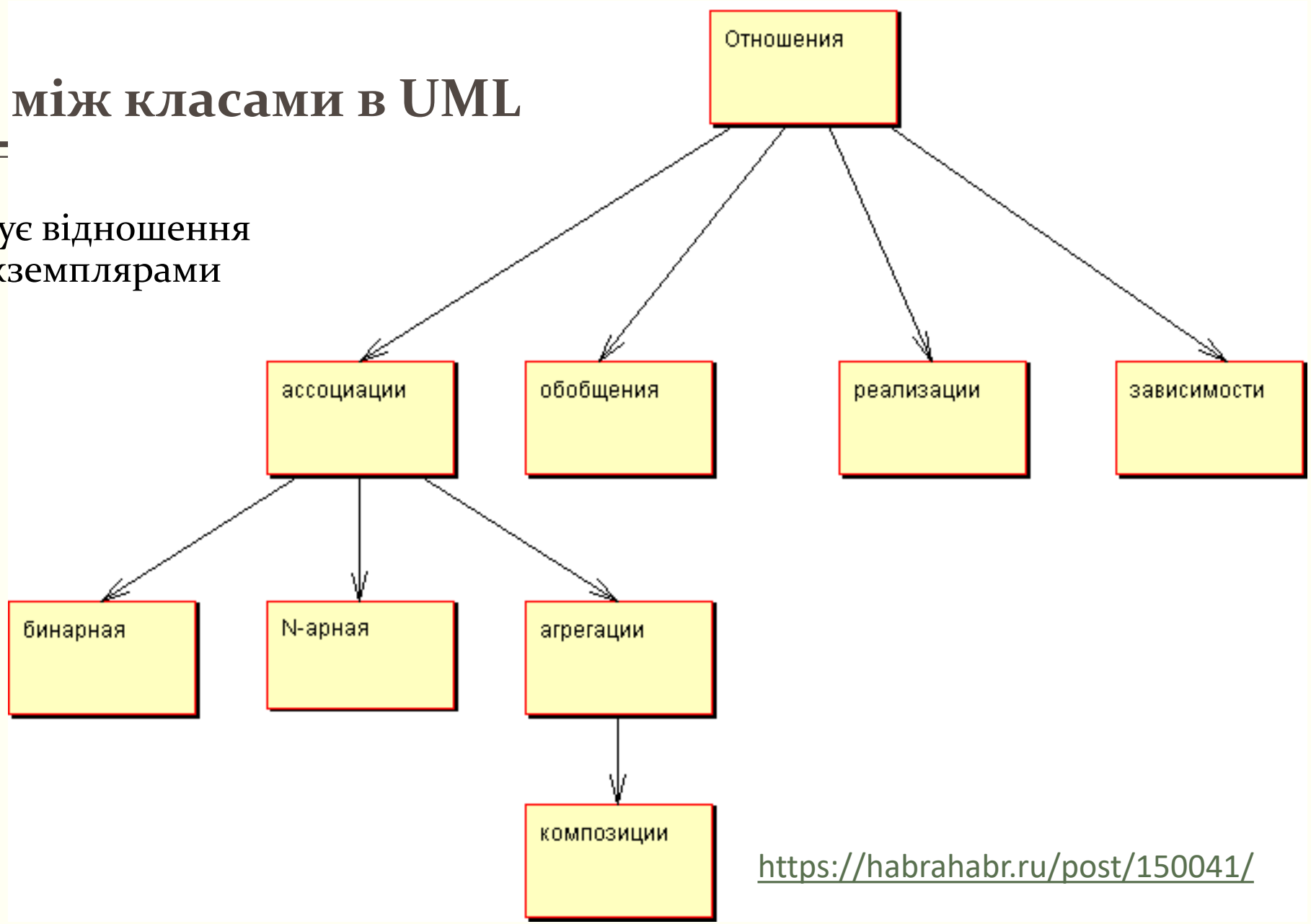
ІНШІ ВИДИ ВІДНОШЕНЬ МІЖ КЛАСАМИ

Питання 2.6.

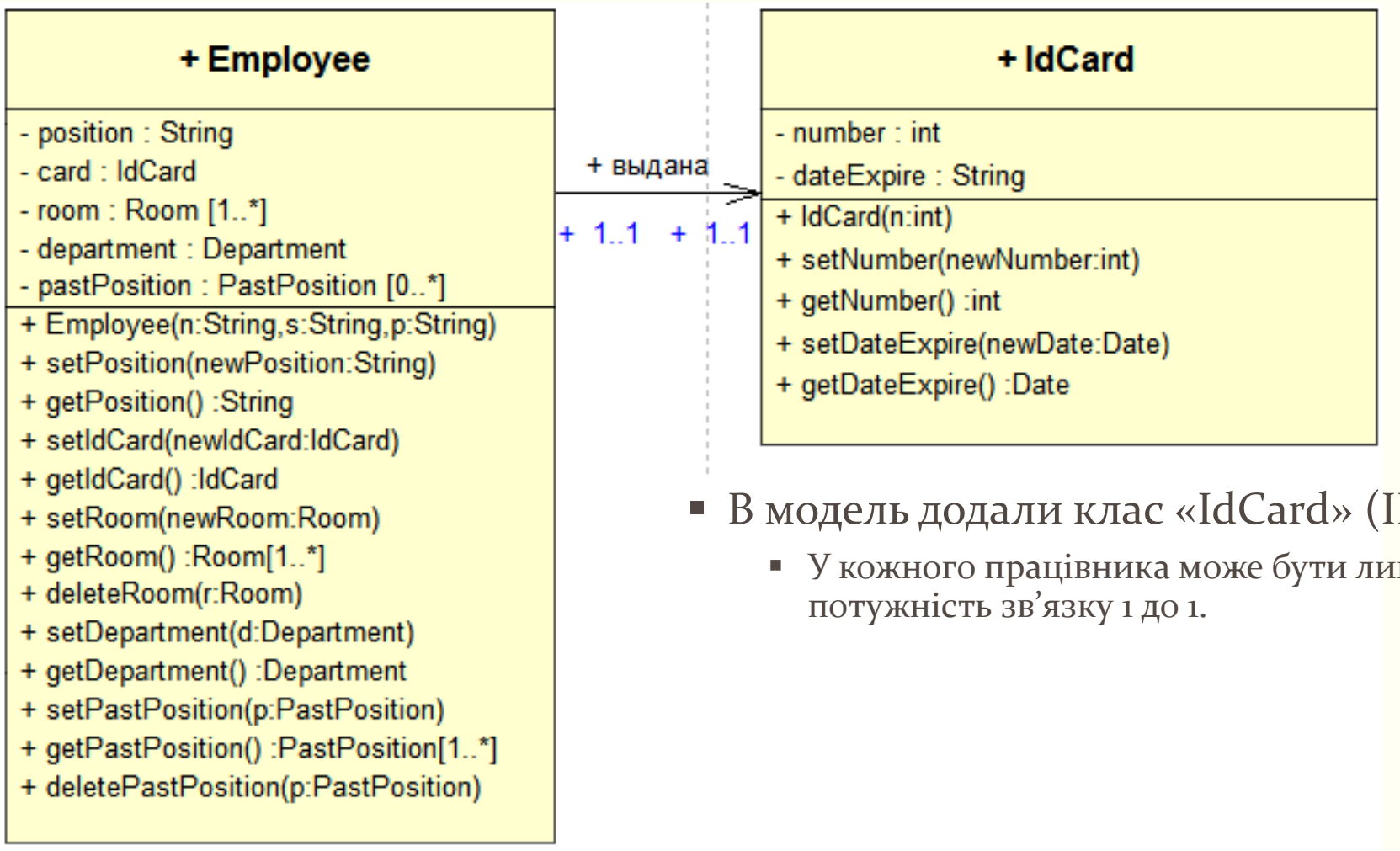
Відношення між класами в UML

- ***Асоціація*** показує відношення між об'єктами-екземплярами класу.



https://habrahabr.ru/post/150041/

Бінарна асоціація



- В модель додали клас «IdCard» (ID-картка працівника).
  - У кожного працівника може бути лише одна ID-картка, потужність зв'язку 1 до 1.

```
public class Employee extends Man{
    private String position;
    private IdCard iCard;
    public Employee(String n, String s, String p){
        name = n;
        surname = s;
        position = p;
    }
    public void setPosition(String newPosition){
        position = newPosition;
    }
    public String getPosition(){
        return position;
    }
    public void setIdCard(IdCard c){
        iCard = c;
    }
    public IdCard getIdCard(){
        return iCard;
    }
}
```

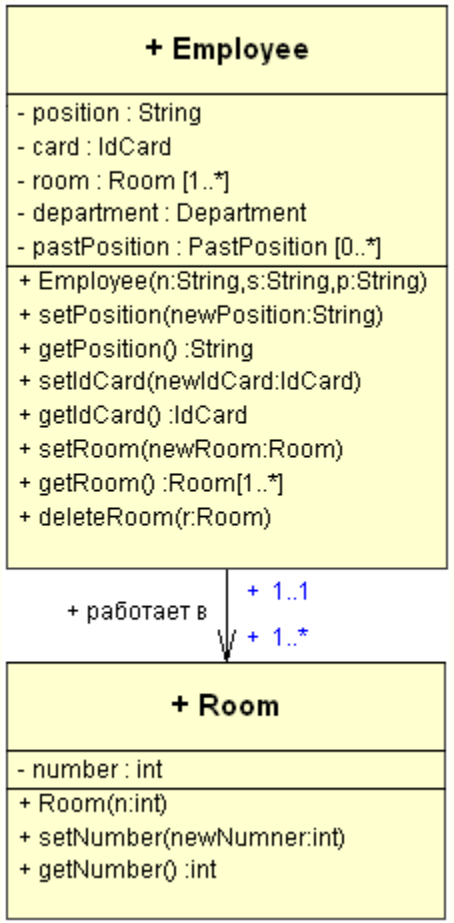
```
public class IdCard{
        private Date dateExpire;
        private int number;
        public IdCard(int n){
                number = n;
        }
        public void setNumber(int newNumber){
                number = newNumber;
        }
        public int getNumber(){
                return number;
        }
        public void setDateExpire(Date newDateExpire){
                dateExpire = newDateExpire;
        }
        public Date getDateExpire(){
                return dateExpire;
        }
}
```

- У тілі програми створюємо об’єкти та зв’язуємо їх:

```
IdCard card = new IdCard(123);
card.setDateExpire(new SimpleDateFormat("yyyy-MM-dd").parse("2015-12-31"));
sysEngineer.setIdCard(card);
System.out.println(sysEngineer.getName() +" работает в должности "+ sysEngineer.getPo
sition());
System.out.println("Удостовирение действует до " + new SimpleDateFormat("yyyy-MM-
dd").format(sysEngineer.getIdCard().getDateExpire()) );
```

  - З екземпляру об’єкта Employee можна дізнатись про зв’язаний з ним об’єкту типу IdCard, значит навигация (стрелочка на линии) направлена от Employee к IdCard.

N-арна асоціація



- Нехай в організації за робітниками закріпляються приміщення.
  - Додамо новий клас Room.

```
public  class Room{
      private int number;
      public Room(int n){
          number = n;
      }
      public void setNumber(int newNumber){
          number = newNumber;
      }
      public int getNumber(){
          return number;
      }
}
```

```
...
private Set room = new HashSet();
...
public void setRoom(Room newRoom){
    room.add(newRoom);
}
public Set getRoom(){
    return room;
}
public void deleteRoom(Room r){
    room.remove(r);
}
...
```

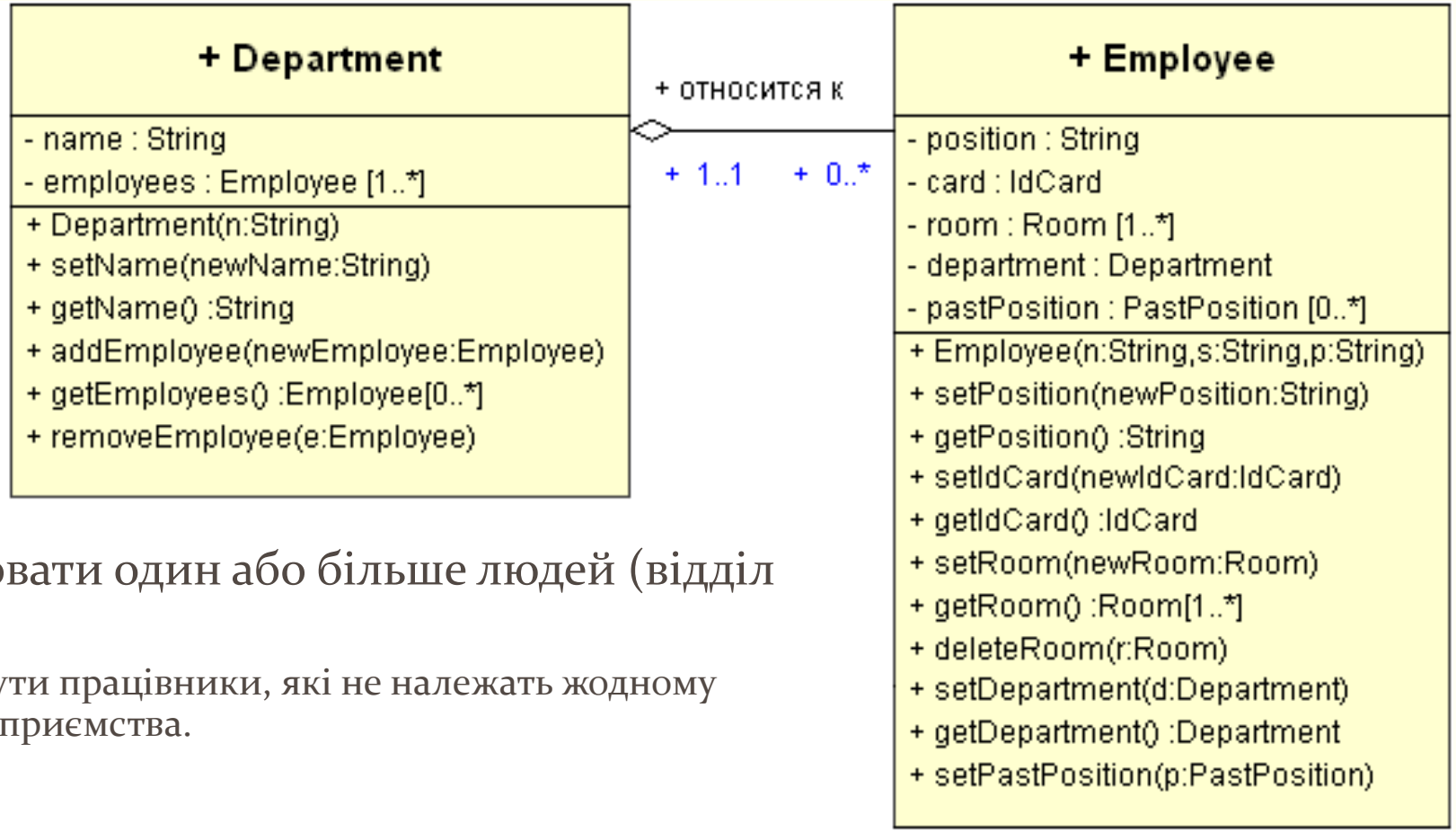
Кожному об’єкту робітник (Employee) може відповідати кілька робочих приміщень. Потужність зв’язку один-до-багатьох.

Використання класу

```
public static void main(String[] args){


    Employee sysEngineer = new Employee("John", "Connor", "Manager");
    IdCard card = new IdCard(123);
    card.setDateExpire(new SimpleDateFormat("yyyy-MM-dd").parse("2015-12-31"));
        sysEngineer.setIdCard(card);
        Room room101 = new Room(101);
    Room room321 = new Room(321);
    sysEngineer.setRoom(room101);
    sysEngineer.setRoom(room321);
    System.out.println(sysEngineer.getName() +" работает в должности "+ sysEngineer.g
etPosition());
    System.out.println("Удостовирение действует до " + sysEngineer.getIdCard().getDat
eExpire());
    System.out.println("Может находиться в помещеньях:");
    Iterator iter = sysEngineer.getRoom().iterator();
    while(iter.hasNext()){
         System.out.println( ((Room) iter.next()).getNumber());
        }
}
```

Агрегація

Введемо в модель клас Department (відділ – структурна одиниця підприємства)



- У кожному відділі може працювати один або більше людей (відділ включає, агрегує людей).
  - Також на підприємстві можуть бути працівники, які не належать жодному відділу, наприклад, директор підприємства.

```
public class Department{
    private String name;
    private Set employees = new HashSet();
    public Department(String n){
        name = n;
    }
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    public void addEmployee(Employee newEmployee){
        employees.add(newEmployee);
        // связываем сотрудника с этим отделом
        newEmployee.setDepartment(this);
    }
    public Set getEmployees(){
        return employees;
    }
    public void removeEmployee(Employee e){
        employees.remove(e);
    }
}
```

- Клас має методи для занесення у відділ нового працівника, видалення працівника та отримання всіх працівників відділу.
  - Від Department-об'єкта можна дізнатись про співробітника
  - Від Employee-об'єкта можна дізнатись, до якого відділу він відноситься.
- Для того, щоб дізнаватись, до якого відділу відноситься деякий працівник, додамо в клас поле і методи для призначення та отримання відділу.

```
private Department department;
...
public void setDepartment(Department d){
    department = d;
}
public Department getDepartment(){
    return department;
}
```

Робота з об’єктами класу

```
Department programmersDepartment = new Department("Программисты");

programmersDepartment.addEmployee(sysEngineer);

System.out.println("Относится к отделу "+sysEngineer.getDepartment().getName());
```

Композиція (Composition)

- Наслідування реалізації виконується розширенням класу за допомогою нового класу на основі відношення «is a».
  - Композиція передбачає компонування класів з інших класів, що базується на відношенні «has-a».
  - Наприклад, Car має Engine, Wheels, SteeringWheel (кермо).
  - Вже стикались з композицією: клас Car містить поля String make та String model

```
class Car extends Vehicle
{
   private Engine engine; // bicycles don't have engines
   private Wheel[] wheels; // boats don't have wheels
   private SteeringWheel steeringWheel; // hang gliders don't have steering wheels
}
```

  - Композиція та наслідування не взаємовиключні.
    - Хоч це і не показано, клас Car успадковує різні члени від суперкласу Vehicle, на додаток до постачання власних полів engine, wheels, steeringWheel.

Проблеми з наслідуванням реалізації

- Потенційно небезпечний механізм: руйнує інкапсуляцію.
  - зміни в суперкласі можуть «зламати» підклас, навіть не змінюючи його.
- Наприклад, придбано бібліотеку Java-класів, один з яких описує календар зустрічей.
  - Хоч доступу до коду немає, припустимо, що частина коду має вигляд

```
public class ApptCalendar
{
   private final static int MAX_APPT = 1000;
   private Appt[] appts;
   private int size;

   public ApptCalendar()
   {
      appts = new Appt[MAX_APPT];
      size = 0; // redundant because field automatically initialized to 0
                // adds clarity, however
   }
```

```
   public void addAppt(Appt appt)
   {
      if (size == appts.length)
         return; // array is full
      appts[size++] = appt;
   }
   public void addAppts(Appt[] appts)
   {
      for (int i = 0; i < appts.length; i++)
         addAppt(appts[i]);
   }
}
```

Хочемо додати лог зустрічей

```
public class LoggingApptCalendar extends ApptCalendar
{
   // A constructor is not necessary because the Java compiler will add
   // noargument constructor that calls the superclass's noargument
   // constructor by default.

   @Override
   public void addAppt(Appt appt)
   {
      Logger.log(appt.toString());
      super.addAppt(appt);
   }

   @Override
   public void addAppts(Appt[] appts)
   {
      for (int i = 0; i < appts.length; i++)
         Logger.log(appts[i].toString());
      super.addAppts(appts);
   }
}
```

- Припустимо, що Ви хочете вести лог кожної зустрічі (appointment), який зберігається у файл.
  - Необхідно розширити ApptCalendar, додаючи поведінку логування в переозначені методи addAppt() та addAppts()
  - Клас LoggingApptCalendar покладається на клас Logger, чий метод void log(String msg) логує рядок у файл (деталі неважливі).
  - Зауважте використання toString() для конвертації об'єкту Appt в об'єкт String, який потім передається методу log().
  - Клас працює не так, як Ви очікуєте!

Клас працює не так, як очікується

- Інстанціюємо клас, додавши кілька екземплярів Appt за допомогою методу addAppts()
  - **LoggingApptCalendar lapptc = new LoggingApptCalendar();**
  - **Lapptc.addAppts(new Appt[] { new Appt(), new Appt(), new Appt() });**
- Якщо також додати виклик System.out.println(msg); в метод log(String msg) класу Logger для виводу аргументів методу, Ви побачите, що log() виводить кожне з 3 очікуваних повідомлень (one per Appt object) двічі.
  - Коли викликається метод LoggingApptCalendar.addAppts(), перше звернення Logger.log() до кожного екземпляру Appt в масиві appts, який передається в addAppts().
  - Потім цей метод викликає метод addAppts() класу ApptCalendar.
  - Метод addAppts() класу ApptCalendar викликає переозначений метод addAppt() класу LoggingApptCalendar для кожного екземпляру класу Appt у масиві appts.
  - addAppt() виконує Logger.log(appt.toString()); з метою логування рядкового представлення його аргумента appt, а Ви закінчуєте з 3 зайвими повідомленнями.
  - Якщо не переозначати метод addAppts(), ця проблема зникне.
    - Проте підклас потрібно прив'язати до implementation detail: метод ApptCalendar.addAppts() викликає addAppt().

Проблеми наслідування

- Зміни в базовому класі можуть порушити роботу підкласу – **fragile base class problem** – одна з базових проблем ООП.
  - Споріднена причина крихкості, яка також пов'язана з переозначеними методами, виникає при додаванні нових методів у суперклас із наступним релізом.
  - Наприклад, у новій версії бібліотеки в клас ApptCalendar додали метод
  - new public void addAppt(Appt appt, boolean unique).
  - Він додає екземпляр класу (appt) в календар, коли unique = false; інакше додає appt лише за умови того, що appt не було додано раніше.
  - Оскільки цей метод було додано після створення класу LoggingApptCalendar, цей клас не переозначує новий метод addAppt() з викликом Logger.log().
  - У результаті екземпляри класу Appt, передані до new addAppt() method не логуються.
- Інша проблема: Ви представляєте метод у підкласі, якого немає в суперкласі.
  - Нова версія суперкласу теж представляє новий метод, який співпадає з ним за сигнатурою.
  - Ваш метод підкласу тепер переозначає метод суперкласу і, ймовірно, не виконує контракт методу суперкласу.
  - Усунути таку проблему неможливо.

Форвардинг (forwarding, обгортання)

- Замість розширення суперкласу створіть приватне поле в новому класі, яке міститиме посилання на екземпляр суперкласу.
  - Ця задача демонструє композицію, оскільки формує залежність “has-a” між новим класом та суперкласом.
  - Крім цього, маємо нові методи екземпляру, виклик відповідного методу суперкласу через збережений у приватному полі екземпляр суперкласу
  - Ця задача відома як forwarding, а нові методи називають forwarding methods.

Скомпонований клас Logging Appointment Calendar

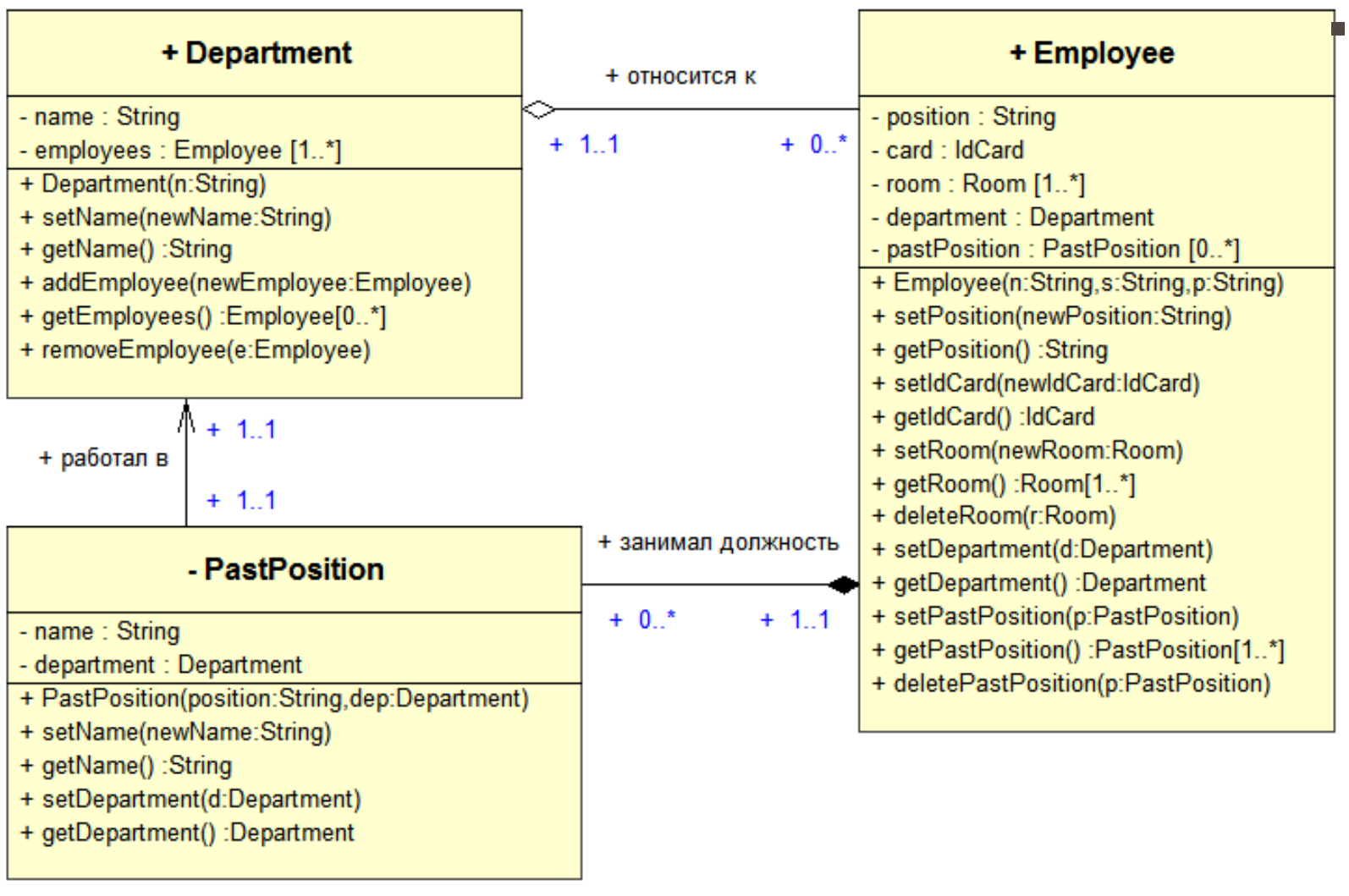
```
public class LoggingApptCalendar
{
   private ApptCalendar apptCal;

   public LoggingApptCalendar(ApptCalendar apptCal)
   {
      this.apptCal = apptCal;
   }

   public void addAppt(Appt appt)
   {
      Logger.log(appt.toString());
      apptCal.addAppt(appt);
   }

   public void addAppts(Appt[] appts)
   {
      for (int i = 0; i < appts.length; i++)
         Logger.log(appts[i].toString());
      apptCal.addAppts(appts);
   }
}
```

- Клас LoggingApptCalendar не залежить від деталей реалізації класу ApptCalendar.
- LoggingApptCalendar – приклад класу-обгортки (*wrapper class*).
- Кожен екземпляр класу LoggingApptCalendar обгортає екземпляр класу ApptCalendar.
- LoggingApptCalendar також є прикладом шаблону проектування Decorator

Коли розширювати клас, а коли його обгортати?

- Розширюйте клас, коли існує “is-a” залежність між суперкласом та підкласом, а також за умови наявності контролю над суперкласом або хорошого проектування та документації суперкласу для подальшого розширення.
  - Інакше використовуйте клас-обгортку.
  - “Design” означає постачання захищених методів, які втручаються у внутрішню роботу класу (для ефективного написання підкласів) та перевіряють, щоб конструктори та метод clone() ніколи не були переозначеними.
  - “Document” означає чітке прописування впливу переозначених методів.
- **Обережно!** Класи-обгортки не повинні використовуватись у callback-фреймворках – object-фреймворках, у яких об’єкт передає посилання на себе іншому об’єкту (this).
  - Останній об’єкт може викликати пізніше методи попереднього об’єкту.
  - Цей “calling back to the former object’s method” відомий як зворотний зв’язок (callback).
  - Оскільки обгорнутий клас не знає про свою обгортку, він передає лише посилання на себе, але отримані callback-и не включають методи класу-обгортки.

Композиція



- Припустимо, що до системи є вимога: зберігати інформацію про попередню посаду робітника на підприємстві.
  - Введемо новий клас **pastPosition** з полями name, department (пов’язує з класом Department).
  - Дані щодо попередніх посад є даними про співробітника, тому між ними є зв’язок «ціле-частина».
  - Також ці дані не можуть існувати без Employee-об’єкта.
  - Знищення Employee-об’єкта повинно призвести до знищення об’єктів pastPosition.

```
private class PastPosition{
    private String name;
     private Department department;
     public PastPosition(String position, Department dep){
         name = position;
         department = dep;
     }
     public void setName(String newName){
         name = newName;
     }
     public String getName(){
         return name;
     }
     public void setDepartment(Department d){
         department = d;
     }
     public Department getDepartment(){
         return department;
     }
}
```

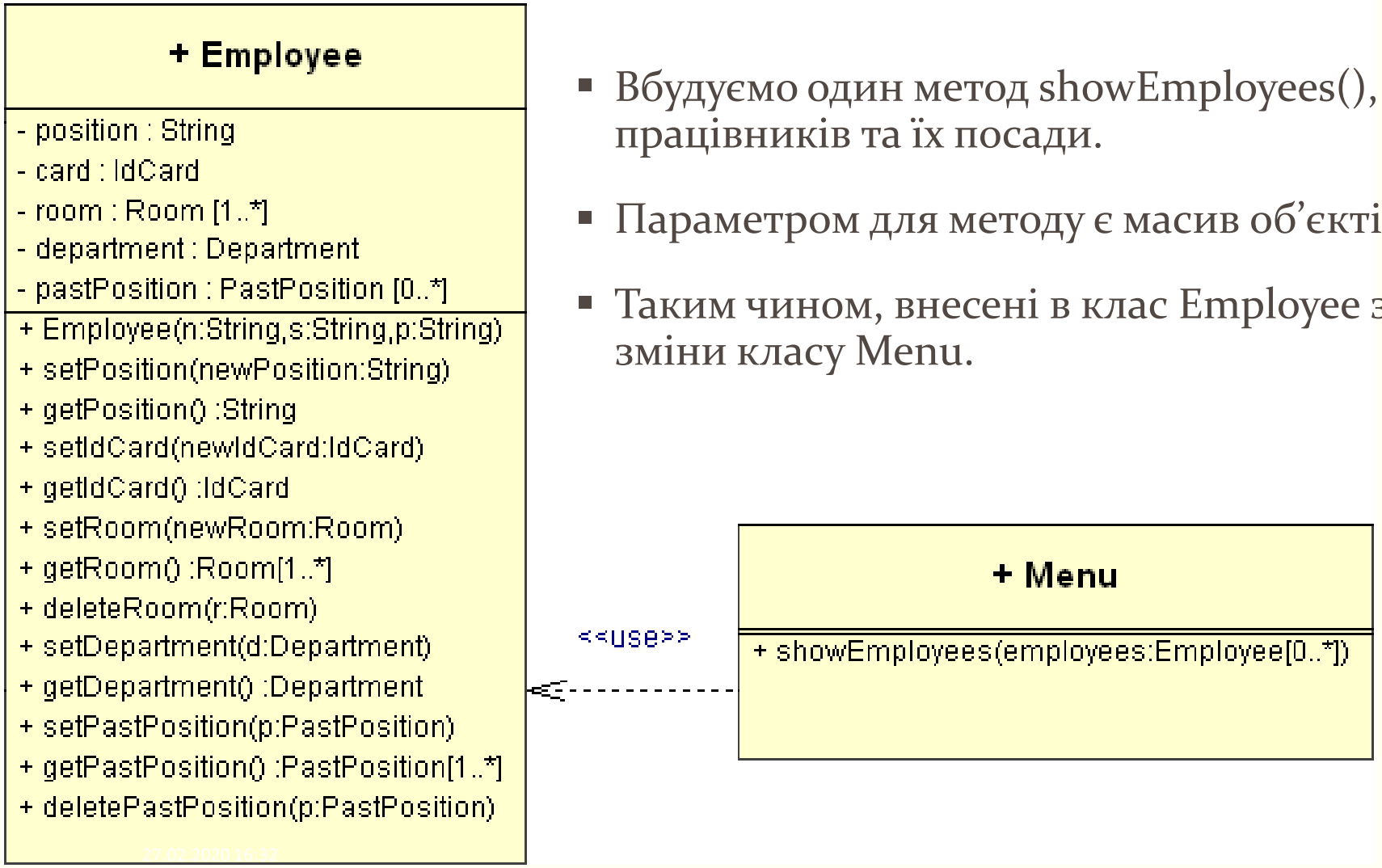
- У клас Employee додамо властивості та методи для роботи з даними щодо попередньої посади:

```
...
private Set pastPosition = new HashSet();
...
public void setPastPosition(PastPosition p){
     pastPosition.add(p);
}
public Set getPastPosition(){
     return pastPosition;
}
public void deletePastPosition(PastPosition p){
     pastPosition.remove(p);
}
...
```

Застосування

```
// изменяем должность
sysEngineer.setPosition("Сторож");
// смотрим ранее занимаемые должности:
System.out.println("В прошлом работал как:");
Iterator iter = sysEngineer.getPastPosition().iterator();
while(iter.hasNext()){
        System.out.println( ((PastPosition) iter.next()).getName());
}
```

Для організації діалогу з користувачем введемо в систему клас Menu



- Вбудуємо один метод showEmployees(), який показує список працівників та їх посади.
- Параметром для методу є масив об'єктів Employee.
- Таким чином, внесені в клас Employee зміни можуть вимагати і зміни класу Menu.

Клас Menu

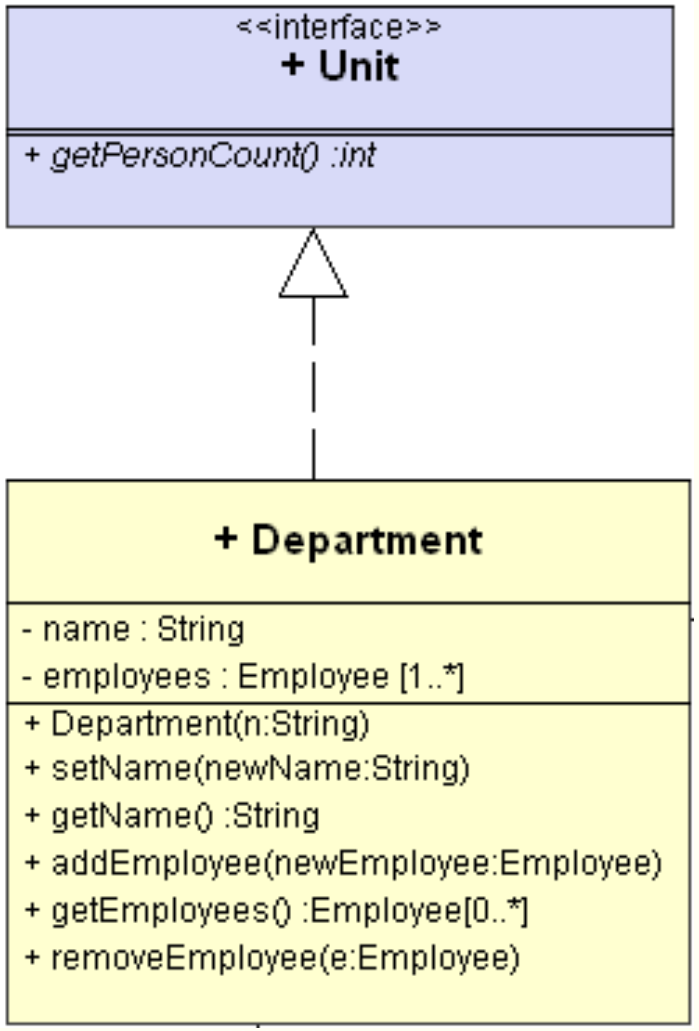
```
public  class Menu{
    private static int i=0;
    public static void showEmployees(Employee[] employees){
        System.out.println("Список сотрудников:");
        for (i=0; i<employees.length; i++){
            if(employees[i] instanceof Employee){
                System.out.println(employees[i].getName() +" - " + employees[i].getPosition());
                        }
                }
    }
}
```

Клас Menu не відноситься до жодної прикладної області, а представляє «системний» клас уявного додатку.

Використання:

```
// добавим еще одного сотрудника
Employee director = new Employee("Федор", "Дубов", "Директор");
Menu menu = new Menu();
Employee employees[] = new Employee[10];
employees[0]= sysEngineer;
employees[1] = director;
Menu.showEmployees(employees);
```

Реалізація



- Як і наслідування, має явне вираження в мові Java: оголошення інтерфейсу та його реалізація деяким класом.
- Для демонстрації створимо інтерфейс Unit.
  - Уявимо, що організація може ділитись не лише на відділи, а й, наприклад, на цехи, філіали тощо.
  - Інтерфейс «Unit» - найбільш абстрактна одиниця поділу.
  - У кожній одиниці працює певна кількість робітників, тому потрібен метод для отримання цієї кількості.
  - Він буде актуальним для кожного класу, який реалізує інтерфейс «Unit».

Реалізація

- Інтерфейс «Unit»:

```
public interface Unit{
     int getPersonCount();
}
```

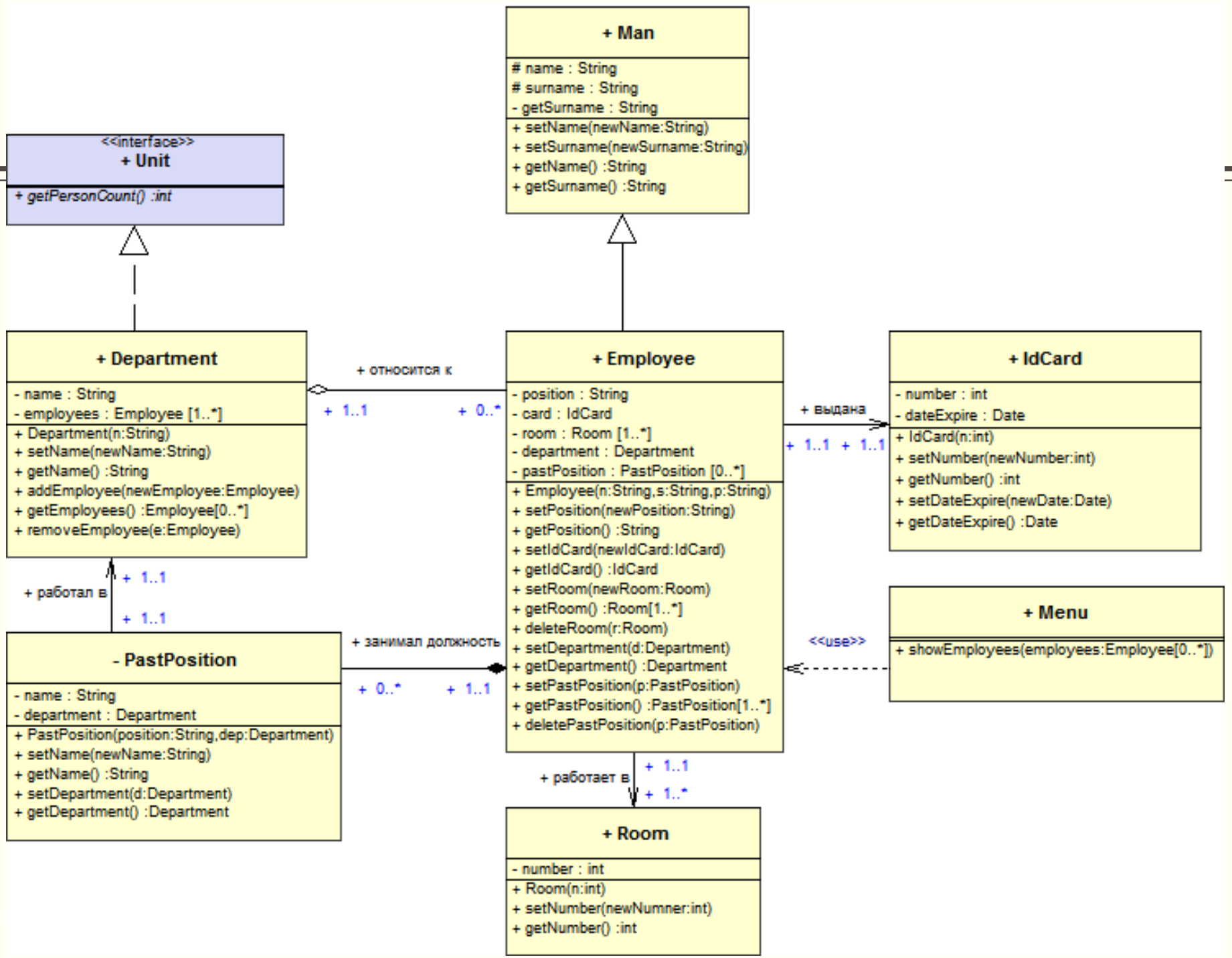
- Реалізація в класі Department:

```
public class Department implements Unit{
    ...
    public int getPersonCount(){
        return getEmployees().size();
    }
```

- Застосування:

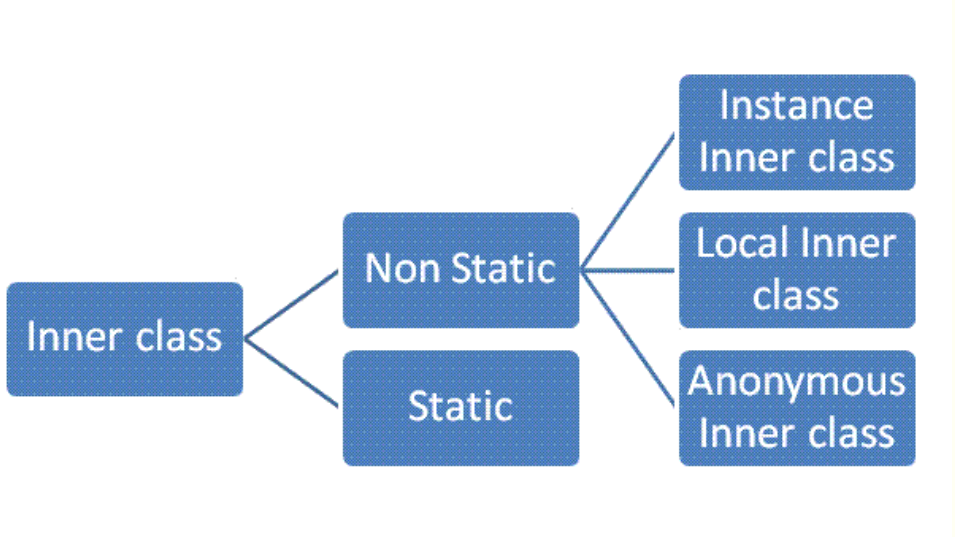
```
System.out.println("В отделе "+sysEngineer.getDepartment().getName()+" работает "
+sysEngineer.getDepartment().getPersonCount()+" человек.");
```

Реалізація методу getPersonCount() не зовсім актуальна для класу Department, оскільки він має метод getEmployees(), який повертає колекцію Employee-об’єктів.


+ Man
# name : String
# surname : String
- getSurname : String
+ setName(newName:String)
+ setSurname(newSurname:String)
+ getName() :String
+ getSurname() :String
<<interface>>
+ Unit
+ getPersonCount() :int
+ Department
- name : String
- employees : Employee [1..*]
+ Department(n:String)
+ setName(newName:String)
+ getName() :String
+ addEmployee(newEmployee:Employee)
+ getEmployees() :Employee[0..*]
+ removeEmployee(e:Employee)
+ относится к
+ 1..1
+ 0..*
+ Employee
- position : String
- card : IdCard
- room : Room [1..*]
- department : Department
- pastPosition : PastPosition [0..*]
+ Employee(n:String,s:String,p:String)
+ setPosition(newPosition:String)
+ getPosition() :String
+ setIdCard(newIdCard:IdCard)
+ getIdCard() :IdCard
+ setRoom(newRoom:Room)
+ getRoom() :Room[1..*]
+ deleteRoom(r:Room)
+ setDepartment(d:Department)
+ getDepartment() :Department
+ setPastPosition(p:PastPosition)
+ getPastPosition() :PastPosition[1..*]
+ deletePastPosition(p:PastPosition)
+ IdCard
- number : int
- dateExpire : Date
+ IdCard(n:int)
+ setNumber(newNumber:int)
+ getNumber() :int
+ setDateExpire(newDate:Date)
+ getDateExpire() :Date
+ выдана
+ 1..1
+ 1..1
+ работал в
+ 1..1
+ 1..1
- PastPosition
- name : String
- department : Department
+ PastPosition(position:String,dep:Department)
+ setName(newName:String)
+ getName() :String
+ setDepartment(d:Department)
+ getDepartment() :Department
+ занимал должность
+ 0..*
+ 1..1
<<use>>
+ Menu
+ showEmployees(employees:Employee[0..*])
+ работает в
+ 1..1
+ 1..*
+ Room
- number : int
+ Room(n:int)
+ setNumber(newNumner:int)
+ getNumber() :int

Вкладені типи

- Класи, що оголошуються зовні будь-якого класу, називають високорівневими (*top-level classes)*.
  - Також, які оголошуються в якості членів інших класів або scopes.
- Java підтримує *вкладені (nested) класи*
  - Вкладені класи допомагають реалізувати високорівневу архітектуру класів (top-level class architecture).
  - Існує 4 види вкладених класів:
    - Вкладені статичні класи (static member classes),
    - Вкладені нестатичні класи (nonstatic member classes),
    - Анонімні класи,
    - Локальні класи.
  - Останні три види відносять до *внутрішніх (inner classes)*.



```
class EnclosingClass
{
   private static int i;

   private static void m1()
   {
      System.out.println(i);
   }

   static void m2()
   {
      EnclosedClass.accessEnclosingClass();
   }

   static class EnclosedClass
   {
      static void accessEnclosingClass()
      {
         i = 1;
         m1();
      }

      void accessEnclosingClass2()
      {
         m2();
      }
   }
}
```

Статичні вкладені класи (Static Member Classes)

- Статичний вкладений клас є ***статичним членом зовнішнього*** (enclosing) ***класу***.
  - Він не має екземпляру зовнішнього класу і не має доступу до його полів та методів.
  - Проте може отримати доступ до статичних полів зовнішнього класу та викликати його статичні методи, навіть якщо вони оголошені приватними.
- Клас EnclosedClass оголошує метод класу accessEnclosingClass() та метод екземпляру accessEnclosingClass2().
  - Оскільки accessEnclosingClass() оголошується статичним, для виклику методу m2() перед ним треба дописувати «EnclosedClass.».

Виклик методів класу та екземпляру статичного вкладеного класу

- Метод main() показує, що перед викликом методу класу потрібно дописувати назву вкладеного (enclosed) класу разом з назвою зовнішнього класу, зокрема, EnclosingClass.EnclosedClass.accessEnclosingClass(); .
  - Потрібно дописувати аналогічні префікси при інстанціюванні вкладеного класу, наприклад,

    EnclosingClass.EnclosedClass ec = new EnclosingClass.EnclosedClass();
  - Після цього можна викликати метод екземпляру звичайним чином, зокрема, ec.accessEnclosingClass2(); .

```
public class SMCDemo
{
   public static void main(String[] args)
   {
      EnclosingClass.EnclosedClass.accessEnclosingClass(); // Output: 1
      EnclosingClass.EnclosedClass ec = new EnclosingClass.EnclosedClass();
      ec.accessEnclosingClass2(); // Output: 1
   }
}
```

Використання статичних вкладених класів для оголошення кількох реалізацій їх Enclosing Class

```
abstract class Rectangle
{
   abstract double getX();
   abstract double getY();
   abstract double getWidth();
   abstract double getHeight();

   static class Double extends Rectangle
   {
      private double x, y, width, height;

      Double(double x, double y, double width, double height)
      {
         this.x = x;
         this.y = y;
         this.width = width;
         this.height = height;
      }

      double getX() { return x; }
      double getY() { return y; }
      double getWidth() { return width; }
      double getHeight() { return height; }
   }
```

```
   static class Float extends Rectangle
   {
      private float x, y, width, height;

      Float(float x, float y, float width, float height)
      {
         this.x = x;
         this.y = y;
         this.width = width;
         this.height = height;
      }
      double getX() { return x; }
      double getY() { return y; }
      double getWidth() { return width; }
      double getHeight() { return height; }
   }

   // Prevent subclassing. Use the type-specific Double and Float
   // implementation subclass classes to instantiate.
   private Rectangle() {}

   boolean contains(double x, double y)
   {
      return (x >= getX() && x < getX() + getWidth()) &&
             (y >= getY() && y < getY() + getHeight());
   }
}
```

Створення та використання різних реалізацій Rectangle

```
public class SMCDemo
{
   public static void main(String[] args)
   {
      Rectangle r = new Rectangle.Double(10.0, 10.0, 20.0, 30.0);
      System.out.println("x = " + r.getX());
      System.out.println("y = " + r.getY());
      System.out.println("width = " + r.getWidth());
      System.out.println("height = " + r.getHeight());
      System.out.println("contains(15.0, 15.0) = " + r.contains(15.0, 15.0));
      System.out.println("contains(0.0, 0.0) = " + r.contains(0.0, 0.0));
      System.out.println();
      r = new Rectangle.Float(10.0f, 10.0f, 20.0f, 30.0f);
      System.out.println("x = " + r.getX());
      System.out.println("y = " + r.getY());
      System.out.println("width = " + r.getWidth());
      System.out.println("height = " + r.getHeight());
      System.out.println("contains(15.0, 15.0) = " + r.contains(15.0, 15.0));
      System.out.println("contains(0.0, 0.0) = " + r.contains(0.0, 0.0));
   }
}
```

```
x = 10.0
y = 10.0
width = 20.0
height = 30.0
contains(15.0, 15.0) = true
contains(0.0, 0.0) = false

x = 10.0
y = 10.0
width = 20.0
height = 30.0
contains(15.0, 15.0) = true
contains(0.0, 0.0) = false
```

Бібліотека класів Java містить багато вкладених статичних класів

- Клас java.lang.Character є зовнішнім для вкладеного статичного класу Subset, чиї екземпляри представляють підмножини символів Unicode.
  - Також наявні приклади в java.util.AbstractMap.SimpleEntry та java.io.ObjectInputStream.GetField.
- **Зауважте!** Коли компілюєте enclosing class, що містить static member class, компілятор створює class-файл для static member class, назва якого складається з назви його зовнішнього класу, знаку $ та назви статичного вкладеного класу (static member class).
  - Наприклад, при компіляції попереднього лістингу будуть створені файли EnclosingClass$EnclosedClass.class та EnclosingClass.class.
  - Цей формат застосовується і до нестатичних вкладених класів (nonstatic member classes).

Нестатичні вкладені класи (Nonstatic Member Classes)

```
class EnclosingClass
{
   private int i;

   private void m()
   {
      System.out.println(i);
   }

   class EnclosedClass
   {
      void accessEnclosingClass()
      {
         i = 1;
         m();
      }
   }
}
```

- Нестатичний вкладений клас є нестатичним членом зовнішнього класу.
  - Кожен екземпляр нестатичного вкладеного класу неявно зв'язаний з екземпляром зовнішнього класу.
  - Методи екземпляру нестатичного вкладеного класу можуть викликати методи екземпляру в зовнішньому класі та отримувати доступ до нестатичних полів екземплярів зовнішнього класу.

```
public class NSMCDemo
{
   public static void main(String[] args)
   {
      EnclosingClass ec = new EnclosingClass();
      ec.new EnclosedClass().accessEnclosingClass(); // Output: 1
   }
}
```

- Посилання ec використовується як префікс для оператора new, який інстанціює EnclosedClass, а отримане від нього посилання береться для виклику accessEnclosingClass(), що виводить 1.
  - Додавання префіксу (з посиланням на зовнішній клас) зустрічається рідко.
  - Замість цього зазвичай викликають конструктор вкладеного класу з конструктора або методу екземпляра зовнішнього класу.

Реалізація To-Do Items як пар «назва-опис»

```
class ToDo
{
   private String name;
   private String desc;

   ToDo(String name, String desc)
   {
      this.name = name;
      this.desc = desc;
   }

   String getName()
   {
      return name;
   }

   String getDesc()
   {
      return desc;
   }

   @Override
   public String toString()
   {
      return "Name = " + getName() + ", Desc = " + getDesc();
   }
}
```

- Припустимо, потрібно тримати to-do list, в якому кожен пункт складається з назви та опису.
  - Створимо клас ToDoList для збереження екземплярів ToDo-пунктів.
  - Список формується на базі нестатичного вкладеного класу ToDoArray, який зберігає екземпляри ToDo-пунктів у growable array;
  - Невідомо, скільки екземплярів буде зберігатись, а Java-масиви мають фіксований розмір.

Зберігання максимуму 2 екземплярів ToDo в екземплярі ToDoArray

```
class ToDoList
{
   private ToDoArray toDoArray;
   private int index = 0;

   ToDoList()
   {
      toDoArray = new ToDoArray(2);
   }

   boolean hasMoreElements()
   {
      return index < toDoArray.size();
   }

   ToDo nextElement()
   {
      return toDoArray.get(index++);
   }

   void add(ToDo item)
   {
      toDoArray.add(item);
   }
```

Разом з постачанням методу add() для зберігання екземплярів ToDo в екземплярі масиву ToDoArray, ToDoList забезпечує методи hasMoreElements() та nextElement() для ітерування по збережених екземплярах та їх повернення.

```
   private class ToDoArray
   {
      private ToDo[] toDoArray;
      private int index = 0;

      ToDoArray(int initSize)
      {
         toDoArray = new ToDo[initSize];
      }

      void add(ToDo item)
      {
         if (index >= toDoArray.length)
         {
            ToDo[] temp = new ToDo[toDoArray.length*2];
            for (int i = 0; i < toDoArray.length; i++)
               temp[i] = toDoArray[i];
            toDoArray = temp;
         }
         toDoArray[index++] = item;
      }

      ToDo get(int i)
      {
         return toDoArray[i];
      }

      int size()
      {
         return index;
      }
   }
}
```

Створення та ітерування по списку ToDoList з екземплярів ToDo

```
public class NSMCDemo
{
   public static void main(String[] args)
   {
      ToDoList toDoList = new ToDoList();
      toDoList.add(new ToDo("#1", "Do laundry."));
      toDoList.add(new ToDo("#2", "Buy groceries."));
      toDoList.add(new ToDo("#3", "Vacuum apartment."));
      toDoList.add(new ToDo("#4", "Write report."));
      toDoList.add(new ToDo("#5", "Wash car."));
      while (toDoList.hasMoreElements())
         System.out.println(toDoList.nextElement());
   }
}
```

- Вивід:

```
Name = #1, Desc = Do laundry.
Name = #2, Desc = Buy groceries.
Name = #3, Desc = Vacuum apartment.
Name = #4, Desc = Write report.
Name = #5, Desc = Wash car.
```

- Бібліотека Java-класів має багато прикладів нестатичних вкладених класів.
  - Наприклад, клас HashMap оголошує приватні класи HashIterator, ValueIterator, KeyIterator та EntryIterator для ітерування по значеннях, ключах та entries хешмепів.
- Код всередині вкладеного класу може отримувати посилання на свій зовнішній клас, використовуючи «.this».
  - Наприклад, якщо код в accessEnclosingClass() потребував би посилання на екземпляр EnclosingClass, було б задано EnclosingClass.this

Анонімні класи

```
abstract class Speaker
{
   abstract void speak();
}

public class ACDemo
{
   public static void main(final String[] args)
   {
      new Speaker()
      {
         String msg = (args.length == 1) ? args[0] : "nothing to say";

        @Override
         void speak()
         {
            System.out.println(msg);
         }
      }
      .speak();
   }
}
```

- Анонімним називають клас без імені.
  - Він не є членом зовнішнього класу,
  - оголошується одночасно з ним як анонімне розширення або анонімна реалізація інтерфейсу
  - Інстанціюється в будь-якому місці, де можна задавати вираз

Екземпляри анонімного класу мають мати доступ до локальних змінних та параметрів з оточуючої області видимості.

- Проте an instance might outlive the method in which it was conceived (as a result of storing the instance's reference in a field), and try to access local variables and parameters that no longer exist after the method returns.

```
interface Speakable
{
   void speak();
}

public class ACDemo
{
   public static void main(final String[] args)
   {
      new Speakable()
      {
         String msg = (args.length == 1) ? args[0] : "nothing to say";

         @Override
         public void speak()
         {
            System.out.println(msg);
         }
      }
      .speak();
   }
}
```

- екземпляру анонімного класу дозволяється доступ лише до локальних змінних та параметрів, які оголошені як final.
  - Java не може дозволити такий нелегальний доступ, який, ймовірно, призведе віртуальну машину до краху.
- Розглядаючи назви константних локальних змінних або параметрів в екземплярі анонімного класу, компілятор виконує дві речі:
  - Якщо змінна примітивного типу, компілятор замінює its name на read-only значення змінної.
  - Якщо змінна посилкового типу, компілятор додає в class-файлі *синтетичну* змінну *та посилання на* local variable/parameter в синтетичній змінній.

Анонімні класи

- Хоч анонімний клас не має конструктора, можна створити instance initializer для роботи зі складною ініціалізацією.
  - Наприклад, new Office() {{addEmployee(new Employee("John Doe"));}};
    інстанціює анонімний підклас of Office та додає один об'єкт типу Employee в цей екземпляр, викликаючи метод Office.addEmployee().
- Анонімні класи – поширена мовна конструкція.
  - Наприклад, потрібно повернути список *.java-файлів.
  - Анонімний клас спрощує використання класів File та FilenameFilter з пакету java.io при вирішенні задачі:

```
String[] list = new File(directory).list(new FilenameFilter()
                {
                   @Override
                   public boolean accept(File f, String s)
                   {
                      return s.endsWith(".java");
                   }
                });
```

Локальні класи

```
class EnclosingClass
{
   void m(final int x)
   {
      final int y = x * 2;
      class LocalClass
      {
         int a = x;
         int b = y;
      }
      LocalClass lc = new LocalClass();
      System.out.println(lc.a);
      System.out.println(lc.b);
   }
}
```

```
public class LCDemo
{
   public static void main(String[] args)
   {
      EnclosingClass ec = new EnclosingClass();
      ec.m(10);
   }
}
```

- Це класи, які оголошено будь-де в якості локальної змінної.
  - Має ту ж область видимості.
  - На відміну від анонімних класів, можна повторно використовувати.
  - Аналогічно до анонімних класів, мають лише enclosing instances, коли використовуються в нестатичному контексті.
- Доступ до екземпляру локального класу може відбуватись з локальних змінних та параметрів оточуючої області видимості.
  - Проте локальні змінні та параметри, до яких отримують доступ, повинні бути оголошені як final.
- Локальний клас оголошує пару полів екземпляру (a і b), які ініціалізуються значеннями константного параметру x та локальної константи y при інстанціюванні LocalClass:
  - Наприклад, new EnclosingClass().m(10);

```
10
20
```

```
class ToDoList
{
   private ToDo[] toDoList;
   private int index = 0;

   ToDoList(int size)
   {
      toDoList = new ToDo[size];
   }

   Iterator iterator()
   {
      class Iter implements Iterator
      {
         int index = 0;

         @Override
         public boolean hasMoreElements()
         {
            return index < toDoList.length;
         }

         @Override
         public Object nextElement()
         {
            return toDoList[index++];
         }
      }
      return new Iter();
   }

   void add(ToDo item)
   {
      toDoList[index++] = item;
   }
}
```

Інтерфейс Iterator та клас ToDoList

- Локальні класи допомагають покращити зрозумілість коду, оскільки їх можна перемістити ближче до місця їх використання.
  - Наприклад, у лістингу оголошено інтерфейс Iterator та клас ToDoList, чий метод iterator() повертає екземпляр локального класу Iter як екземпляр Iterator (оскільки Iter реалізує Iterator).

```
interface Iterator
{
   boolean hasMoreElements();
   Object nextElement();
}
```

Створення та ітерування по ToDoList екземплярів ToDo з повторно використовуваним ітератором

```
public class LCDemo
{
   public static void main(String[] args)
   {
      ToDoList toDoList = new ToDoList(5);
      toDoList.add(new ToDo("#1", "Do laundry."));
      toDoList.add(new ToDo("#2", "Buy groceries."));
      toDoList.add(new ToDo("#3", "Vacuum apartment."));
      toDoList.add(new ToDo("#4", "Write report."));
      toDoList.add(new ToDo("#5", "Wash car."));
      Iterator iter = toDoList.iterator();
      while (iter.hasMoreElements())
         System.out.println(iter.nextElement());
   }
}
```

- Екземпляр класу Iterator, що повертається від iterator(), містить елементи (items) ToDo в порядку їх додавання до списку.
- Хоч використовувати повернений Iterator-об'єкт можна лише раз, виклик iterator() можливий будь-де, де потрібен новий Iterator-об'єкт.
  - Це значне покращення в порівнянні з минулою реалізацією.

```
public class InnerLeakDemo
{
   public static void main(String[] args)
   {
      class Outer
      {
         String s = "outer string";

         class Inner
         {
            String s = "inner string";

            void print()
            {
               System.out.println(s);
               System.out.println(Outer.this.s);
            }
         }
      }

      Outer o = new Outer();
      Outer.Inner oi = o.new Inner();
      oi.print();
   }
}
```

Внутрішні класи та витоки пам’яті

- Екземпляри внутрішніх класів містять неявні посилання на їх зовнішні класи.
  - Пролонгування існування екземпляру внутрішнього класу (наприклад, зберігання його посилання в статичній змінній) може призвести до витоку пам’яті.
  - Екземпляр зовнішнього класу може посилатись на великий граф об’єктів, що не можна зібрати збирачем сміття через тривале існуванння посилання на внутрішній клас.
- Метод main() оголошує локальний клас Outer, який оголошує нестатичний вкладений клас Inner.
  - І Outer, і Inner оголошують поле екземпляру String s, а Inner також вносить метод print().
  - Далі метод main() інстанціює цей локальний клас, а потім – його вкладений клас Inner.
  - Вивід показує, що поле s класу Outer існує стільки ж, скільки і посилання на клас Inner.
  - Це приклад витоку пам’яті.

```
inner string
outer string
```

ДЯКУЮ ЗА УВАГУ!

Наступна тема: поліморфізм, інтерфейси та узагальнене програмування в Java

Інтерфейси всередині класів та класи всередині інтерфейсів

```
class X
{
   interface A
   {
   }

   static interface B
   {
   }
}
```

- Інтерфейси можуть вкладатись у класи.
  - Вони будуть статичними, навіть без ключового слова static.
  - Доступ до таких інтерфейсів стандартний.
- Наприклад, задамо
  - class C implements X.A {} або
  - class D implements X.B {}.
- Як і з вкладеними класами, вкладені інтерфейси допомагають реалізувати високорівневу архітектуру шляхом впровадження вкладених класів.
  - Collectively, these types are nested because they cannot (as in Listing 5-14's Iter local class) or need not appear at the same level as a top-level class and pollute its package namespace.
  - The java.util.Map.Entry interface and HashMap class is a good example.

Tightly Binding a Class to an Interface

```
public interface Addressable
{
   class Address
   {
      private String boxNumber;
      private String street;
      private String city;

      public Address(String boxNumber, String street, String city)
      {
         this.boxNumber = boxNumber;
         this.street = street;
         this.city = city;
      }

      public String getBoxNumber()
      {
         return boxNumber;
      }

      public String getStreet()
      {
         return street;
      }

      public String getCity()
      {
         return city;
      }

      @Override
      public String toString()
      {
         return boxNumber+" - "+street+" - "+city;
      }
   }

   Address getAddress();
}
```

- Classes can be nested within interfaces. Once declared, a class is considered to be static even when it is not declared static.
- Although nowhere near as common as nesting an interface within a class, nested classes have a potential use,
- Listing 5-18 declares an Addressable interface that describes any entity associated with an address, such as a letter, parcel, or postcard.
- This interface declares an Address class to store the address components, and an Address getAddress() method that returns the address.

```
Addressable[] addressables =
{
   new Letter(new Addressable.Address("10", "AnyStreet", "AnyTown")),
   new Parcel(new Addressable.Address("20", "Doe Street", "NewTown")),
   new Postcard(new Addressable.Address("30", "Ender Avenue", "AnyCity"))
};

for (int i = 0; i < addressables.length; i++)
   System.out.println(addressables[i].getAddress());
```

- Assuming the existence of Letter, Parcel, and Postcard classes whose constructors take Address arguments, the following code fragment shows you how you could construct an array of addressables and then iterate over it, obtaining and printing each address.

- Notice that you must specify Addressable.Address to access the nested Address class.
  - Можна використати статичний імпорт для спрощення коду.
- Навіщо вкладати Address замість того, щоб зробити його високорівневим класом?
  - Основна ідея – існування тісного зв'язку між класами Addressable та Address, який потрібно охопити.
  - Також Ви можете вже мати інший високорівневий клас з іменем Address та хочете запобігти конфлікту імен.
- Вкладення класу в інтерфейс багатьма вважається поганою практикою, оскільки це йде врозріз з ідеями ООП та notion інтерфейсу.
  - Проте краще знати про цю можливість, оскільки вона може знадобитись на практиці.