جامعة المنارة
MANARA UNIVERSITY

الجلسة الأولى - برمجة 3

الغاية من الجلسة: مقدمة في البرمجة غرضية التوجه JAVA من خلال مثال بسيط مع بعض الأفكار المهمة

مقدمة في لغة الجافا:

لغة الجافا هي لغة برمجة قوية وشائعة الاستخدام تستخدم في تطوير مجموعة واسعة من التطبيقات، بدءاً من تطبيقات الويب وحتى تطبيقات سطح المكتب وتطبيقات الأجهزة المحمولة. تم تطوير لغة الجافا لأول مرة من قبل شركة سان ميكروسيستمز (Sun Microsystems) في عام 1995، ومنذ ذلك الحين أصبحت واحدة من أكثر اللغات شعبية في العالم.

من أهم مميزات لغة الجافا:

1. المحمولية (Portability): تعمل تطبيقات الجافا على مختلف الأنظمة والمنصات بما في ذلك ويندوز، لينكس، وماك، مما يجعلها مناسبة لتطوير تطبيقات متعددة الأنظمة.

2. الأمان (Security): لغة الجافا مصممة لتكون آمنة، حيث تمتلك ميزات تأمينية مثل إدارة الذاكرة وتنفيذ الكود في بيئة معزولة تماماً (JVM) مما يمنع الوصول غير المصرح به للموارد الحساسة.

3. الكفاءة (Performance): بفضل تحسيناتها المستمرة، أصبحت لغة الجافا أكثر كفاءة في التنفيذ عبر السنوات، ولكن قد تكون أقل كفاءة بعض الشيء مقارنة ببعض اللغات الأخرى.

4. المجتمع القوي (Strong Community): تتميز لغة الجافا بمجتمع تطوير قوي ونشط، مما يعني وجود العديد من المكتبات والأدوات المفيدة والموارد التعليمية المتاحة لمساعدة المطورين.

المثال الآتي يظهر استخدام لغة الجافا لإنشاء صف يمثل المستطيل ويوفر طرقاً لحساب المساحة والمحيط ومقارنة بين مستطيلين. هذا المثال يوضح القدرة على إنشاء كائنات مخصصة (في هذه الحالة: المستطيل) وتنفيذ العديد من العمليات عليها باستخدام الطرق والتحكم في الوصول إلى بياناتها باستخدام الطرق العامة والخاصة (getters و setters).

في البداية، سننشئ صفاً يسمى Rectangle يحتوي على واصفات طول وعرض للمستطيل وطرقاً لتعيين والحصول على قيم هذه الواصفات، وأيضًا طرقاً لحساب المساحة والمحيط، بالإضافة إلى طريقة لمقارنة بين مستطيلين وإعادة المستطيل الأكبر. ثم سنستخدم هذه الصف في الطريقة الرئيسية (main).

```
class Rectangle{
```

نعرف واصفتين خاصتين private أحدهما تعبر عن الطول والأخرى عن العرض.

```
  private double length;
  private double width;
```

نئشئ الباني الافتراضي وتاخذ فيه الواصفات قيماً صفرية أما في حال كانت منطقية فتاخذ القيمة false وفي حال كانت مراجعاً (أغراضاً) فتاخذ القيمة null أي لا شيء:

```
//   Default Constructor
  public Rectangle(){
```

```
    // Width is 0 and Length is 0
  }
```

ننشئ الباني بوسطاء:

```
//   Constructor
  public Rectangle(double length, double width){
    this.length = length;
    this.width = width;
  }
```

ننشئ الطرق setLength و setWidth التي تعطي قيم للواصفات التي هي الطول والعرض:

```
//   Setters
  public void setLength(double length) {
    this.length = length;
  }

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

ننشئ طرق الـ getLength و getWidth التي تعيد قيم الواصفات التي هي الطول والعرض:

```
//   Getters
  public double getLength(){
    return length;
  }

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

الطريقة الآتية من أجل حساب مساحة المستطيل والتي هي الطول*العرض:

```
//   Calculate area
  public double calculateArea(){
    return length * width;
  }
```

الطريقة الآتية من اجل حساب محيط المستطيل (ضعفي مجموع الطول والعرض):

```
//   Calculate perimeter
  public double calculatePerimeter(){
    return 2 * (length + width);
  }
```

الطريقة الآتية هي من أجل مقارنة مستطيلين من حيث المساحة وتعيد المستطيل الأكبر وبالتالي النمط المعاد من الطريقة compareRegtangles هو نمط المستطيل أي Rectangle:

```
//   Compare rectangles and return the larger one
  public Rectangle compareRectangles(Rectangle rect1) {
    double area = calculateArea();
    double area1 = rect1.calculateArea();
```

حسبنا مساحة المستطيل الأول وهو الذي سيتدعي الطريقة أي أنه الغرض الحالي this وخزناها في area وحسبنا مساحة المستطيل الثاني rect1 وخزناها في area1.

```
    if (area1 > area) {
      return rect1;
    } else {
      return this;
}
}
}

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

هنا ننشئ مستطيلاً أولاً rect1 حيث طوله 10 وعرضه 5 ومستطيلاً ثانياً rect2 طوله 6 وعرضه 8.

```
//     Create rectangles
    Rectangle rect1 = new Rectangle(10 ,5);
    Rectangle rect2 = new Rectangle(6 ,8);
```

وهنا في السطرين التاليين نطبع معلومات المستطيل الأول ونطبع معلومات المستطل الثاني:

```
//     Print dimensions
    System.out.println("Rectangle 1 dimensions: Length = " + rect1.getLength() + ", Width = " + rect1.getWidth());
    System.out.println("Rectangle 2 dimensions: Length = " + rect2.getLength() + ", Width = " + rect2.getWidth());
```

نطبع في السطرين التاليين مساحة المستطيل الأول rect1.calculateArea() ومساحة المستطيل الثاني rect2.calculateArea():

```
//     Calculate and print areas
    System.out.println("Area of Rectangle 1: " + rect1.calculateArea());
    System.out.println("Area of Rectangle 2: " + rect2.calculateArea());
```

هنا نطبع محيط الأول والثاني:

```
//     Calculate and print perimeters
    System.out.println("Perimeter of Rectangle 1: " + rect1.calculatePerimeter());
    System.out.println("Perimeter of Rectangle 2: " + rect2.calculatePerimeter());
```

في الأسطر التالية نقارن بين المستطيل rect1 والمستطيل rect2 من خلال استدعاء الطريقة compareRectangles ثم نطبع معلومات المستطيل الأكبر الذي تم تخزينه في largerRectangle:

```
//     Compare rectangles and print the larger one
    Rectangle largerRectangle = rect1.compareRectangles(rect2);
    System.out.println("The larger rectangle is: Length = " + largerRectangle.getLength() + ", Width = " +
largerRectangle.getWidth());
  }}
```

التمرير بالقيمة:

في لغة الجافا، عندما تمرر قيمة من نوع primitive (مثل `int`، `double`، `char`، إلخ) فأنت في الواقع تمرر نسخة من القيمة المخزنة في المتغير. هذا يعني أن أي تعديلات تتم على المعامل داخل الدالة لا تؤثر على القيمة الأصلية المخزنة في الدالة التي يحدث فيها الاستدعاء.

إليك كيفية عمل التمرير بالقيمة:

1. نسخة من القيمة: عندما تمرر نوع primitive، مثل `int` أو `double`، فأنت تمرر نسخة من القيمة الفعلية المخزنة في المتغير.

2. المعاملات لا تتغير: بما أنك تعمل مع نسخ من القيم، فإن أي تعديلات تتم على المعاملات داخل الدالة تُطبق فقط على تلك النسخ. تظل القيم الأصلية المخزنة في الدالة التي يحدث فيها الاستدعاء دون تغيير.

التمرير بالمرجع:

في لغة الجافا، عندما تمرر مرجعاً لكائن (غرضاً) كوسيط لدالة، فإن التعديلات التي تتم على خصائص الكائن داخل الدالة تُعكس على الكائن الأصلي أي أن التعديل يتم على مكان وجود القيم في الذاكرة.

هنا مثال بسيط لتوضيح فكرة التمرير بالقيمة:

```java
package javaapplication16;

class A{
    int x;
    A(int xx){
        x = xx;
    }
    public void modify(int y){
        y = y + 1;
        System.out.println("The value in the modify method: " + y);

    }
}
public class NewClass {
    public static void main(String[] args) {
        A a = new A(2);
        int value = 5;
        System.out.println("The value before modify method: " + value);
        a.modify(value);
        System.out.println("The value after modify method: " + value);


    }

}
```

خرج البرنامج السابق هو:

```
Output - JavaApplication16 (run)
run:
The value before modify method: 5
The value in the modify method: 6
The value after modify method: 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

المثال التالي هو لتوضيح التمرير بالمرجع:

```java
package javaapplication16;

class A{
    private int x;
    public A(){

    }
    public A(int xx){
        x = xx;
    }
    public int getX(){
        return x;
    }
    public void modifyObject(A a){
        a.x = a.x + 1;
       System.out.println("The attribute x of a1 in modifyObject: " + a.x);
    }
}
public class NewClass {
    public static void main(String[] args) {
       A a = new A();
       A a1 = new A(5);
       System.out.println("The attribute x of a1 before modifyObject: " + a1.getX());
       a.modifyObject(a1);
       System.out.println("The attribute x of a1 after modifyObject: " + a1.getX());


    }
```

خرج البرنامج السابق:

```
Output - JavaApplication16 (run)
run:
The attribute x of a1 before modifyObject: 5
The attribute x of a1 in modifyObject: 6
The attribute x of a1 after modifyObject: 6
BUILD SUCCESSFUL (total time: 0 seconds)
```

إذاً لاحظ أن الغرض a1 قبل الاستدعاء modifyObject ليس كما هو بعد الاستدعاء بالتالي التغيير الذي طرأ عليه في الطريقة modifyObject قد انتقل إلى الطريقة الأساسية main.

المثال التالي يوضح فكرة طريقة تعيد غرضاً، حيث لدينا موظف Employee وله اسم name وراتب salary وباني بوسطاء بالإضافة لـ getters و setters وطريقة تجمع راتبي موظفين:

```
//Class to represent an employee
class Employee {
  private String name;
  private double salary;

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

//   Getter and setter methods
  public String getName(){
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public double getSalary(){
    return salary;
  }

  public void setSalary(double salary) {
    this.salary = salary;
  }

//   Method to add salary of another employee and return the same object
  public Employee addSalary(Employee other) {
    this.salary += other.getSalary();
    return this;
  }
}
```

```
public class Main {
  public static void main(String[] args) {
//      Creating employees
    Employee emp1 = new Employee("Youssef", 3000);
    Employee emp2 = new Employee("Ahmed", 2500);

//      Calling addSalary method and printing the same object
    Employee totalEmployee = emp1.addSalary(emp2);

//      Printing the total salary
    System.out.println("Total salary: " + totalEmployee.getSalary());
  }
}
```

فيكون الخرج:

```
Output - JavaApplication16 (run)
run:
Total salary: 5500.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

اختبار تساوي غرضين:

لو أردنا اختبار تساوي الغرضين من صف الـ Employee اللذان عرفناهما في الطريقة main السابقة:

هل يمكننا أن نكتب: if (emp1 == emp2) ؟؟

إن المعامل == يستخدم للإشارة إلى أن الغرضين هما نفسهما تماماً أي أنهما يشيران إلى نفس الموقع في الذاكرة وبالتالي عندما نكتب الشرط الآتي: if (emp1 == emp2) نقصد به هل الغرضان يشيران إلى نفس الموقع في الذاكرة؟ بالتالي سيعيد false حتى لو كان محتوى الغرضين هو نفسه أي حتى لو كان الاسم والراتب في الغرض الأول هما نفسهما الاسم والعمر في الغرض الثاني.

بالتالي متى تعطي == قيمة true؟

عندما يشير الغرضان إلى نفس الموقع في الذاكرة كأن نسند غرض أول إلى آخر، أي مثلاً لو كتبنا:

```
public class Main {
    public static void main(String[] args) {
        // Creating employees
        Employee emp1 = new Employee("AA", 3000);
        Employee emp2 = emp1;


        System.out.println(emp1==emp2);
    }
}
```

عندئذ سيكون الخرج هو: true لأنهما يشيران إلى نفس الموقع في الذاكرة.

الآن لو أردنا أن نختبر أن غرضين يحملان نفس المعلومات ماذا نستخدم؟

نستخدم الطريقة equals الموجودة اساساً في الجافا بعد أن نقوم بتجاوزها override ونكتبها كما نريد.

بمعنى أننا لو استدعينا الطريقة equals في الطريقة main كما يلي:

```
public class Main {
    public static void main(String[] args) {
        // Creating employees
        Employee emp1 = new Employee("AA", 3000);
        Employee emp2 = new Employee("AA", 3000);

        System.out.println(emp1.equals(emp2));
    }
}
```

الخرج سيكون false، لماذا؟ لأن طريقة الـ equals التي استدعيناها هي موجودة في الجافا وتختبر تساوي غرضين من خلال المعامل == والذي يختبر إذا كان الغرضان يشيران إلى نفس الموقع في الذاكرة!

بالتالي يجب أن نعيد تعريف الطريقة equals (أي نتجاوزها override) بالشكل الذي نريد في صف الـ Employee بالشكل:

```
public boolean equals(Employee e){
    if (this.name == e.name && this.salary == e.salary)
        return true;
    else
        return false;
}
```

وبالتالي صار بإمكاننا أن نستخدم الطريقة equals في الـ main:

```
public class Main {
    public static void main(String[] args) {
        // Creating employees
        Employee emp1 = new Employee("AA", 4000);
        Employee emp2 = new Employee("AA", 3000);

        System.out.println(emp1.equals(emp2));
    }
}
```

سيكون الخرج للكود السابق هو false لأن فعلاً الغرضين مختلفان من حيث الاسم والراتب، بينما لو كتبنا:

```
public class Main {
    public static void main(String[] args) {
        // Creating employees
        Employee emp1 = new Employee("AA", 4000);
        Employee emp2 = new Employee("AA", 4000);

        System.out.println(emp1.equals(emp2));
    }
}
```

سيكون الخرج هنا هو true لأنهما يحويان نفس المعلومات.

النسخ العميق Deep Copy:

في الحالة العادية عندما تسند غرضاً إلى غرض، أو تمرر غرضاً في طريقة بالتالي أي تعديل على هذا الغرض سوف يؤثر على الغرض الأساسي لكن أحياناً تحتاج إلى أن تنسخ غرضاً دون أن تؤثر على الغرض الأساسي، كيف ذلك؟ من خلال النسخ العميق، إليك مثال:

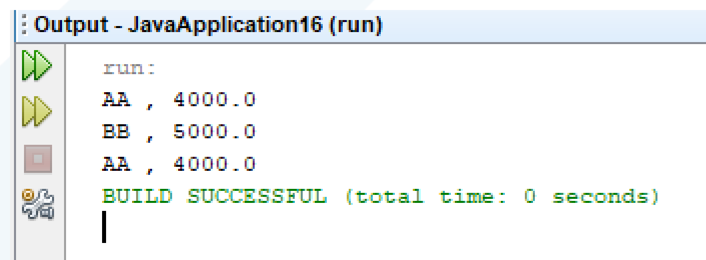
نضيف الطريقة الآتية إلى صف الموظف:

```
public Employee deepCopy(Employee e) {
    Employee e1 = new Employee(e.name, e.salary);
    return e1;
}
```

ونختبر في الـ main:

```
public class Main {
    public static void main(String[] args) {
        Employee emp1 = new Employee("AA", 4000);
        System.out.println(emp1.getName() + " , " + emp1.getSalary());

        Employee emp2 = emp1.deepCopy(emp1);

        emp1.setName("BB"); emp1.setSalary(5000);

        System.out.println(emp1.getName() + " , " + emp1.getSalary());

        System.out.println(emp2.getName() + " , " + emp2.getSalary());


    }
}
```

بالتالي سيكون الخرج:



بينما لو وضعنا في الـ main:

```
Employee emp2 = emp1;
```

وقمنا بالتعديل على emp1 عندئذ سينتقل التعديل إلى emp2 وكذلك لو عدلنا emp2 سيتأثر أيضاً emp1.