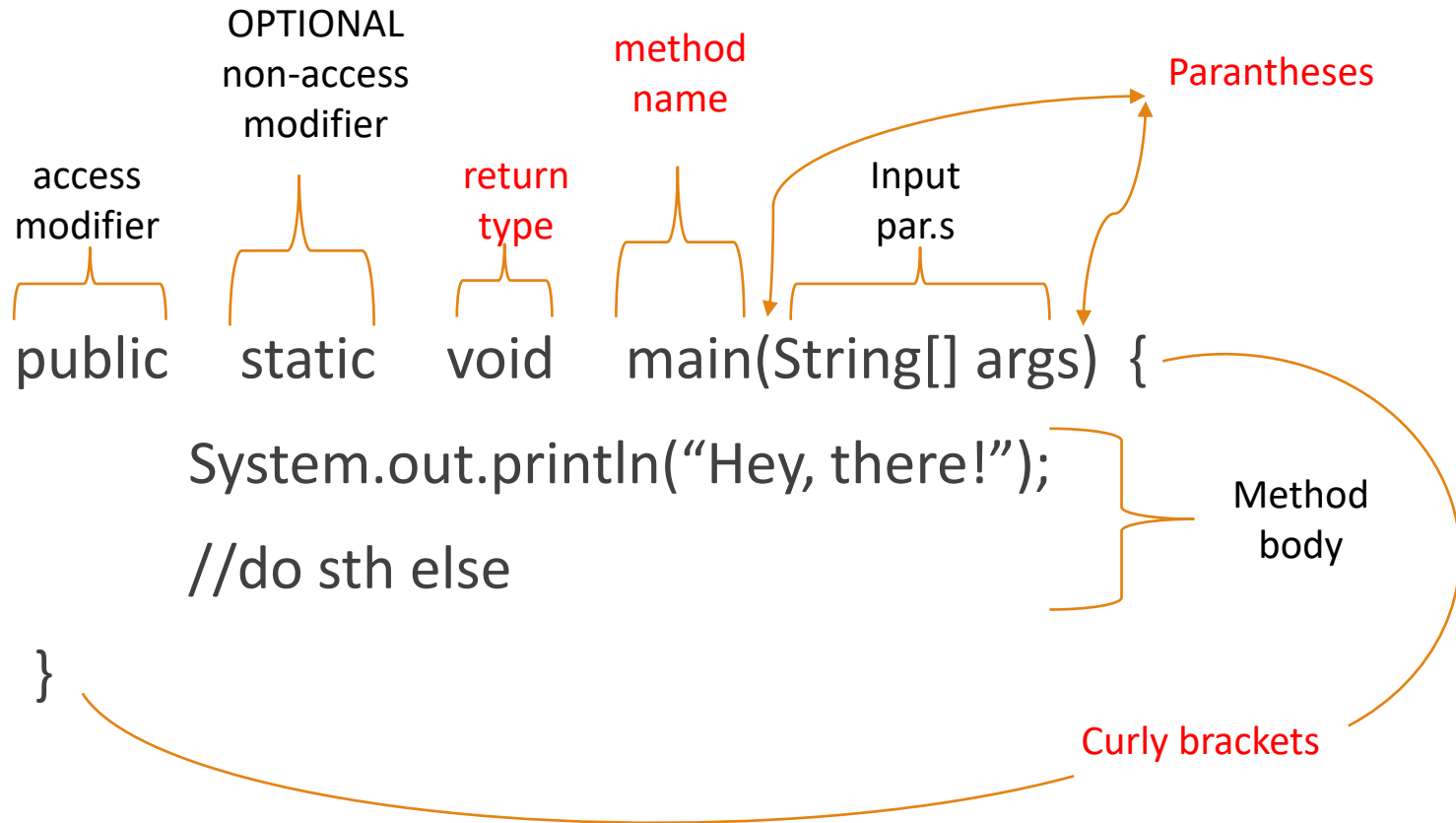
Methods

Methods are regular functions that define the behavior of the object. It can take any name, the **keywords**.

It may return **any type** or **void** as in C programming language.

```
class MathOper {  
    double a, b;  
  
    void setValues(double x, double y) {  
        a = x; b = y;  
    }  
  
    double sum() { // no parameters are given – object properties will be used  
        return a + b;  
    }  
}
```

Methods (cont.)



Methods (cont.)

Unlike C, there can be **more than one method** with the same name within the same class, **if and only if**,

1. Different number of input parameters
2. The same number of input parameters with different (opt: order of) data types.

```
class PrintUtil {
```

```
void print() { System.out.println("Default print!"); }
```

```
void print(int value) { System.out.println("Integer argument: " + value); }
```

```
void print(float value) { System.out.println("Float argument: " + value); }
```

```
void print(int val1, float val2) {  
    System.out.println("Int arg: " + val1 + ", Float arg: " + val2); }
```

```
}
```


method
overloading

Methods (cont.)

In java, you can design a method so that it receives an *arbitrary number of arguments* of the same data type using **varargs**.

public void numOper1(int... nums){} 

public void numOper2(int num1, int... nums){} 

public void numOper3(int... nums, int num1){} 

Be careful with method overloading!!!

Read more: [Late binding or dynamic binding](#)

Topics



Using classes

Constructors

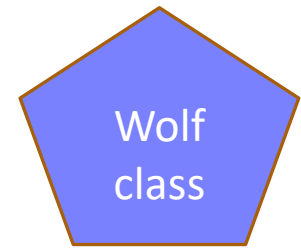
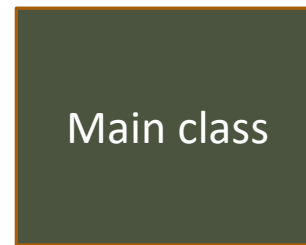
Access Modifiers

- public
 - accessible from anywhere
- private
 - accessible from only within the class
- DEFAULT (package-protected)
 - accessible from anywhere within the same package and the within the class itself
- protected
 - accessible from anywhere as DEFAULT
 - in addition: child classes anywhere

Defining and using classes

Option 1: Two separate (source) files

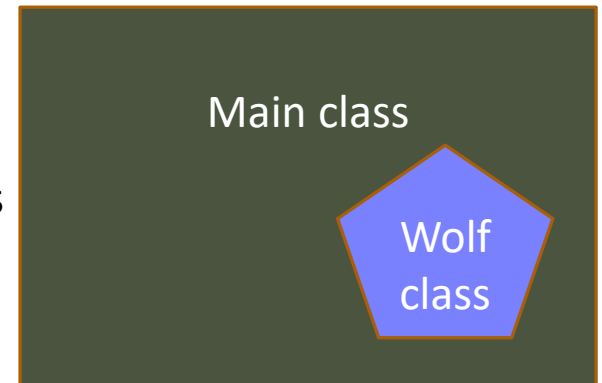
Main.java and Wolf.java



Option 2: Both classes are in the same file

The fileName (source file) shall be the same as the Main class

The Main class shall be declared as **public**



Using Objects

Create a MyApp.java file and write the below code in it:

```
// include the Wolf class in your code:
class Wolf {
    ...
}

// This is the main class of the application
public class MyApp {

    public static void main( String args[]) {
        Wolf rocky = new Wolf();
        System.out.println("Rocky has "+rocky.teeth+" teeth");
    }

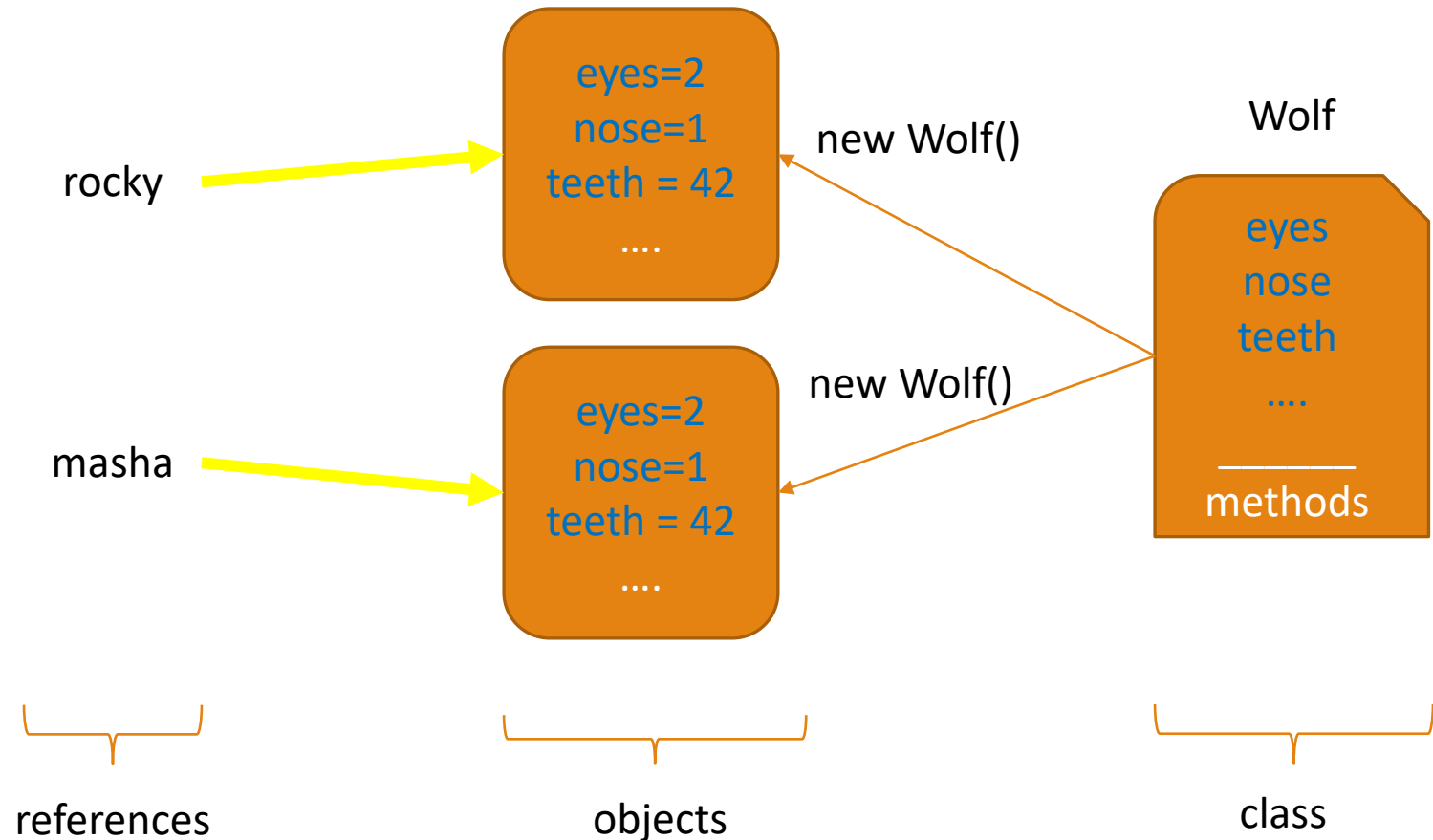
}
```

*Note that: after the compilation (javac MyApp.java) two classes created: **Wolf** and **MyApp***

Two Wolf objects

```
class Wolf {  
    ...  
}  
  
public class MyApp {  
  
    public static void main( String args[]) {  
        Wolf rocky = new Wolf();  
        Wolf masha = new Wolf();  
  
        rocky.teeth = 40; // we set the tooth count for Rocky  
        masha.teeth = 38; // we set the tooth count for Masha  
  
        System.out.println("Rocky has "+rocky.teeth+" teeth");  
        System.out.println("Masha has "+masha.teeth+" teeth");  
    }  
  
}
```

Two Wolf objects (cont.)



On creating objects

Class is a file that declares the structure of the object. (**schema** or **template** or **blueprint** of the objects to be defined)

Object is the instance of the class.

The *new* keyword creates an *instance of an object*. In below example ***someName*** is not an object – it is a *reference* to the created object:

SomeClass **someName** = new SomeClass();

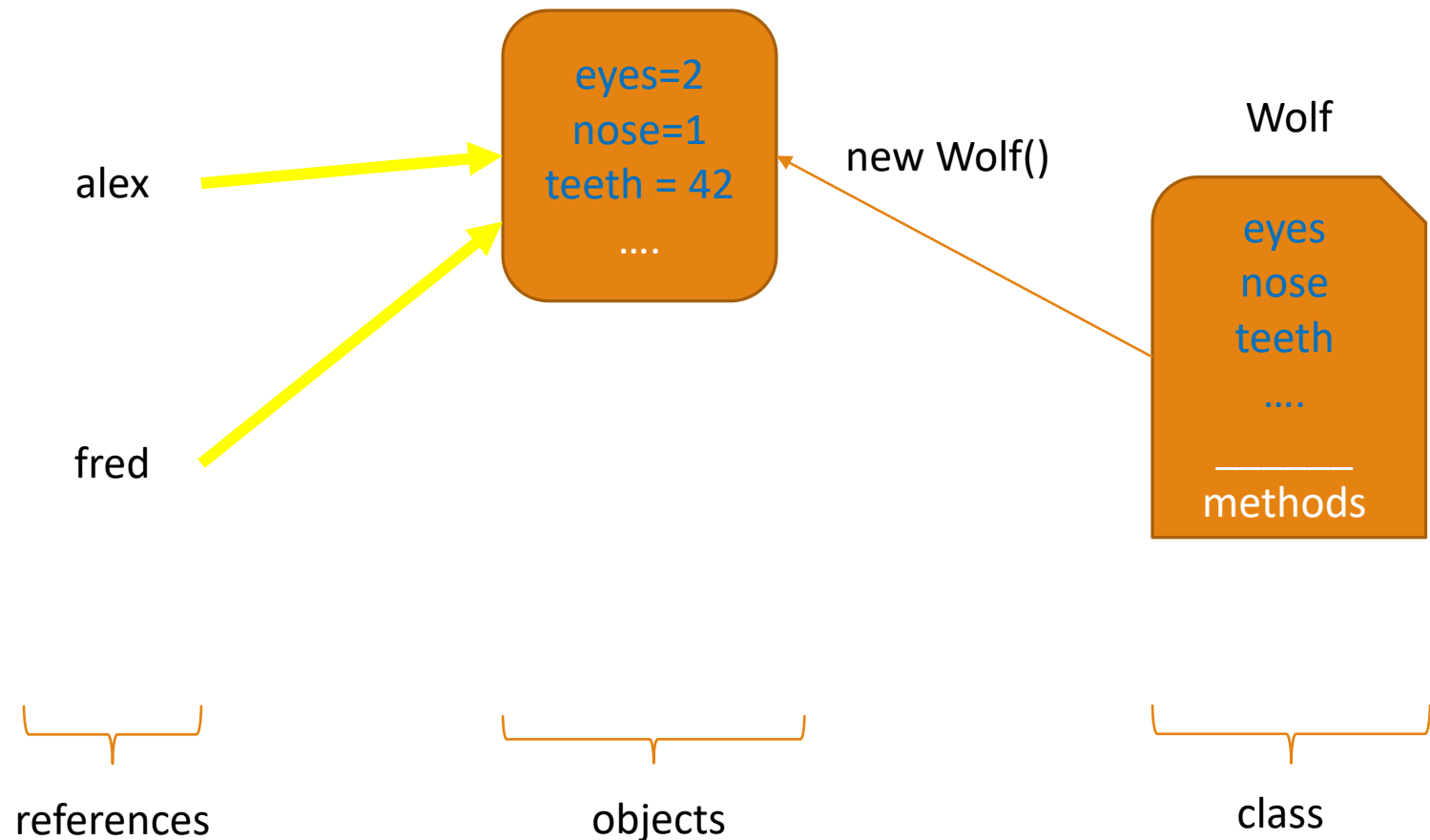
Reference to the object The object

On creating objects (cont.)

In below example, both variables refer to the same object and changing the value of one, will impact another as well:

```
Wolf alex = new Wolf();  
Wolf fred = alex;  
  
System.out.println("Alex has "+alex.teeth+" teeth");      // 42  
System.out.println("Fred has "+fred.teeth+" teeth");      // 42  
  
alex.teeth = 35;  
  
System.out.println("Alex has "+alex.teeth+" teeth");      // 35  
System.out.println("Fred has "+fred.teeth+" teeth");      // 35
```

On creating objects (cont.)



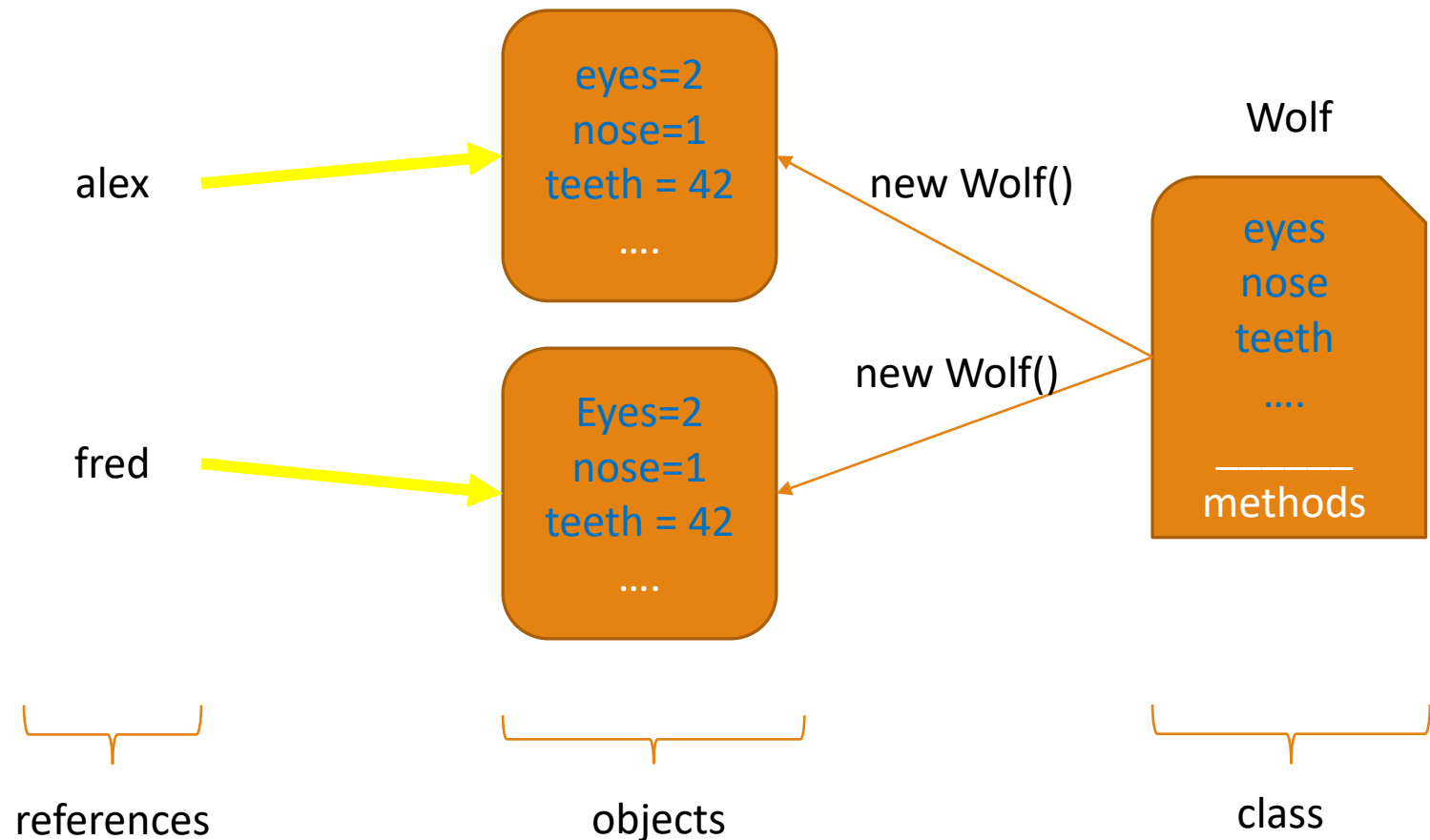
On creating objects (cont.)

Now, when we create two different objects, each variable will refer to the corresponding object:

```
Wolf alex = new Wolf();  
Wolf fred = new Wolf();  
  
System.out.println("Alex has "+alex.teeth+" teeth"); // 42  
System.out.println("Fred has "+fred.teeth+" teeth"); // 42  
  
alex.teeth = 35;  
  
System.out.println("Alex has "+alex.teeth+" teeth"); // 35  
System.out.println("Fred has "+fred.teeth+" teeth"); // 42
```

Homework: test the provided behavior.

On creating objects (cont.)



Constructors

In previous initializations of the Wolf object, we would just call “new Wolf()”.

Wolf() is an **empty** or **default constructor** of the class that just creates an object with default values. The format of the **empty constructor** is as follows:

```
ClassName() {  
    //some code here  
}
```

Constructors have the same name as class name and do not have a return type. When class has no constructor, the empty constructor is added by default. That is why it's called a **default constructor**.

Note: constructors are not methods (functions).



Constructors (cont.)

Classes can have several constructors:

```
class Wolf {  
    // members declared here.  
  
    Wolf() { // an empty constructor  
        legs = 4; eyes = 2; ...; hairColor = "gray";  
    }  
  
    Wolf(String color) { // constructor that sets wolf's color upon creation  
        legs = 4; eyes = 2; ...; hairColor = color;  
    }  
  
    Wolf(int eyeCnt, String color) { // eyes and color are set  
        legs = 4; eyes = eyeCnt; ...; hairColor = color;  
    }  
  
    // remaining code  
}
```

constructor
overloading

Constructors (cont.)

Now the Wolf object can be created in a different way:

```
// With default parameters
Wolf alex  = new Wolf();

// With a color parameter
Wolf jax   = new Wolf("white");

// With eye and color parameters
Wolf susan = new Wolf(2, "white");
```

Constructors (cont.)

There can be also a copy constructor:

```
class Wolf {  
    // members declared here.  
    Wolf() { /* an empty constructor */}  
    Wolf(String color) { /* constructor that sets wolf's color upon creation */  
}  
    Wolf(int eyeCnt, String color) { /* eyes and color are set */}  
    Wolf(Wolf anotherWolf) { /* making a copy of a wolf object */  
        eyeCnt = anotherWolf.eyeCnt;  
        color = anotherWolf.color; //this applies to all the fields  
    }  
    // remaining code  
}
```

Constructors (cont.)

Now the Wolf object can be created using another object of the same type as well:

```
// With default parameters
Wolf alex  = new Wolf();

// With a color parameter
Wolf jax   = new Wolf("white");

// With eye and color parameters
Wolf susan = new Wolf(2, "white");

// With the copy constructor
Wolf copyOfJax = new Wolf(jax);
```

The “*this*” keyword

“*this*” keyword helps refer to the object itself inside the class code:

```
class MathOper {  
    double a, b;  
  
    void setValues(double a, double b) {  
        this.a = a;  
        this.b = b;  
    }  
}
```

Questions



Topics

Encapsulation
Association

Encapsulation

```
class ClassName {  
  
    private type1 property1;  
    private type2 property2;  
    ...  
    private typeN propertyN;  
  
    type1 getProperty1() {  
        return this.property1;  
    }  
  
    void setProperty1(type1 property1) {  
        this.property1 = property1;  
    }  
    ...  
}
```

Properties /
members /
fields /
state

Functionality /
methods /
behavior

Encapsulation

Read more: [https://en.wikipedia.org/wiki/Encapsulation_\(computer_programming\)](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming))

Association

Association is a relationships build between objects of different types.

There are **two** types of association: aggregation and composition.

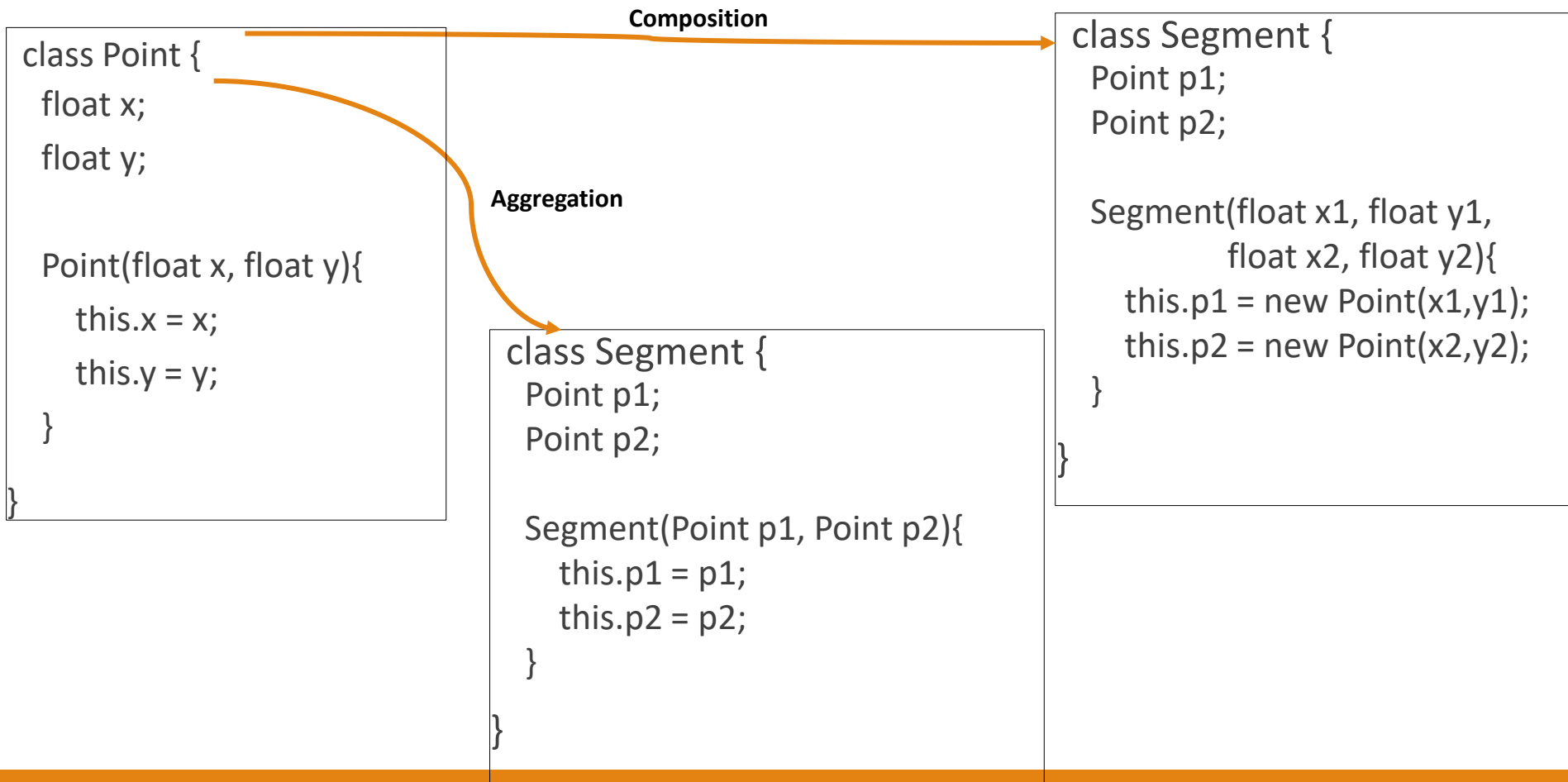
Aggregation: in this type of relationships the objects involved do not necessarily need each other to survive.

Example: Car and Windshield Wiper

Composition: in this type of relationships the objects involved strongly dependent on each other, one cannot survive without the other.

Example: House and Dining Room

Association



Topics

UML class diagrams

UML

Unified Modeling Language (UML) is used to systematically and semantically design complex software systems and communicate easily. There are some more applications, but we will talk about software design only.

Developed: Rational Software (1995)

Adopted as a standard:

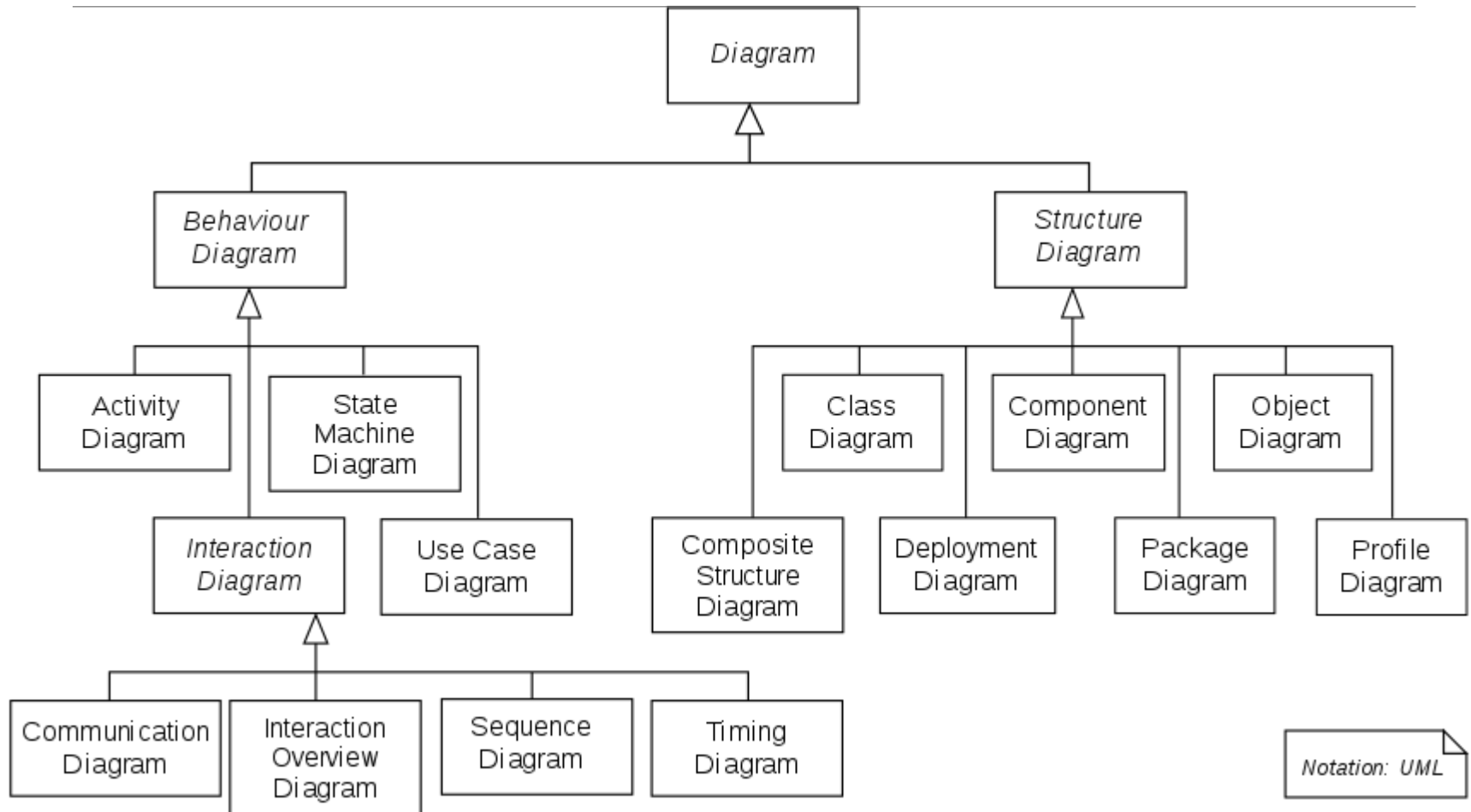
- Object Management Group (1997)
- International Organization for Standardization (ISO)

Latest versions: UML 2.x

Online tools:

- <https://app.diagrams.net/>

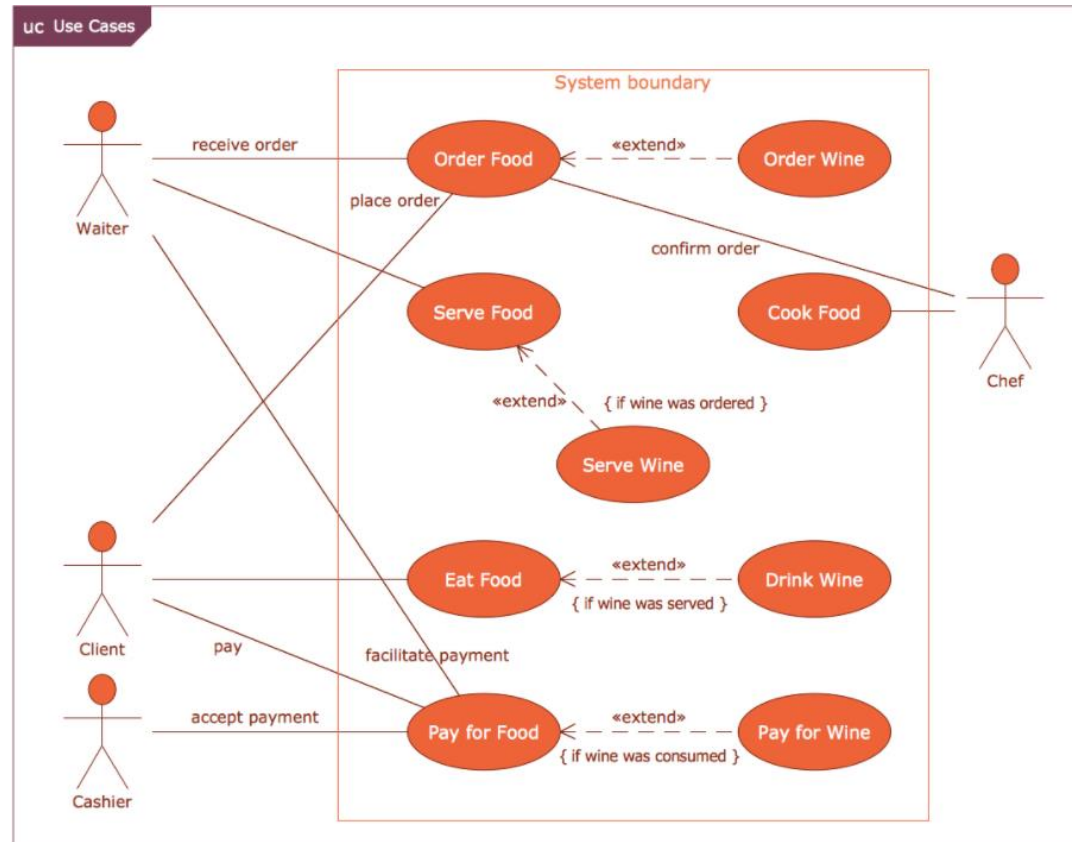
UML diagrams



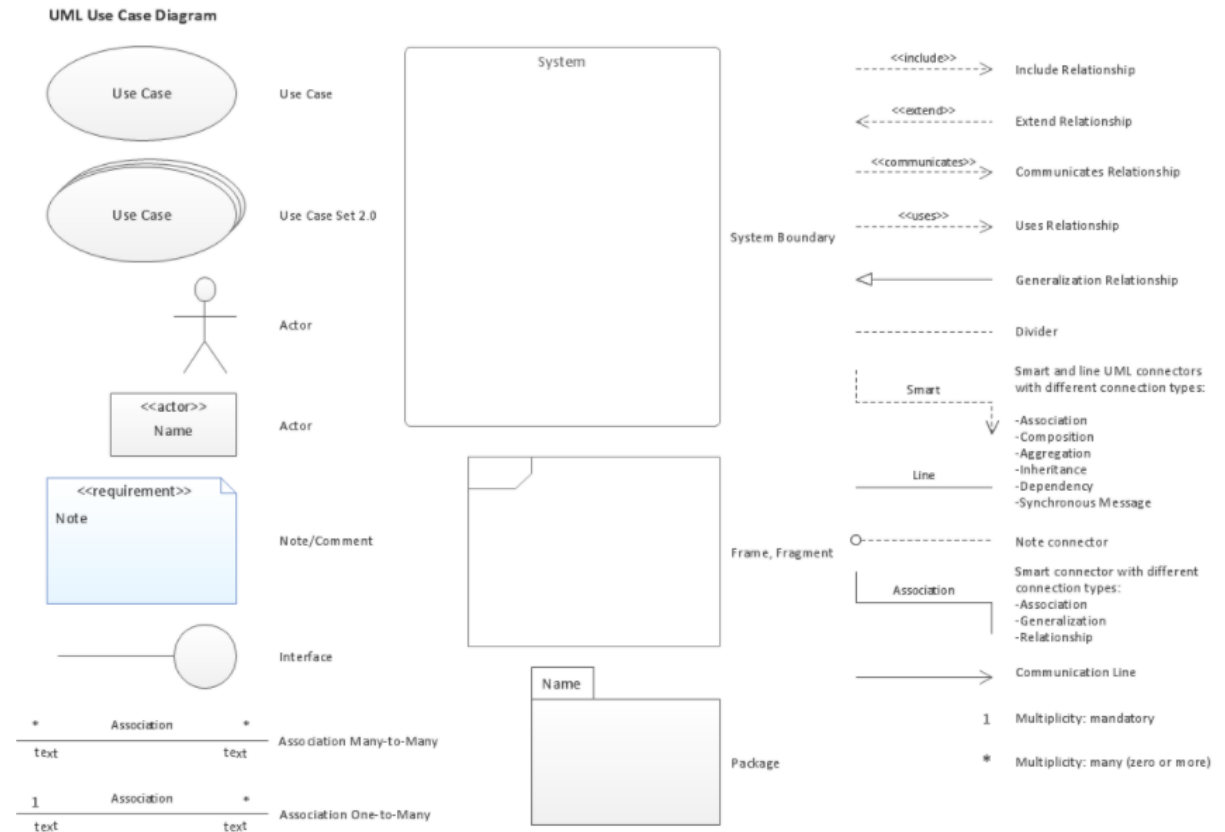
UML – Use-case diagrams

Use-case diagrams are used to graphically represent how users and software (system) interacts.

It is one of the easily understood communication tool (medium) for stakeholder.



UML Use-case diagrams (components)

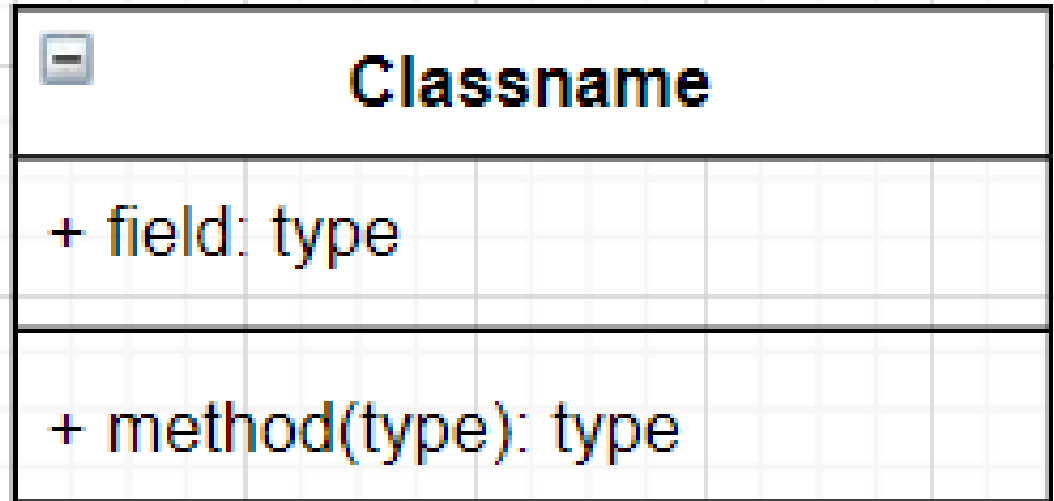


UML – Class diagrams

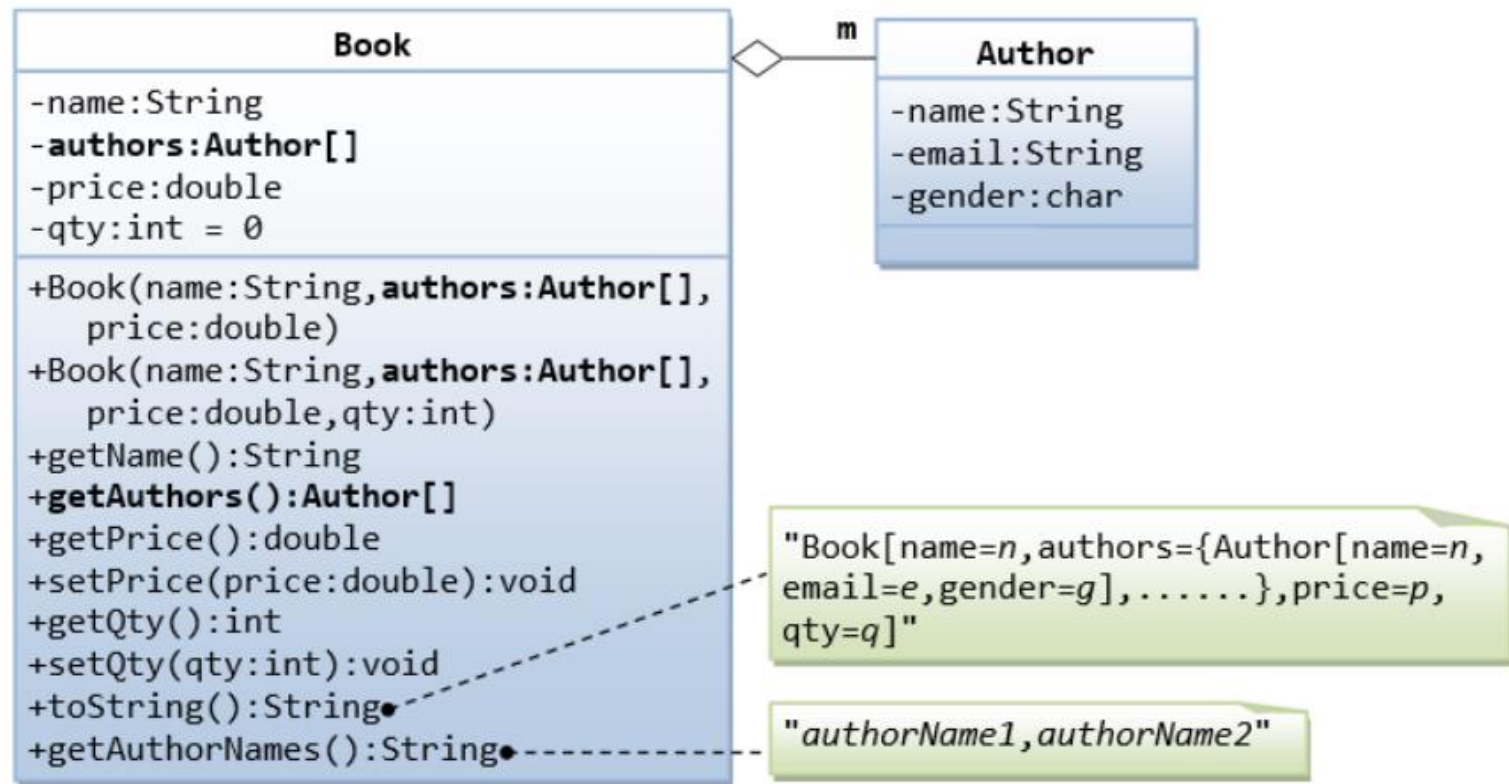
Class diagrams are used to graphically represent the structure of the system by showing its classes, and

- Their attributes and behavior
- Relationships between these classes and objects

It is a powerful tool to do object-oriented design and as well communicate easily.



UML – Class diagrams



UML – Class diagrams - components

Visibility – The members of the class can have different accessibility levels, thus different ways of representing them in a class diagram:

+ Public

- Private

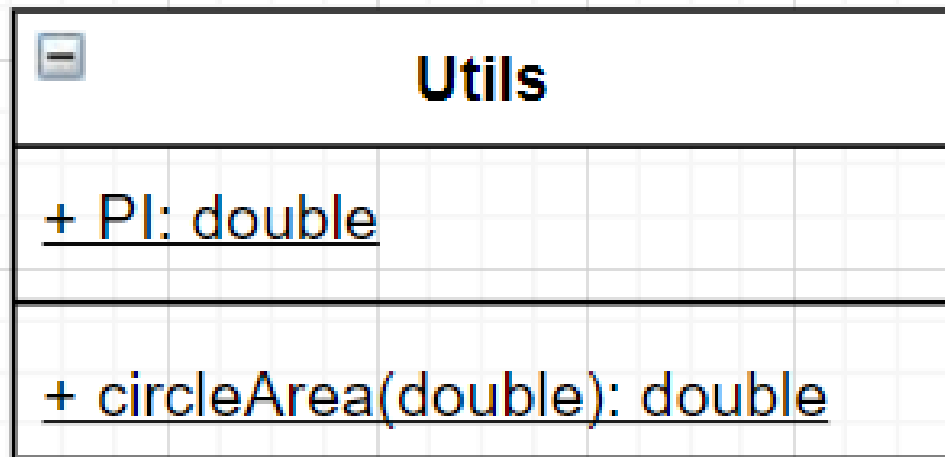
Protected

~ Default (package-protected)

UML – Class diagrams - components

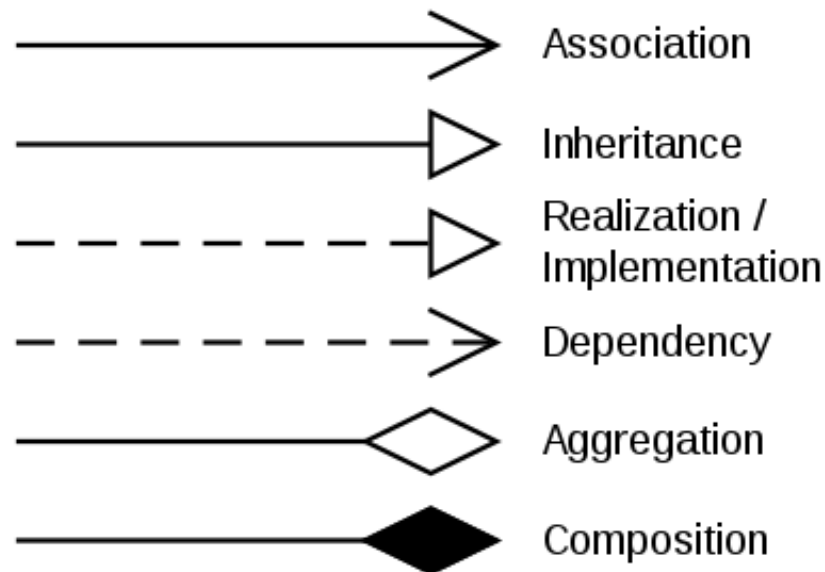
Scope – The members of the class can be either **instance** or **static**, thus different ways of representing them in a class diagram:

The only difference is the **static** is represented in **underlined**.



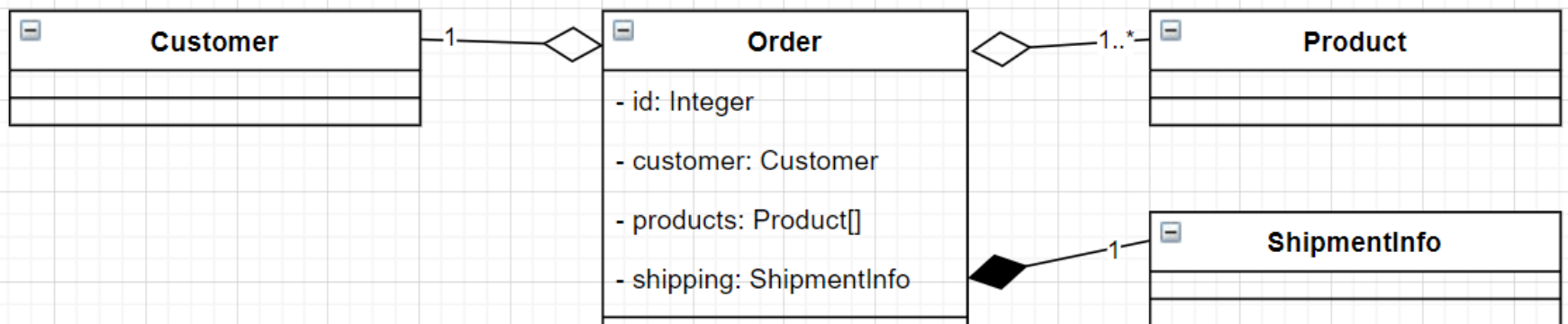
UML – Class diagrams - components

Relationships – classes and their objects can have relationships with other classes or their objects, thus different ways of representing them in a class diagram:

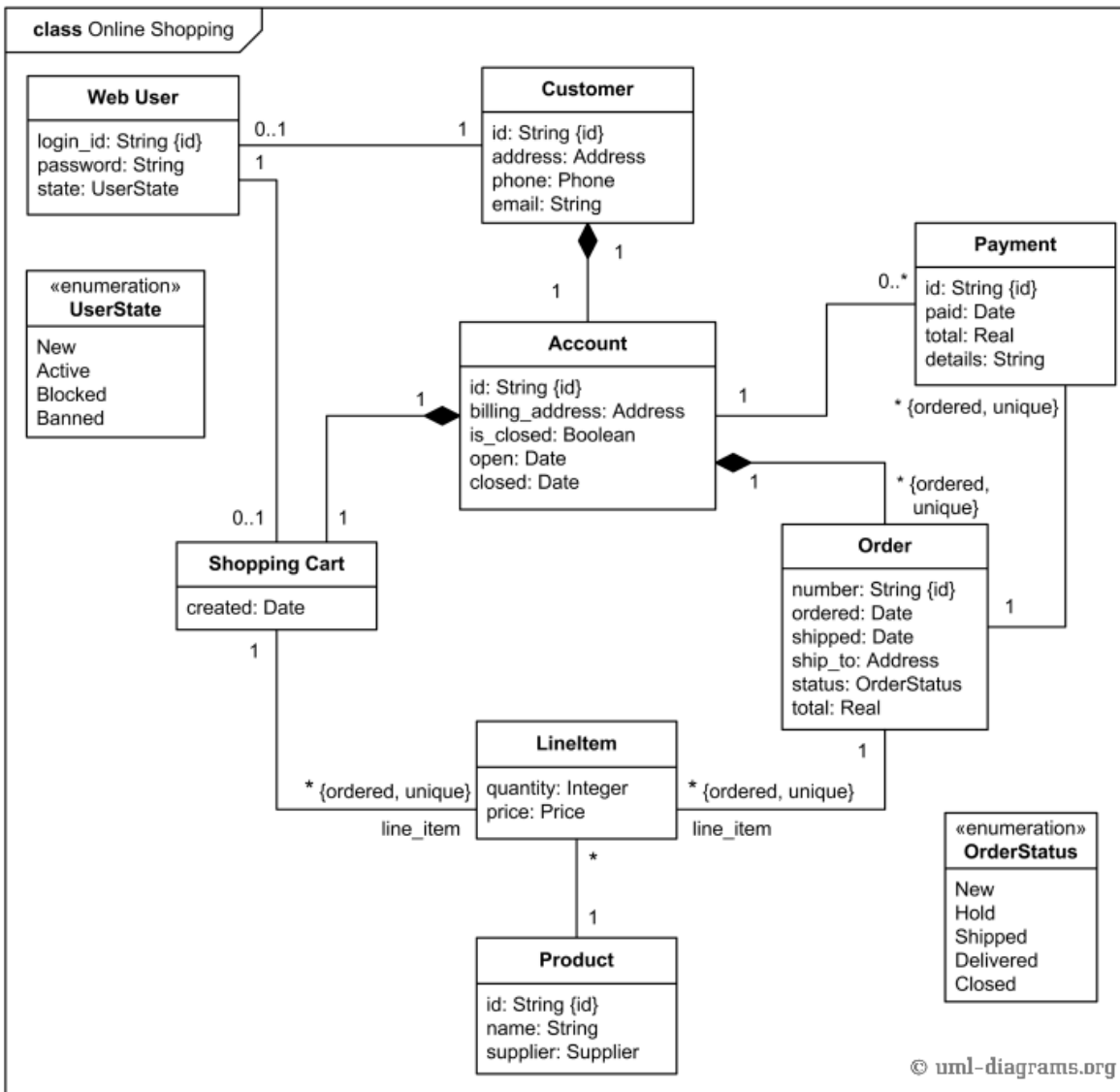


UML – Class diagrams - components

Aggregation vs Composition in class diagrams:



UML – Class diagrams - example



Questions

