320341 Programming in Java

JACOBS
UNIVERSITY

Fall Semester 2015

Lecture 3: Object Oriented Programming

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Outline

JACOBS UNIVERSITY

Fundamental Principles

- **Abstraction**
- **Encapsulation**
- Method
- Message Passing
- **Inheritance**
- Object
- Class
- **Polymorphism**
- Taxonomy

Objectives

Two main objectives

- Introduce the principles of object oriented programming
- Apply the principles to real application scenarios

Terminology

Abstraction

Encapsulation

Inheritance

Polymorphism

Object

Class

Method

Message Passing

Introduction

Object-oriented programming (OOP) has become the predominant programming paradigm

- Replaces structured procedural paradigms of the 1970s
- Suitable for large and complex software
- Based on modeling the problem domain thus problem oriented
- More modular with code being organized into classes that intercommunicate

Abstraction

JACOBS UNIVERSITY

Basic Definition

- The act of creating classes to simplify aspects of reality using distinctions inherent to the problem

Earliest application of symbolic abstraction to programming languages was in late 1950s with symbolic assemblers

Class

The template or blueprint from which objects are created

- The basic element of object-oriented modeling
- Description of the organization & actions shared by one or more similar objects
- Creating an object from a class is called creating an `instance of a class`

Performs the following functions

- During development, provides *interface to interact with definition of objects*
- At runtime describes *how objects behave in response to messages*
- At runtime it is a *source of new objects*

Class

Example

Object state

Object behavior

```
class Account {

    private String accNumber;

    private double balance;


   public  double getBalance() {
        return balance;
    }
   public  void deposit(double amount) {
        balance += amount;
    }
   public  void withDraw(double amount) {
        if (balance >= amount)
            // execute withdrawal
    }


}
```

Instance fields

methods

Object

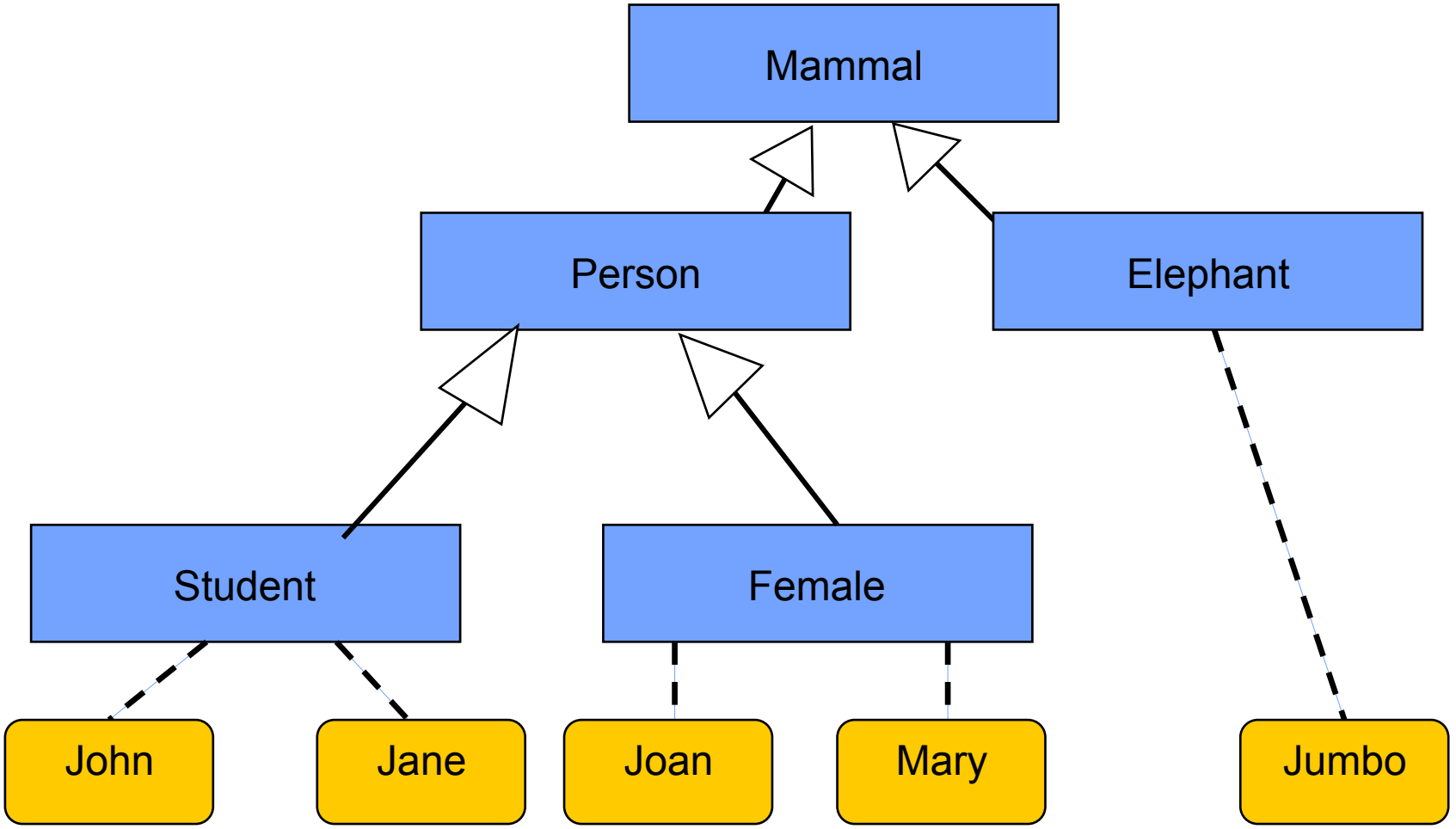
An identifiable item either real or abstract with well-defined role in problem domain

Three key characteristics of an object

- An object has ***state*** – represented by the object's data model
- An object has ***behavior*** – represented by methods you can call
- An object has ***identity*** – used to distinguish different objects with the same behavior and state
- Both a **data carrier** & **executes actions** (Robson, 1981)

Object

An object is an *instance of a class* *(classes in blue, instances in brown)*



Object

JACOBS UNIVERSITY

More General Definition (Armstrong, 2006)

- An ***individual***, ***identifiable*** item (real or abstract)
- Contains ***data*** about itself & ***descriptions of its manipulations*** of the data
- Example: `John, Jane, Bill, Mary` and `Jumbo` are all objects

Encapsulation

Combining data and behavior in one class and hiding implementation details from users of the object

- **Implementation hiding + Wrapping data & methods within classes**

Implementation Hiding

- Separates *things that change* from *things that stay the same*

- Library developers want to be able to do modifications/ improvements such that client programmers' code is not affected

- Achieved by *hiding implementation specific details* of the library implementation

Encapsulation

JACOBS UNIVERSITY

Implementation Hiding cont…

- Access specifiers (public, private, protected, “friendly”) are used to distinguish what’s available to client programmers and what’s not
- Access control is usually referred to as “implementation hiding”
- For encapsulation to work, methods should never directly access instance fields in a class other than their own
- Programs should interact with objects only through the object’s methods
- Thus encapsulation gives objects “Black-Box” behavior which is key to reuse

Example

Method used internally

Hidden Method

```
class Account {

    private String accNumber;

    private double balance;


  public  double getBalance() {
        …
        return checkCredit(accNumber);
   }
   public  void deposit(double amount) {
        balance += amount;
   }
   public void withDraw(double amount) {
        if (balance >= amount)
            // execute withdrawal
   }
  private double checkCredit(String accNum) {
        // check account credit
   }

}
```

Method

A way to *access*, *set* or *manipulate* an object's information

- The fundamental element of an object-oriented program
- Typically a method ***sends messages*** to other objects that invoke the object's methods

Example

```
class Account {

    private String accNumber;

    private double balance;


  public  double getBalance() {
        return balance;
   }
  public  void deposit(double amount) {
        balance += amount;
   }

}
```

Class methods

Message Passing

JACOBS UNIVERSITY

The process by which *an object sends data to another object or asks the other object to invoke a method*

Example

```
class Account {
     static private String accNumber = “001“;
     static private double balance = 200.0;

   public static double getBalance() {
         return balance;
    }
}
public class Client {
    public static void main(String[] args) {
        double bal = Account.getBalance();
        System.out.println("Balance is "+bal);
    }
}
```

Message passing

Inheritance

Introduced in 1967 in **Simula** programming language

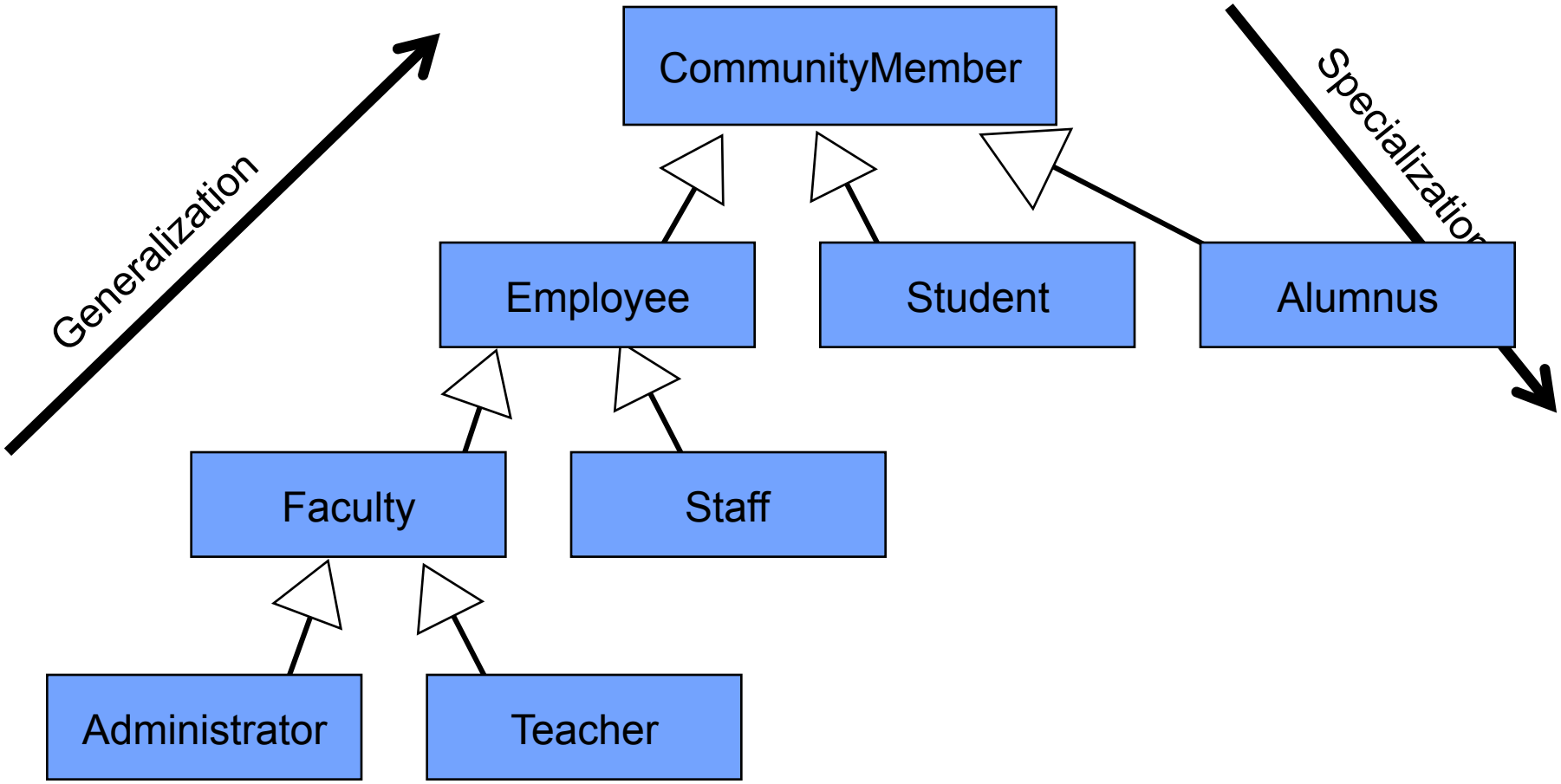
- Mechanism that allows *data* and *behaviour* of one class to be included in or used as the *basis for another class*

Example

- Mammal: Person & Elephant subclasses
- Person: Mammal as superclass

- Person: Student & Female subclasses

Example



- Expresses **classification**, **specialization**, **generalization**, **approximation**
- It captures some form of **abstraction** called *super-abstraction*

Polymorphism

JACOBS UNIVERSITY

General concept

- ***The ability of different objects to respond to the same message and each implement the method appropriately***

Polymorphism

JACOBS UNIVERSITY

Basic Concept

- Separates interface from implementation (decouples *what* from *how*)

- *Ability to hide different implementations behind a common interface*

- *Ability of different objects to respond to the same message and invoke different responses*

- In some literature, polymorphism liked to *dynamic binding*, *late binding* or *run-time binding*

Taxonomy

Construct	Concept	Definition
Structure	**Abstraction**	Create classes to simply aspects of reality in problem domain
	Class	Description about types of objects (data + operations)
	Encapsulation	Hide internal structure & behavior to limit receivable messages
	Inheritance	One class is included in / used as the basis for another class
	Object	Identifiable item (real or abstract) containing data & operations
Behavior	**Message Passing**	Object sends data to another object /asks another object to invoke a method
	Method	A way to access, set or manipulate an object' information
	Polymorphism	Different classes respond to the same message and each implement it appropriately

OO Design

JACOBS UNIVERSITY

Two steps

- First find classes (abstraction); add methods to each class
- Find relationships between classes

Rule of thumb

- Classes *correspond to nouns* in the problem analysis; *methods correspond to verbs*

- Note that some classes are implicit

Example

In an order processing system, some of the nouns are

- **Item**, **Order**, **Shipping address**, **Payment** and **Account**

The nouns may lead to classes `Item`, `Order`, etc

Consider verbs

- Items are ***added*** to orders, Orders are ***shipped*** or ***cancelled***, payments are ***applied*** to orders

Identify which object has major responsibility to carry out the task

- Ex: new item is added to an order implies order object in charge
- `add` should be a method of `Order` class; it takes an `Item` as parameter

Relationships

The most common relationships between classes are

- Dependence (“`uses-a`”)
- Aggregation (“`has-a`”)
- Inheritance (“`is-a`”)

Relationships

JACOBS UNIVERSITY

Depends-on or “uses-a” relationship

- The most obvious & most general e.g., `Order` class uses the `Account` class
- ***A class depends on another class if its methods use or manipulate objects of that class***
- ***Goal is to minimize coupling between classes***

Aggregation or “has-a” relationship

- A very concrete relationship
- Ex: an `Order` object contains `Item` objects
- Containment means that objects of class `A` contain objects of class `B`

Relationships

JACOBS UNIVERSITY

Inheritance or “is-a” relationship

- Expresses the relationship between ***more specialized*** and ***more general class***

- Ex: an **`RushOrder`** class inherits from an **`Order`** class

- The specialized `RushOrder` class has special methods for *priority handling* and a *different method for computing charges*

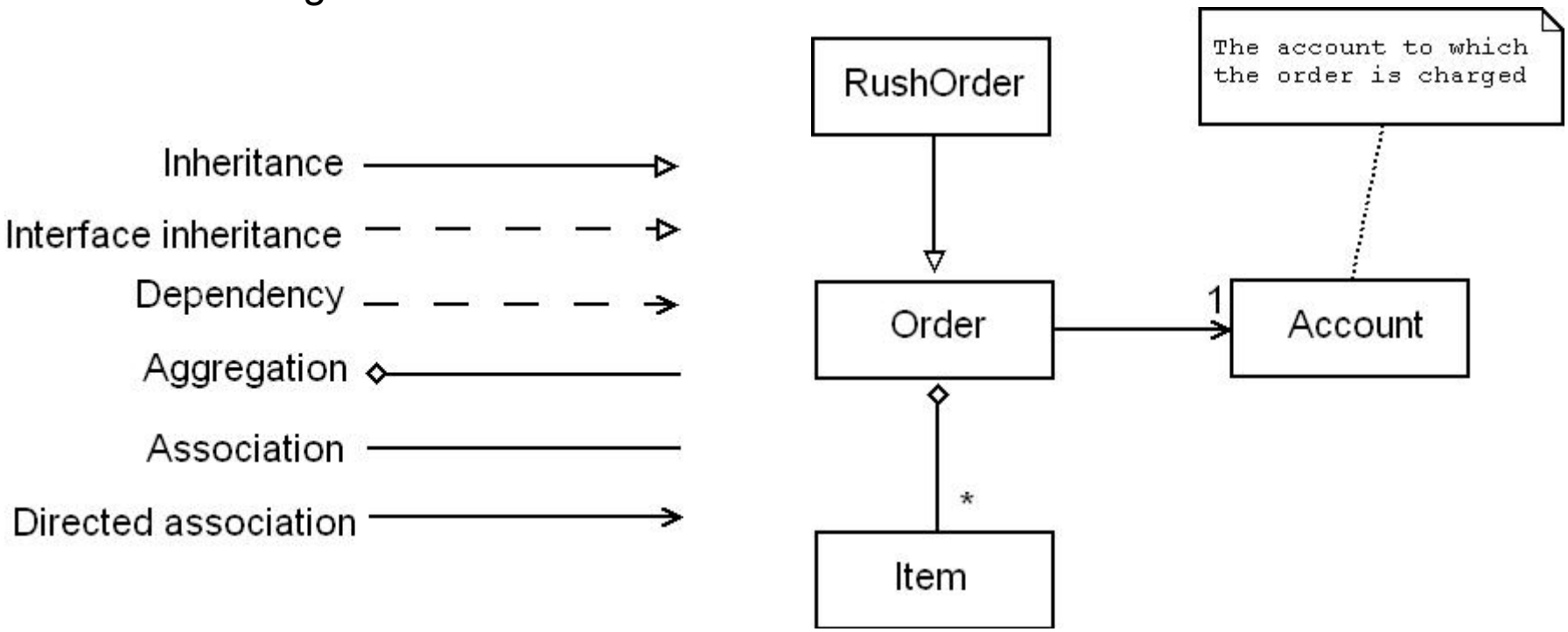
- Its other methods like *adding items* & *billing* are inherited from the `Order` class

- In general, if class `A` **extends** class `B`, class `A` inherits methods from class `B` but has more capabilities

Class Diagrams

UML (Unified modeling Language) is used to draw class diagrams

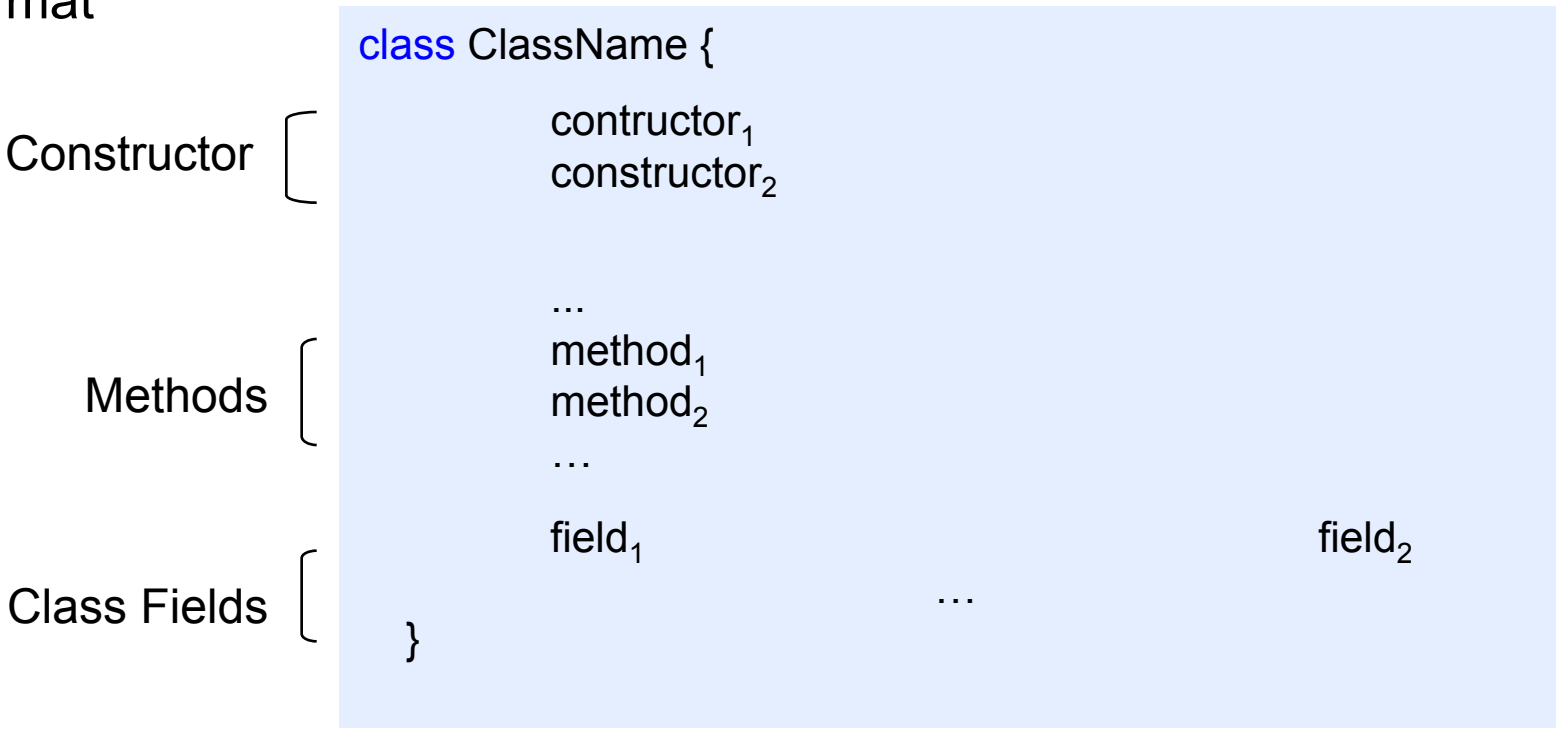
- Class diagram



Defining Classes

JACOBS UNIVERSITY

Format



- Use *constructors* to construct new objects from classes
- Constructor: special method to create & initialize objects
- Note that fields are declared outside method bodies

Example

Constructor

Methods

Class Fields

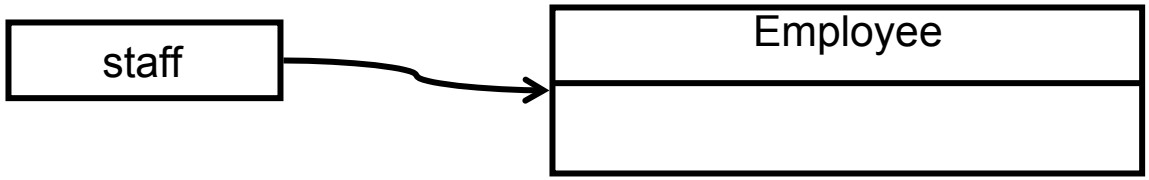
```
class Employee {

    // constructor

    public Employee (String n, double s, int year, int month, int day) {
            name = n;                                                    ….

    }    // end of constructor

  // a method
  public String getName() {
       return name;
   }
// more methods
  ……

  // instance fields
            private String name;
            private double salary;
            private Date hireDay;

}
```

How to Creating Objects

Objects are created from classes using the `new` operator

- The new operator allocates storage for the new object on the memory heap
- In the code segment, “staff” is a reference to the Employee object

```
Employee staff; // staff is a reference to an object of type Employee (initially null)
staff = new Employee(„Harry Hacker“, 2000, 2005, 7, 1); // creates the Employee object
```



Method Invocation

```
String name = staff.getName(); // This is how you call a method of the object
```

Example

JACOBS UNIVERSITY

public class
- has main method
- one per source file

nonpublic class
- 0 or more

```
// EmployeeTest.java
public class EmployeeTest { // only one public class in a source file
  public static void main() {…}
  …
}
 class Employee
{                                                                    //
constructor
 public Employee (String n, double s, int year, int month, int day)
{ …}

public String getName() {…}
public double getSalary() {…}
public Date getHireDay() {…}
public void raiseSalary(double byPercentage) {…}

 // instance fields
  private String name;
  private double salary;
  private Date hireDay;
}
```

Defining Classes

public methods
→ any method in any class can call these methods

private fields
→ no outside method can read or write these fields

```
class Employee
{                                                                    //
constructor
 public Employee (String n, double s, int year, int month, int day)
{ …}

public String getName() {…}
public double getSalary() {…}
public Date getHireDay() {…}
public void raiseSalary(double byPercentage) {…}

 // instance fields
  private String name;  // accessed by Employee class methods only
  private double salary; // accessed by Employee class methods only
  private Date hireDay; // accessed by Employee class methods only

}
```

Two fields are themselves object references : 'name' and 'hireDate'

- A class can have fields of class type

Constructors

```
 class Employee
{                                                                                    //
constructor
 public Employee (String n, double s, int year, int month, int day)   {
                name = n;
                salary = s;
                GregorianCalender calendar = new GregorianCalender(year, month-1, day);
                hireDay = calender.getTime();
  }
…
}
```

- A constructor method is called during object creation

```
Employee emp = new  Employee(„James Bond“, 100000, 1950, 1, 1);
```

- Sets instance fields for the Employee object referenced by *emp* as follows

```
name = „ James Bond“;
salary = 100000;
hireDay = January 1, 1950;
```

Constructors

JACOBS UNIVERSITY

Constructor Properties

- A constructor is always called with the `new` operator
- A constructor has the same name as its class
- A class can have more than one constructor
- A constructor takes zero or more parameters
- A constructor has no return value

Implicit Parameter (this)

Using `this`

```
 public void raiseSalary(double byPercent)
{                                                          double raise =
this.salary * byPercent / 100;
  this.salary += raise;
  }
```

- Helps to distinguish between local variables and instance fields

Static Fields and Methods

When an instance field is defined as `final`

- The field must be initialized during object construction
- The field may not be modified after object creation

- Example

```
class Employee
{
  …
  private final String name;
  }
```

- For mutable objects, final will not change the object reference, but the object itself is not constant

Static Fields and Methods

JACOBS UNIVERSITY

Static Fields

- There will be only one field per class
- Static fields belong to the class, not individual objects (class-wide)
- Example

```
class Employee
{
  …
  private int id;
  private static int nextId = 1;
  }
```

- Each employee object has its own *id* field
- Only one *nextId* field that's shared by all objects
- Called as follows

Employee.nextId

```
ClassName.staticFieldName;
Employee.nextId; // for Example
```

Static Fields and Methods

JACOBS UNIVERSITY

Constants

- Static constants are commonly used
- Static constants declaration preceded by `static final`
- Example

```
public class Math
{
  …
  public static final double PI = 3.14159;              }
```

- The constant is called like *Math.PI*
- Class fields should be declared private, but public constants are OK
- Other examples of public constants

```
public class System
{
  …
  public static final PrintStream out = …;              }
```

Static Fields and Methods

JACOBS UNIVERSITY

Static Methods

- Class methods that don't operate on objects
- Example

```
Math.pow(x, a) // x^a
```

- Does not use any Math object to carry its task
- Static methods don't operate on objects
  - ž Can't access instance fields from a static method
  - ž Static methods can access static fields within their class

```
public static int getNextId() {
        return nextId;   // return static fields
}
```

- Static methods are invoked by supplying the name of the class

```
int n = Employee.getNextId();
```

Parameter Passing

JACOBS UNIVERSITY

Call by Value

- Methods get the value that the caller provides
- A method cannot modify the stored value of the variable

Call by Reference

- Methods get the location of the variable that the caller provides
- A method can modify the stored value of the variable

The Java Programming Language always uses call by value

- A method gets a copy of all parameter values
- A method cannot modify the contents of parameter variables passed to it

Example

There are two kinds of parameters

- Primitive types (parameter values are not modified)
- Object references (parameter value can be modified)

```
double percent = 10;
harry.raiseSalary(percent);
```

- After a call to *`raiseSalary`*, the value of *`percent`* remains *10*
- *The method cannot change a primitive type parameter*

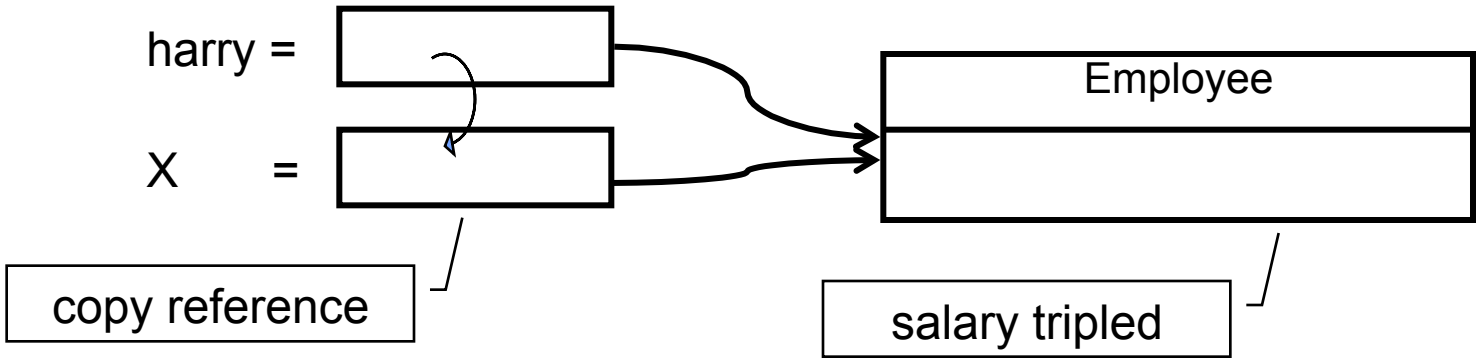
Example

```
public static void tripleSalary(Employee x) {

  x.raiseSalary(200);                                    }
```

Calling

```
harry = new Employee(…);
tripleSalary(harry);
```

- Result – the salary has been tripled in the original object



- **Object references are passed by value in Java**

Parameter Passing Summary

- Primitive parameters are not modified

- The state of an object parameter can be changed

- A method cannot make an object parameter refer to a new object

Class Initialization

Overloading

- Occurs if *several methods have the same name, but different parameters*
- The compiler distinguishes overloaded methods by their **signature**

- A method **signature** is a combination of:
  - A method's name
  - The number of parameters
  - The type of parameters and
  - The order of parameters

- A method call cannot be distinguished by *return types*

- In Java, any method (including constructors) can be overloaded

Example

The **String** class has four public methods called `indexOf`

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

Default Field Initialization

JACOBS UNIVERSITY

If a class provides at least one constructor, its illegal to construct objects without construction parameters

When no constructor is given, default values for fields are set

- Fields are automatically set to default values as follows
  - ❑ Numbers are set to 0
  - ❑ Boolean are set to false
  - ❑ Object references are set to null

Default Field Initialization

Default Constructors

- A constructor with no parameters

```
public Employee()
{
        name = „“;
        salary = 0;
        hireDay = new Date();

}
```

Explicit Initialization

Values are assigned to fields as part of the declaration

- Fields can be assigned values during the class definition

```
class Employee
{
        …
        private String name = „“;
}
```

- The assignment is carried out *before* the constructor executes

Initialization Blocks

This is a third way to initialize fields in a class

- Block of code that is executed whenever an object of a class is executed
- Example

```
class Employee
{
  public Employee (String n, double s, int year, int month, int day)   {
                                                                    …


  }
  public Employee() {} {…}
  private static int nextId;
  private int id;
  private String name;
  private double salary;

  // object initialization block
  {
            id = nextId;
            nextId++;
  }
}
```

Block runs first before any constructor

Parameter Names

Parameter Names

1. Prefix each parameter with an “a”

```
public  Employee (String aName, double aSalary)
{
        name = aName;
        salary = aSalary;
}
```

2. Use instance variables with the same name as corresponding field name, and this to distinguish them in method body

```
public  Employee (String name, double salary)
{
        this.name = name;
        this.salary = salary;
}
```

Object Destruction

There is *no explicit destructor methods to reclaim unused memory*

Java relies on **automatic garbage collection** to reclaim memory

However some objects use non-Java resources which must be reclaimed prior to garbage collection (e.g., **files**, **handles**, etc)

Object Destruction

The **`finalize`** method is invoked before the object is swept away and when the **VM** exits

```
protected void finalize()
{
        super.finalize();
                ...
}
```

- The finalize method **guarantees that certain actions are performed before an object is garbage collected** or before the VM exits

- *Objects that allocate external resources should provide a finalize method*

Example

```
public class ProcessFile {
        private Stream file;
        public ProcessFile (String path)   {
                file = new Stream(path);
        }

        public void close()  {
                if (file != null) {
                        file.close();
                        file = null;
                }
        }

        protected void finalize() throws Exception  {
                super.finalize();
                close();
        }
}
```

Class Design Hints

- Always keep data private

- Always initialize data

- Not all fields need individual field accessors and mutators

Class Design Hints

- Use standard form for class definitions

- Break-up classes with too many responsibilities

- Let names of classes and methods reflect their responsibilities
  - ❑ Names – nouns
  - ❑ Methods - verbs
    - accessor methods begin with get (e.g., *getSalary*)
    - Mutator methods begin with set (*setSalary*)

Reading Assignment

Horstmann, C. & Cornell, G. (2013) Core Java, Volume I-Fundamentals, 9th Edition. Prentice Hall. Chapter 4.