

# اليوم التدريبي السادس

الجلسة	الموضوع التدريبي	الزمن
الأولى	مقدمة إلى أسلوب البرمجة الكائنية.	١٠٠ ق
الثانية	مبادئ البرمجة الكائنية. - التجريد abstraction. - التغليف encapsulation. - التوريث inheritance.	١٠٠ ق
الثالثة	مبادئ البرمجة الكائنية. - التعددية polymorphism.	١٠٠ ق

# الجلسة الأولى

## ■ مقدمة إلى أسلوب البرمجة الكائنية object-oriented programming

صممت لغة الجافا لتكون لغة برمجة شيئية-object-oriented programming language.

○ تعتمد على التجريد abstraction والتغليف inheritance والتعددية polymorphism لغرض توفير قدرأ كبيراً من المرونة وإعادة الإستخدام أثناء تطوير البرمجيات.

# مقدمة

■ نستعرض في هذا اليوم أسلوب البرمجة الكائنية بشكل منهجي ونتعرف على أهم المفاهيم التي يعتمد عليها.

# الأصناف والكائنات classes & objects

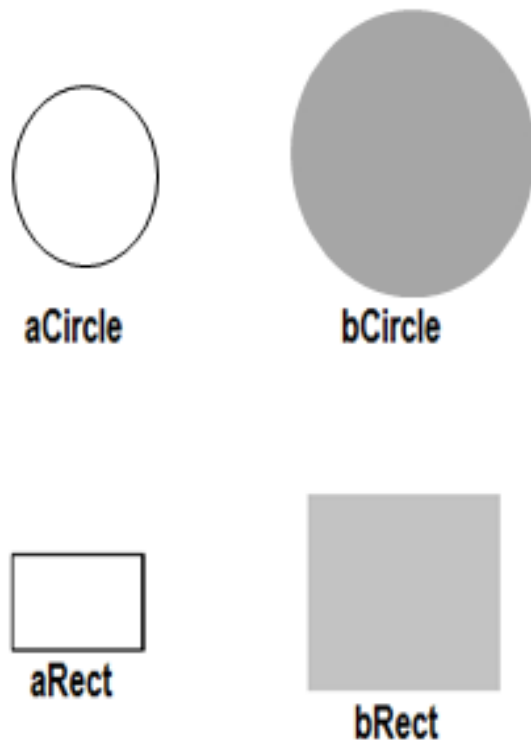
- يعتمد أسلوب البرمجة الكائنية إلى تجزئة البرنامج إلى وحدات برمجية مستقلة ومتفاعلة.
- يتم تمثيل تلك الأجزاء البرمجية من خلال إنشاء الكائنات القادرة على إنجاز مهمة أو مهام معينة من خلال سلوكياتها وتفاعلاتها مع بعضها البعض.
- الكائن object هو حالة محددة instance من حالات الصنف.

```

public class ShapeDemo {
    public static void main(String [] args) {
        Circle aCircle = new Circle();
        Circle bCircle = new Circle(2, "grey");
        Rectangular aRect = new Rectangular();
        Rectangular bRect = new Rectangular(2,3,"grey");
    }
}

```

### إنشاء الكائنات



```

public class Circle {
    private double radius, area;
    private String color;
    public Circle() {
        radius = 1.0; area = 0.0; color = "white";
    }
    public Circle(double val, String str) {
        radius = val; color = str;
        area = radius * radius * Math.PI;
    }
    public double getArea() {
        return area;
    }
}

```

### الأصناف

```

public class Rectangular {
    private int width, height;
    private String color;
    public Rectangular() {
        width = 2; height = 1; color = "white";
    }
    public Rectangular(int val1, int val2, String str) {
        width = val1; height = val; color = str;
    }
}

```

شكل (١-١): يستخدم الصنف class لإنشاء الكائنات objects. والكائن هو حالة محددة من الصنف instance of a class.

# الأصناف والكائنات: أكثر من هياكل بيانات

- يمثل الصنف class البنية الأساسية للبرمجة الكائنية.
- يستخدم لإنشاء الكائنات المتفاعلة لأداء مهام النظام.

# عناصر الصنف: الخصائص والسلوك attributes and behavior

■ يتألف الصنف في البرنامج الشيئي من:

(١) متغيرات variables:

■ تمثل خصائص attributes الكائن.

■ كما تمثل مجموعة المتغيرات الحالة الداخلية state للكائن.



# عناصر الصنف: الخصائص والسلوك attributes and behavior

■ يتألف الصنف في البرنامج الشيئي من:

(٢) طرق methods:

■ تعمل على المتغيرات، مع شرط ضمان سلامة الحالة الداخلية state للكائن.

■ تمثل سلوك behavior الصنف.

مثلا..

```
public class Circle {  
    private double radius, area;  
    private String color;  
    public Circle() {  
        radius = 1.0; area = 0.0; color = "white"; }  
    public Circle(double val, String str) {  
        radius = val; color = str;  
        area = radius * radius * Math.PI; }  
  
    public double getArea() {  
        return area; }  
}
```

■ يحتوي الصنف Circle على

- متغير radius وهي الخاصية التي تمثل كائن الدائرة.
- تقوم طريقة احتساب المساحة getArea() بتمثيل سلوكه.

# عناصر الصنف: المتغيرات والطرق

■ تُعرف مجموعة المتغيرات `variables` والطرق `methods` التي يحتويها الصنف بعناصر الصنف `class members`.

■ يتم التعامل مع عناصر الصنف من خلال إنشاء الكائنات وتنفيذ العمليات (أي الطرق) عليها.

```
Circle bCircle;  
bCircle = new Circle(2, "grey");  
System.out.println("area of bCircle: "  
                    + bCircle.getArea() );
```

# أسلوب البرمجة الكائنية

■ تعريف أصناف النظام المختلفة بحيث:

- تحتوي على خصائص محددة attributes وتقوم بتنفيذ سلوك معين behavior.
- تُطبق مبدأ حماية البيانات من خلال تعريف بيانات محمية private ومغلقة encapsulated بالطرق العامة public للمحافظة على الحالة الداخلية state للصنف.

# محددات الوصول access modifiers

■ تحدد إمكانية الوصول إلى عناصر الصنف:

○ public لجعل العنصر متاح للإستخدام من اي صنف آخر.

○ private لجعل العنصر متاح للإستخدام من داخل الصنف نفسه فقط.

○ القيمة الافتراضية: public.

# محددات الوصول access modifiers

■ يبرز دورها لتطبيق مبدأ مهم من مبادئ البرمجة الكائنية، وهو مبدأ حماية البيانات لغرض المحافظة على الحالة الداخلية state للكائن.

# The protected Modifier

- The protected modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package

Visibility increases



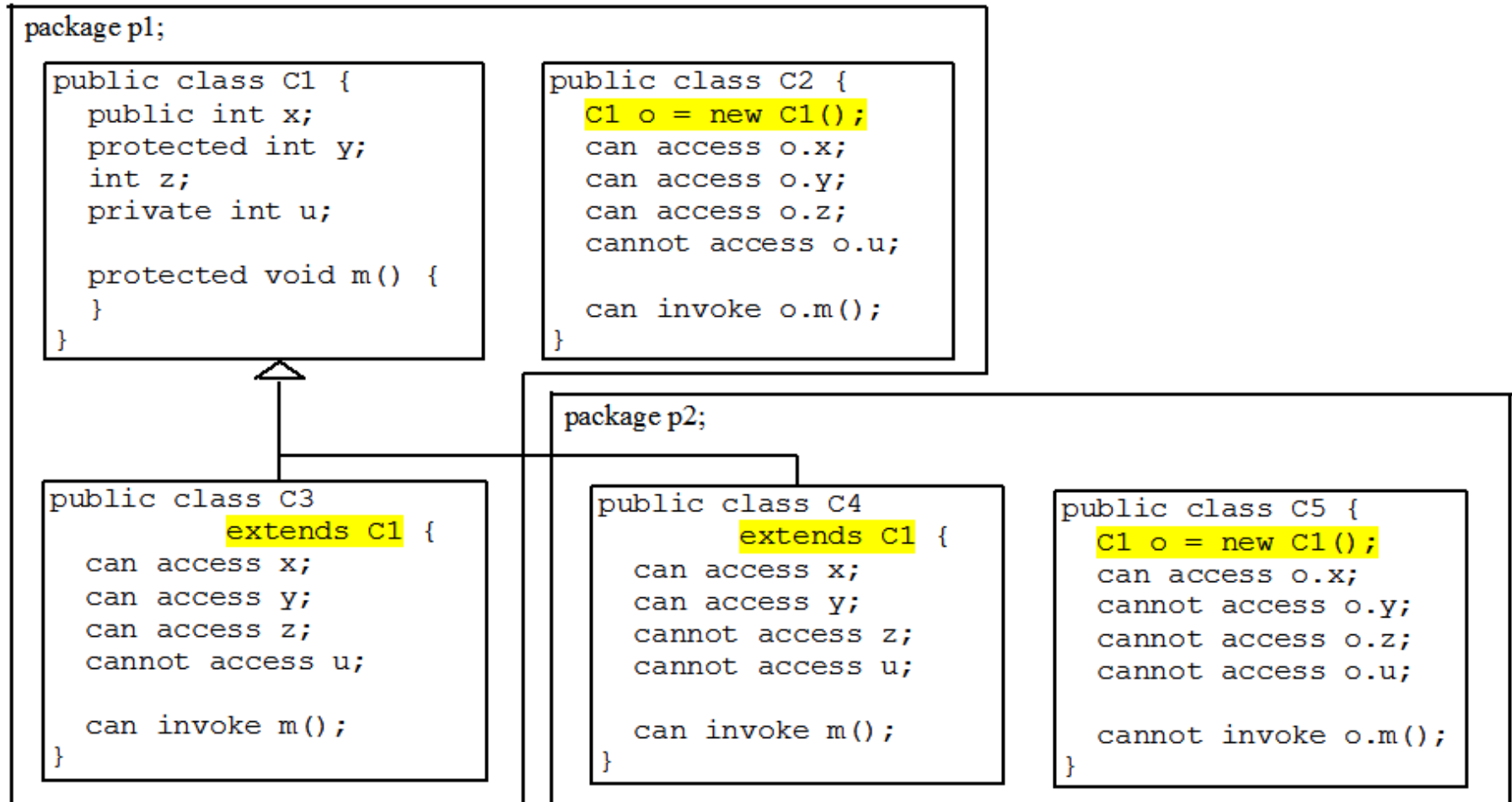
private, none (if no modifier is used), protected, public



# Accessibility

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

# Visibility Modifiers



# تغليف البيانات بالعمليات encapsulation

■ ربط البيانات بالعمليات (أو الطرق) الممكنة عليها  
وتجميعها داخل الكائن أو الصنف.

■ تغليف الحالة الداخلية للكائن، ليظهر الكائن  
ومحتوياته بشكل مغلف (كما الكبسولة!).

← لا يمكن التعامل مع بيانات الكائن إلا من خلال  
الطرق المحددة لذلك والمعرفة داخل الكائن نفسه.

# تغليف البيانات بالعمليات encapsulation

■ ليس جمع محتويات الكائن داخل كبسولة فحسب، ولكن التأكد من أن جزءاً فقط مما في الكبسولة يبدو ظاهراً أو مرئياً من الخارج.

← الطرق العامة public methods

■ هذه الأجزاء الظاهرة هي المدخل الذي يمكن مستخدمي هذا الصنف من الاستفادة من خدماته التي يقدمها.

```

1 public class Circle {
2     /** The radius of this circle */
3     private double radius=1.0;
4
5     /** return the area of this circle */
6     public double getArea() {
7         return radius * radius * Math.PI;
8     }
9 }

```

تغليف البيانات

تعريف الطرق العامة للوصول  
إلى البيانات المغلفة وذلك لحماية  
الحالة الداخلية للكائن

# لماذا التغليف؟

■ حماية الحالة الداخلية للصنف.

■ مثال:

```
Circle aCircle = new Circle(4.5);
```

تنشئ هذه الطريقة الكائن aCircle بنصف قطر مساو لـ 4.5، كما تقوم أيضاً بتخزين قيمة 63.6 في متغير المساحة (راجع الطريقة المنشئة للصنف).

```
aCircle.radius = 6.0; // you could do this if  
radius was public
```

# لماذا التغليف؟

■ لو لم تتم حماية المتغير radius، يصبح من الممكن تنفيذ الجملة التالية:

```
aCircle.radius = 6.0; // you could  
do this if radius was public
```

يؤدي ذلك إلى وقوع خلل في حالة الكائن aCircle:

← متغير المساحة لا يزال يحمل قيمة 63.6،

← بالتالي، الطريقة getArea ستسترجع قيمة خاطئة عن مساحة الكائن.

# تغليف البيانات بالعمليات encapsulation

■ الجزء الثاني من تطبيق مبدأ التغليف يكمن في القدرة على تحديد العمليات الممكنة على البيانات المخزنة داخل الكائن.

← يجب أن لا يزيد عدد الطرق methods الناتجة من عملية التصميم على احتياجات الكائن وبياناته.



# أسئلة ذاتية

■ باستخدام تعريف الصنف Circle الظاهر في الشكل (٦-١)، أكتب الطريقة setDimensions التي تستقبل قيمة جديدة للمتغير radius (شرط ان لا تقل القيمة عن الصفر)، كما يجب أن تحدث قيمة المتغير area للمحافظة على تناسق بيانات الكائن. تسترجع الطريقة قيمة منطقية (true/false) للدلالة على تنفيذ عملية التحديث.

## أسئلة ذاتية

■ نكتب الجمل التالية لإنشاء كائن دائرة aCircle وتعيين قيمة 4.5 لمتغير نصف القطر لهذا الكائن:

```
Circle aCircle = new Circle();  
aCircle.setDimensions(4.5);
```

■ هل من الممكن القيام بالشيء نفسه باستخدام الجمل التالية؟ علل إجابتك.

```
Circle aCircle = new Circle();  
aCircle.radius=4.5; aCircle.area=63.6;
```

# أسئلة ذاتية

■ ينص مبدأ التغليف على ضرورة تعريف المتغيرات باستخدام المحدد الخاص `private` وتعريف العمليات على تلك المتغيرات على شكل طرق عامة `public methods`. ما الغرض من ذلك؟

■ قد يحتوي تعريف الصنف المغلف `encapsulated class` على بعض الطرق `methods` المعرفة باستخدام محدد الرؤية `private`، علل ذلك.

## تطبيق عملي (٦-١)

■ اكتب الصنف Counter يكون مسؤولاً فقط عن تنفيذ عمليات العد. يقوم الصنف بالاحتفاظ بقيمة العداد شرط أن يكون عدداً صحيحاً وموجباً، ويحتوي على الطرق العامة التالية (انظر الحقيقة).

■ أكتب البرنامج TestCounter وذلك لاختبار الصنف Counter شرط أن يحتوي البرنامج على جمل التكرار التالية (انظر الحقيقة).

طرق الاسترجاع والتغيير

## Accessor & Mutator Methods

تستخدم هذه الطرق كوسيلة لجعل البيانات المخزنة داخل متغيرات الصنف ظاهرة للعالم الخارجي.

# طرق الاسترجاع Accessors

توفر إمكانية الوصول إلى المتغيرات:

```
aCircle.getRadius();
```

```
aCircle.getArea();
```

وتتم إضافة تعريف لتلك الطرق داخل صنف الدائرة  
Circle class بالشكل التالي:

```
public double getRadius() {
```

```
    return radius; }
```

```
public double getArea() {
```

```
    return area; }
```

# طرق التغيير Mutators

■ توفر إمكانية التحكم في قيم البيانات المخزنة:

```
aCircle.setRadius(6);
```

■ ويتم تعريف الطريقة setRadius داخل الصنف

:Circle

```
public void setRadius(double newRadius) {  
    radius = newRadius;  
}
```

# طرق التغيير وحماية البيانات

■ قد يبدو استخدام طرق التغيير Mutators مناقضاً لمبدأ حماية البيانات

○ مستخدم الصنف أصبح قادراً الآن على الوصول إلى البيانات وتغيير قيمها!



# طرق التغيير وحماية البيانات

■ إن الوصول إلى بيانات المتغير من خلال طريقة method يختلف تماماً عن إعطاء الصلاحية للوصول إليها بشكل مباشر.

■ من خلال الطرق يمكننا:

- التحكم في القيم التي سيتم تخزينها في المتغيرات
- القيام باللائم للمحافظة على الحالة الداخلية للصنف.

# طرق التغيير وحماية البيانات

مثلاً، الأسلوب السليم لكتابة طريقة تغيير قيمة نصف القطر السابقة يجب أن يكون كالتالي:

```
public void setRadius(double newRadius) {  
    radius = newRadius;  
    area = newRadius*newRadius*Math.PI;  
}
```

# طرق الاسترجاع والتغيير

## Accessor & Mutator Methods

عادة ما تتم تسمية هذه النوعية من الطرق (accessors and mutators) باستخدام كلمة `get` متبوعة باسم المتغير المراد استرجاع قيمته (مثل `getRadius`)، أو كلمة `set` متبوعة باسم المتغير المراد تغيير قيمته (مثل `setRadius`).

# الإسم الداخلي للكائن: `this`

■ عند الإشارة إلى عنصر من عناصر الكائن، فإننا نستخدم اسم الكائن ورمز النقطة (.) dot operator واسم العنصر المراد الوصول إليه.

■ للوصول إلى عناصر الصنف من داخله، نستخدم إسم العنصر بشكل مباشر:

```
public Circle() {  
    radius = 1.0; }
```

# الإسم الداخلي للكائن: **this**

■ من الممكن الوصول إلى المتغيرات المعرفة داخل الصنف باستخدام الإسم الداخلي (this):

```
/** construct a circle object of a  
fixed radius */
```

```
public Circle() { this.radius = 1.0; }
```

← يستخدم الصنف الاسم **this** للإشارة إلى الكائن نفسه

# أسئلة ذاتية

أكتب تعريفاً للصنف Person لتمثيل إسم الشخص ومتغير آخر لتمثيل عمره، وطريقة منشئة تستقبل قيم البيانات عند الإنشاء. كما يتضمن التعريف على طرق تغيير وطرق إسترجاع لكل متغير من متغيرات الصنف. بالإضافة، يحتوي تعريف الصنف على طريقة reset لتغيير قيمة كلا المتغيرين بقيم جديدة يتم إستقبالها من خلال المعاملات.

# أسئلة ذاتية

## أعد كتابة الطريقة :setDimensions

```
public boolean setDimensions(double r) {  
    if (radius < 0) return false;  
    radius = r; area = getArea();  
    return true;}  
}
```

■ شرط أن يظهر جزء الرأس للطريقة كالتالي (يتطابق  
إسم المعامل مع إسم متغير الصنف):

```
boolean setDimensions(double radius)
```

## تطبيق عملي (٥-٢)

■ اكتب الصنف Temperature لتمثيل درجات الحرارة باستخدام كلا المقياسين: السلسيوس Celsius و الفهرنهايت Fahrenheit (انظر الحقيبة لتفاصيل الصنف).

■ اكتب البرنامج TemperatureDemo لتجربة الصنف Temperature، بحيث يقوم باختبار جميع الطرق العامة المتوافرة في الصنف.



# عناصر الحالة instance members والعناصر الساكنة static members

■ نوعان من العناصر التي قد يحتويها الصنف:

- نوع يعتمد على حالة محددة (أي كائن معين) من الصنف، تعرف العناصر المنتمية إلى الكائن باسم عناصر الحالة instance members.
- نوع آخر لا ينتمي إلى كائن معين بل إنه مشترك بين جميع الكائنات. تعرف العناصر المنتمية إلى الصنف باسم العناصر الساكنة static members.

# عناصر الحالة instance members

■ تعرف المتغيرات المستخدمة لتمثيل بيانات الكائن بمتغيرات الحالة instance variables.

○ كل كائن ينشأ من الصنف يحتوي على قيمه الخاصة.

■ تعرف العمليات التي تعمل على بيانات الكائن بطرق الحالة instance methods.

○ تنفذ عملياتها على متغيرات الحالة instance vars.

■ من غير الممكن استخدام المحدد static مع هذه النوعية من العناصر.

```
1 public class Circle{
2     // INSTANCE VARIABLES
3     private double radius; // radius of this circle
4
5     // CONSTRUTORS
6     /* construct a circle object of a fixed radius value */
7     public Circle() {
8         radius = 1.0;
9     }
10    /* construct a circle object of some radius value */
11    public Circle(double radius) {
12        this.radius = radius;
13    }
14    // INSTANCE METHODS
15    /* return the area of this circle */
16    public double getArea() {
17        return radius * radius * Math.PI;
18    }
19 }
```

متغيرات الحالة

instance variable

طرق الحالة

instance methods

# العناصر الساكنة static members

■ هنالك بعض الحالات التي تستدعي استخدام بعض عناصر الصنف باستقلالية تامة عن أي كائن.

■ تعرف هذه النوعية من العناصر باسم العناصر الساكنة static members ذلك أن قيمها لا ترتبط بكائن معين، بل تبقى ساكنة أثناء التنفيذ ولا تتغير من كائن إلى آخر.

■ تستخدم المحدد static.

```

1 public class Circle{
2     private double radius; // radius of this circle object
3     public static String shape = "Circle"; // shape of ALL circle objects
4
5     /* construct a circle object of a fixed radius value */
6     public Circle() {
7         radius = 1.0;
8     }
9     /* construct a circle object of some radius value */
10    public Circle(double radius) {
11        this.radius = radius;
12    }
13    /* return the area of this circle */
14    public double getArea() {
15        return radius * radius * Math.PI;
16    }
17    /* return the area of a circle with radius r */
18    public static double getArea(double r) {
19        return r * r * Math.PI;
20    }
21    // display shape info
22    public void displayRadius() {
23        System.out.println("Radius: " + radius);
24    }
25    public static void displayShape() {
26        System.out.println("Shape: " + shape);
27    }
28 }

```

متغير ساكن  
static variable

طريقة ساكنه  
static methods

طريقة ساكنه  
static methods

```

/** The radius of this circle */
private double radius;
/** Name of shape created by this class */
private static String shape = "Circle";

public Circle(double newRadius) {
    radius = newRadius;
}

```

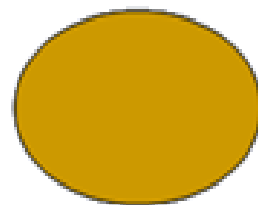
```

aCircle = new Circle(1.0);
bCircle = new Circle(2.0);

```



aCircle



bCircle

radius=1.0;  
Shape=same as class

radius=4.0;  
Shape=same as class

Circle.radius ☒  
Circle.shape ☒

Class name: Circle  
Data (attributes)  
radius = \_\_\_\_  
Shape="Circle"  
Methods (behavior)  
getArea()  
getArea(double r)  
displayArea()  
displayShape()

شكل (١-٤): تشترك الكائنات في قيمة المتغير الساكن static variable. في حين تختلف القيم المخزنة في متغير الحالة instance variable من كائن لآخر.

# المتغيرات الساكنة

بشكل عام، فإنه من غير الممكن الوصول إلى متغيرات الحالة إلا من خلال إنشاء الكائنات أولاً ومن ثم استخدامها للوصول إلى تلك المتغيرات، على خلاف المتغيرات الساكنة والتي من الممكن الوصول إليها من خلال استخدام اسم الصنف فقط (شرط أن تكون معرفة public):

**ClassName.variable**

# الطرق الساكنة

■ بالنسبة للطرق الساكنة، تكون قادرة على القيام بدورها بغض النظر عن البيانات المخزنة داخل الكائن (أي أنها لا تعتمد في عملها على قيم متغيرات الحالة instance variables).

■ من الممكن تنفيذ هذه النوعية من الطرق دون إنشاء كائن: **ClassName.method()**



# الطرق الساكنه

الطرق التي تعرف على أنها ساكنة static تخضع لعدة قيود مثل:

- تتمكن من إستدعاء طرق ساكنة فقط.
- تتعامل مع بيانات ساكنة فقط (سواء من المتغيرات أو المعاملات).
- لا تستطيع أن تشير إلى this أو super وذلك لإرتباط هذه المعارف بالكائن (أو الحالة).

# مثال

■ من أشهر أنواع الطرق الساكنة: `main ( )`

○ تستدعى قبل إنشاء أي كائن.

○ بالتالي، لا يمكنها التعامل سوا مع العناصر الساكنة (سواء من المتغيرات أو الطرق).

```
public static void main(...)
```

# أسئلة ذاتية

هل من الممكن استدعاء طريقة حالة non-static methods من طريقة ثابتة static method؟ هل من الممكن استدعاء طريقة ثابتة static method من داخل طريقة حالة non-static method؟

استخدمنا في تعريف الصنف Circle الاستدعاء: Math.PI وضح الأجزاء المختلفة لهذا الاستدعاء، وماذا تستنتج بالنسبة للبيانات المستدعاة، هل هي ساكنة static أم غير ساكنة non-static.

## تدريب عملي (٦-٣)

■ اكتب الصنف MyDate لتمثيل التاريخ، بحيث يحتوي على المتغيرات المناسبة لتحديد التاريخ (انظر الحقيقية لتفاصيل الصنف).

■ اكتب البرنامج MyDateDemo لاختبار الصنف MyDate بحيث يعرض قائمة رئيسية لإختيار العملية (انظر الحقيقية للتفاصيل).

## تطبيق عملي (٤-٦)

■ اكتب الصنف ElectricalFormulas لغرض  
تجميع عدة طرق ساكنة static methods لتنفيذ  
القوانين الكهربائية التالية واسترجاع النتائج (لن  
يحتوي الصنف على أي تعريف لمتغيرات حالة  
instance variables). (انظر الحقيقة للقوانين).

■ استخدم طرق الصنف ElectricalFormulas  
لكتابة برنامج لحساب القيم الكهربائية (انظر  
الحقيقة).

# الجلسة الثانية

## ■ مبادئ البرمجة الكائنية

- التجريد abstraction.
- التغليف encapsulation.
- التوريث inheritance.

# أسلوب البرمجة الكائنية

لا يعتمد فقط على استخدام الكائنات والأصناف، بل إنه يعتمد أيضاً على بعض المبادئ الأساسية والتي لا بد من اتباعها عند تصميم البرامج.

# مبادئ البرمجة الكائنية

■ أهم المبادئ التي يعتمد عليها أسلوب البرمجة الكائنية:

- تغليف البيانات بالعمليات encapsulation.
- التجريد abstraction.
- التوريث inheritance.
- التعددية polymorphism.



# التجريد abstraction

■ هو مبدأ التصميم الذي يسعى لحماية الكيفية التي يعمل بها الصنف ويقوم بإظهار الماهية فقط.

■ يعرف أيضاً بمبدأ

إخفاء المعلومات information hiding.

# التجريد abstraction

■ المبرمج المستخدم لخدمات صنف آخر لا يكون بحاجة إلى معرفة "كيف" ينفذ الخدمات، وإنما يحتاج إلى معرفة "ماذا" يقدم من خدمات.

■ يتم تطبيق هذا الأسلوب أثناء كتابتنا للبرامج من خلال تطبيق مبدأ التغليف encapsulation وكتابة الأصناف المغلفة بشكل محكم.

# التغليف encapsulation

■ إضافة إلى ذلك، فإن مبدأ التغليف يعتمد على إخفاء التفاصيل المغلفة داخل "الكبسولة" بحيث يكون الصنف قادراً على تغيير الطريقة الداخلية التي يقوم بها لتنفيذ عمله دون أن يؤثر ذلك في عمل النظام العام.

# تصميم أصناف مغلفة

## well-encapsulated classes

- (١) كتابة التعليقات في بداية تعريف الصنف لوصف كيفية استخدامه.
- (٢) تعريف متغيرات الحالة instance variables بأنها خاصة private.
- (٣) تعريف طرق عامة public methods لتنفيذ جميع العمليات الممكن إجراؤها على بيانات الحالة الخاصة private instance variables.

# تصميم أصناف مغلفة

## well-encapsulated classes

- (٤) كتابة التعليقات في بداية كل طريقة عامة لتوضيح استخدامها والدور الذي تقوم به، وعدد ونوع المعاملات الممررة لها، ونوع الناتج المتوقع.
- (٥) تحويل جميع الطرق المساندة (التي تقوم بدور داخلي) إلى طرق خاصة `private methods`.

# تصميم أصناف مغلفة

## well-encapsulated classes

من خلال تطبيق مبدأ إخفاء المعلومات وتغليفها، يصبح من الممكن تعريف الأصناف وإعادة استخدامها في البرامج المختلفة مع ضمان عملها بالأسلوب نفسه.

← إنشاء مكتبات من الأصناف:

### Java Class Library (API)

# مكتبة الأصناف في لغة جافا

- تحتوي لغة جافا على مجموعة كبيرة من أصناف الكائنات الجاهزة للاستخدام، وقد تم تصنيفها وتجميعها في مكتبة الأصناف للغة جافا API.
- تم تنظيم الأصناف في تلك المكتبة باستخدام أسلوب الحزم packages.
- تتوفر محتويات المكتبة بالتنظيم نفسه في موقع المكتبة الرسمي على الإنترنت.

قائمة بأسماء الحزم  
package names

قائمة بأسماء الأصناف  
class names

(اخترنا الصنف Scanner)

Java™ Platform Standard Ed. 6

All Classes Packages

java.applet  
java.awt

SAXResult  
SAXSource  
SAXTransformerFac  
Scanner  
Scanner class in java.util  
ScheduledExecutor  
ScheduledFuture  
ScheduledThreadPool  
Schema  
SchemaFactory  
SchemaFactoryLoad  
SchemaOutputReso  
SchemaValidationEv  
PrintContent

Overview Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS  
SUMMARY: NESTED | FIELD | CONSTR | METHOD  
DETAIL: FIELD | CONSTR | METHOD

FRAMES NO FRAMES  
DETAIL: FIELD | CONSTR | METHOD

Java™ Platform Standard Ed. 6

java.util

**Class Scanner**

java.lang.Object  
└ java.util.Scanner

All Implemented Interfaces:

Iterator<String>

public final class Scanner  
extends Object  
implements Iterator<String>

http://docs.oracle.com/javase/6/docs/api/java/util/Scanner.html

شرح للصنف Scanner

شكل (1-1): توثيق الصنف Scanner كما يبدو في مكتبة الأصناف Java API.



# مثال (١): الصنف Scanner

- متوفر من خلال الحزمة java.util.
- مسؤولاً عن تنفيذ عمليات قراءة البيانات من التدفقات المختلفة، سواء كان من شاشة المستخدم أو من الملفات.
- قمنا خلال الأيام التدريبية الماضية بتعريف الكائنات من هذا الصنف واستخدامها لتنفيذ عمليات قراءة البيانات من الشاشة.

```
// creating a Scanner object (keyboard)  
Scanner keyboard = new Scanner(System.in) ;  
System.out.println("enter your name: ");  
// calling public method nextInt  
// on the keyboard object  
String name = keyboard.nextLine() ;  
System.out.println("hello " + name);
```

## مثال (٢): الصنف JOptionPane

■ متوفر من خلال مكتبة الأصناف الرسومية  
java.javax.

■ مسؤولاً عن تنفيذ عمليات عرض/ قراءة البيانات  
باستخدام النوافذ ومربعات الحوار.

■ قمنا خلال الأيام التدريبية الماضية باستخدام  
عمليات هذا الصنف لتنفيذ عمليات عرض وقراءة  
البيانات من خلال النوافذ.

## مثال (٢): الصنف JOptionPane

■ نلاحظ بأن هذا الصنف يقوم بتعريف العمليات على أنها ساكنة static وبالتالي يتم إستدعائها بإستخدام اسم الصنف دون الحاجة إلى إنشاء الكائنات.

```
// calling the public STATIC methods:  
// showInputDialog and showMessageDialog  
// of the JOptionPane class, with out the  
// need for creating an object  
String name =  
JOptionPane.showInputDialog("Enter your  
name");  
JOptionPane.showMessageDialog(null, "hello " +  
name);
```

# مكتبة الأصناف في لغة جافا

تعتبر مكتبة الأصناف من إحدى أهم مميزات لغة جافا ، حيث إنها تحتوي على عدد كبير جداً من الأصناف المفيدة والتي يمكن إعادة استخدامها بشكل فعال في مختلف أنواع البرامج مثل الرسومات المعقدة، وواجهة المستخدم، والطباعة، والشبكات، الحماية، وقواعد البيانات، والوسائط المتعددة، وغيرها.

# أسئلة ذاتية

■ عند كتابة واجهة الصنف class interface لا يتم عرض الأعضاء المعرفة باستخدام محدد الرؤية الخاص private، علل ذلك.

# أسئلة ذاتية

■ إطلع على تعريف واجهة الصنف String في مكتبة الأصناف من خلال موقعها على الإنترنت، وإبحث عن الطرق الممكن تنفيذها للقيام بالتالي:

- إسترجاع مصفوفة من الحروف chars للنص.
- إسترجاع قيمة الـ hash code للنص.
- إسترجاع قيمة نصية مماثلة لقيمة النص، بعد حذف الفراغات الظاهرة في بداية أو نهاية النص.



## أسئلة ذاتية

أكتب جملة واحدة لكل طريقة من الطرق التي تم  
تحديدتها في السؤال السابق، لتنفيذها على كائن  
النص التالي (لاحظ ظهور الفراغات في بداية النص  
ونهايته):

```
String sentence = "    A mother is  
the truest friend    ";
```

# الجلسة الثانية

## ■ مبادئ البرمجة الكائنية

التجريد abstraction. ☒

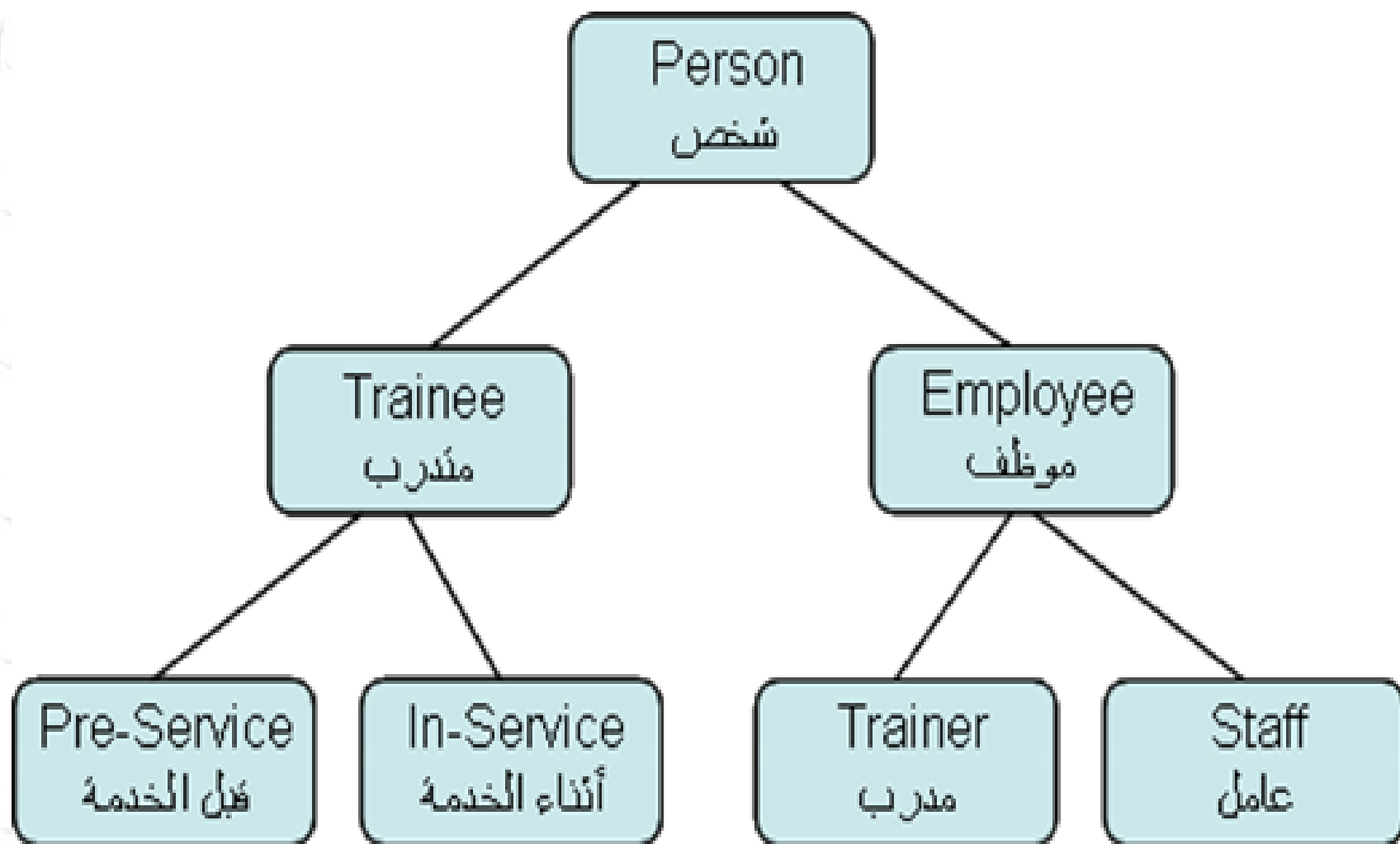
التغليف encapsulation. ☒

التوريث inheritance. ☒

# التوريث inheritance

■ تنظيم الأصناف وذلك عن طريق إيجاد الخصائص والسلوكيات المشتركة.

← كتابة أصناف مشتقة من أصناف معرفة مسبقاً دون الحاجة إلى إعادة كتابة الخصائص والسلوكيات المشتركة.

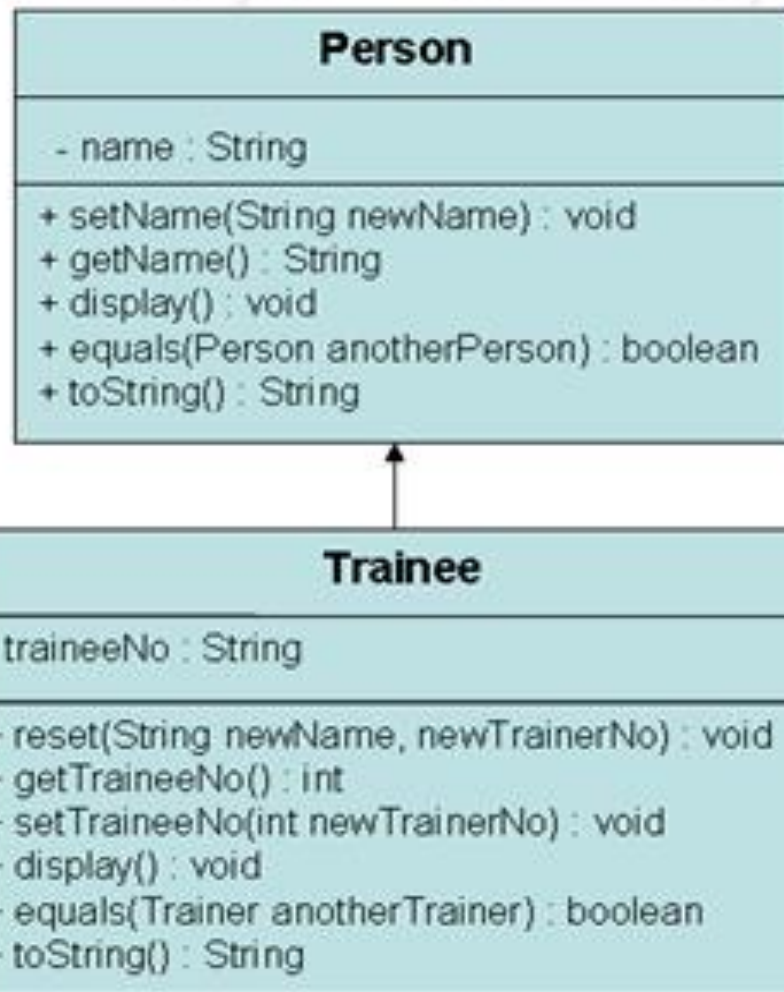


**شكل (٧-١): تصميم كائنات الأشخاص في نظام معهد التدريب.**

Person
- name : String
+ setName(String newName) : void + getName() : String + display() : void + equals(Person anotherPerson) : boolean + toString() : String

**شكل (١-٨): تمثيل الصنف Person بلغة النمذجة الموحدة UML.**  
**يمثل الرمز "-" العناصر الخاصة private members بينما يستخدم الرمز "+" لتمثيل العناصر العامة public members.**

```
public class Person {  
    private String name;  
    public Person() {  
        name = "No name yet.";  
    }  
    public Person(String initialName) {  
        name = initialName;  
    }  
    public void display() {  
        System.out.println("\n Name: "+name);  
    }  
    // rest of code in listing 6-4  
} // Person
```



**شكل (٦-٩):** يعتبر الصنف Trainee صنف مشتق من Person. وهو يمثل الصنف الفرعي sub-class، في حين يمثل Person الصنف العلوي super-class في هذه العلاقة.

```
public class Trainee extends Person {  
    // INSTANCE VARIABLE  
    private int traineeNo;  
  
    public Trainee() {  
        super(); // Call for Person class  
        traineeNo = 0;  
    } // constructor.  
  
    public Trainee(String initialName,  
                    int initialtraineeNo) {  
        super(initialName);  
        traineeNo = initialtraineeNo;  
    } // constructor  
  
    .. continues on next slide
```



```
// INSTANCE METHODS
```

```
public void reset(String newName,  
                  int newTraineeNo) {  
    setName(newName);  
    traineeNo = newTraineeNo;  
}
```

```
public void display() {  
    super.display();  
    System.out.println(" Trainee  
                        Number: " + traineeNo);  
}
```

```
    // rest of code in listing 6-4
```

```
} // Trainee
```

```
public class TraineeDemo {  
    public static void main(String [] args) {  
        Trainee t = new Trainee();  
        t.display();  
        System.out.println();  
        t.reset("Fahad", 1234);  
        t.display();  
        t.setName("Fahad Abdullah");  
        // setName appears in Person and yet the  
        // object t of type Trainee is capable of  
        // executing it as a result of inheritance  
        t.display(); } // main  
} // TraineeDemo
```

# Command Prompt

```
C:\javaCode>java TraineeDemo
```

```
Name: No name yet.
```

```
Trainee Number: 0
```

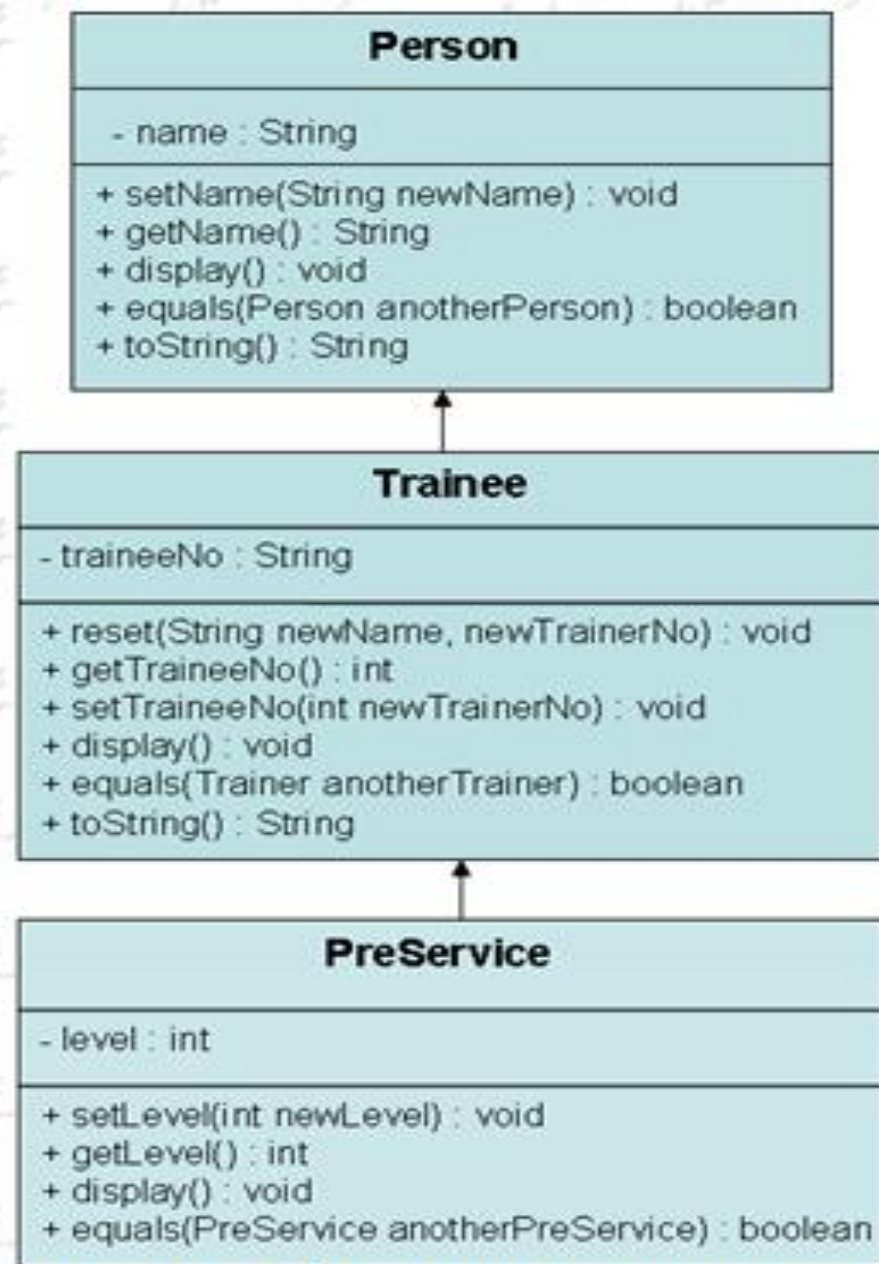
```
Name: Fahad
```

```
Trainee Number: 1234
```

```
Name: Fahad Abdullah
```

```
Trainee Number: 1234
```

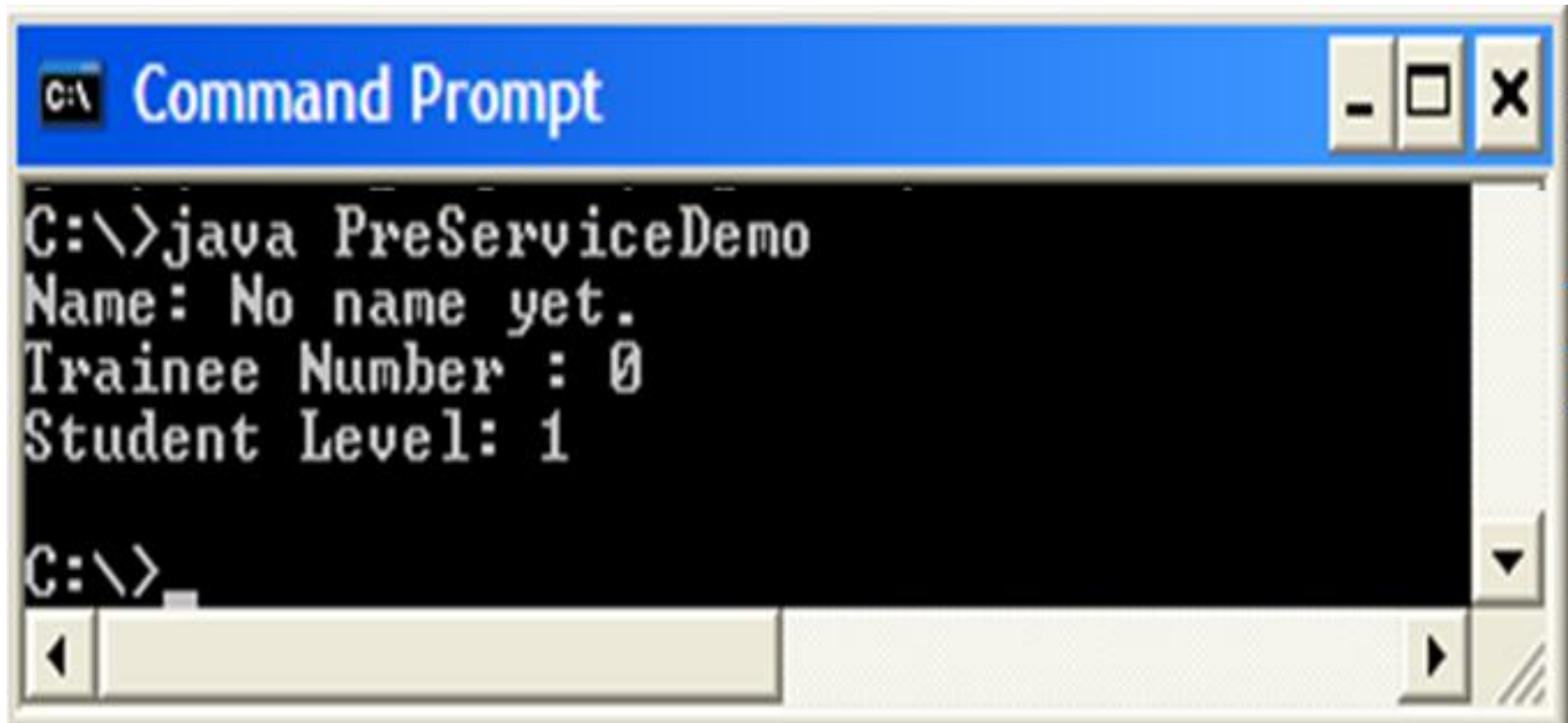
```
C:\javaCode>
```



**شكل (٦-١١): يرث الصنف PreService من الأصناف العلوية Person و Trainer ويضيف تعاريف للمتغيرات والطرق الخاصة به.**



```
public class PreServiceDemo {  
    public static void main(String[] args) {  
        PreService student = new PreService();  
        student.display(); } // main  
    } // PreServiceDemo
```



The screenshot shows a Windows Command Prompt window with a blue title bar labeled "C:\ Command Prompt". The window contains the following text:

```
C:\>java PreServiceDemo  
Name: No name yet.  
Trainee Number : 0  
Student Level: 1  
  
C:\>
```

The output of the Java program is displayed line by line. At the bottom of the window, there is a scroll bar and navigation buttons.

# توافق الأنواع بين الأصناف المشتقة

- من خلال علاقة التوريث بين الأصناف، يتمكن الصنف المشتق sub-class من إحتواء جميع العناصر من الصنف العلوي super-class.
- ← أي كائن منشئ من صنف مشتق:
  - يمثل حالة instance من ذلك الصنف المشتق.
  - بالإضافة، يمثل حالة instance من الصنف العلوي. (والعكس غير صحيح).

# ملاحظة

## ■ ملاحظة :

لاستخدام الأصناف الموجودة في مشروع package x  
بداخل مشروع package y، يجب:

١- اضافة package x بداخل Library

٢- أو عمل export ل package x ك Jar file

و من ثم اضافة Jar file بداخل مشروع package y

ويجب استخدام import لاسم package مثل:

**import traineedemo.Trainee;**



مثلاً..

كل كائن Trainee هو أيضاً كائن Person  
ولكن لا يمكن إعتبار كل Person على أنه  
Trainee.

← الكائنات من الأنواع المشتقة قادرة على تمثيل أي  
نوع من الأنواع العلوية، بالإضافة إلى نوعها.

# توافق الأنواع بين الأصناف المشتقة

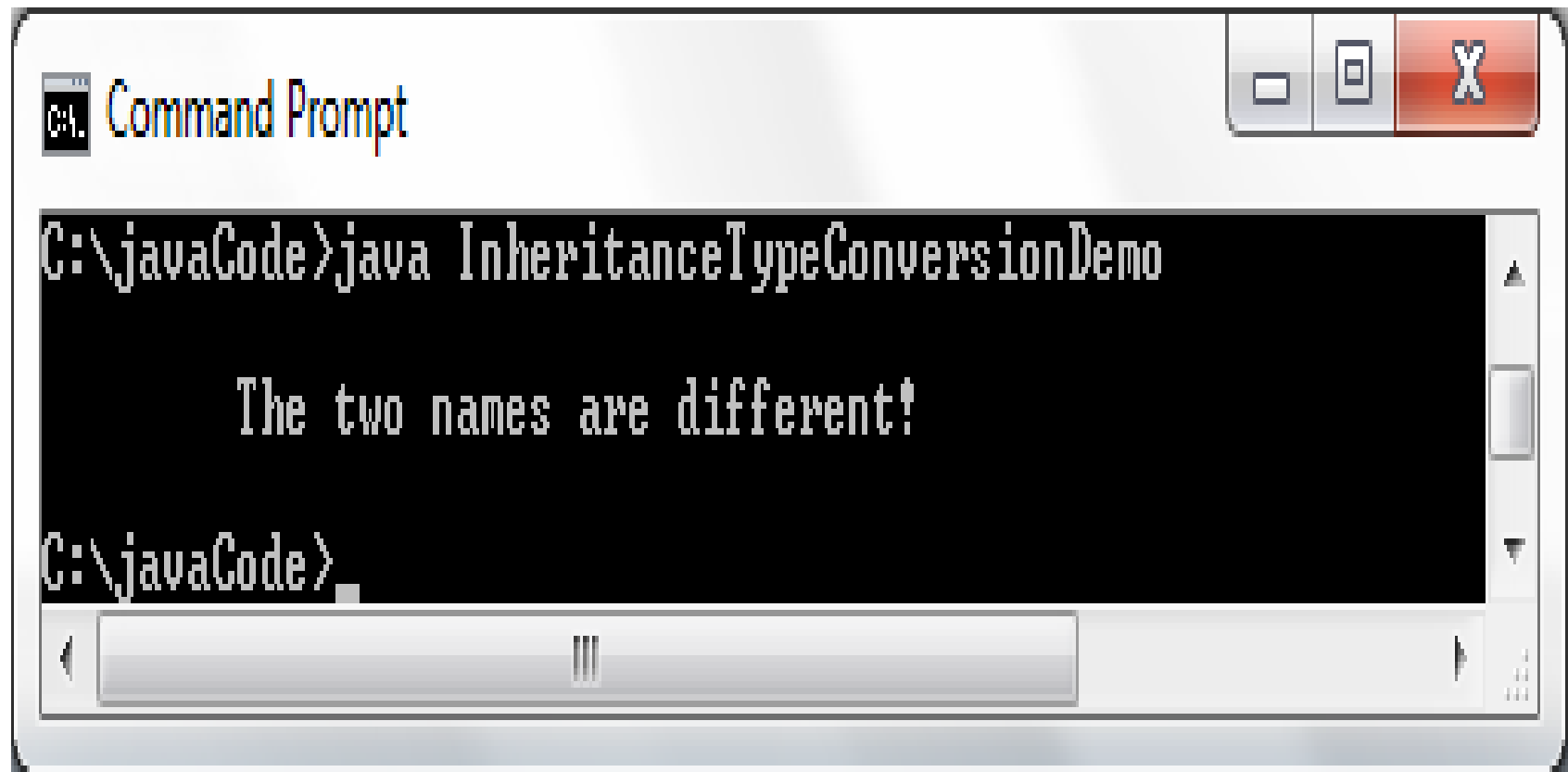
■ يوفر ذلك قدراً كبيراً من المرونة:

○ كتابة الطرق التي تحتوي معاملاتها على أصناف علوية لتكون قادرة على إستقبال أي من الأصناف العلوية أو المشتقة.

○ كتابة الطرق التي تسترجع أصناف علوية لتكون قادرة على إسترجاع أي من الأصناف العلوية أو المشتقة.

■ يساعد ذلك في تطبيق مبدأ التعددية polymorphism، كما سنرى لاحقاً.

```
class InheritanceTypeConversionDemo {
    public static void main(String[] args) {
        Trainee student1 = new
            Trainee("Ibrahim Hassan", 12345);
        PreService student2 = new
            PreService("Ahmed Ali", 456780, 1);
        compareNames(student1, student2); }
    public static void compareNames
        (Person p1, Person p2) {
        if (p1.getName().equals(p2.getName()))
            System.out.println(" identical!");
        else System.out.println(" different!");
        } }
```



```
Command Prompt

C:\javaCode>java InheritanceTypeConversionDemo

    The two names are different!

C:\javaCode>
```

# Object

■ الصنف العلوي الافتراضي لجميع الأصناف في لغة الجافا.

← إذا لم يتم المبرمج بتحديد صنف علوي، يقوم مترجم الجافا بتحديد الصنف Object كصنف علوي super-class للصنف الجديد.

# Object

■ الصنف العلوي لجميع الأصناف في لغة الجافا.

■ الغرض: استخدام النوع Object عند تحديد المعاملات المعلنه للطرق لغرض جعل تلك الطرق قادرة على التعامل مع الكائنات من أي نوع عند إرسالها عن طريق المعاملات الفعلية.

← مكتبات الأصناف: استقبال الكائنات من خلال طرقها والقيام بالمهام المطلوبة منها بغض النظر عن نوع الكائنات المرسله.

# Object

■ يقوم الصنف Object بتعريف بعض الطرق مثل طريقة toString وطريقة equals.

← جميع الأصناف تكون محتوية على تعريف لتلك الطرق.

← يمكن إعادة تعريفها إذا رغب المبرمج بذلك.

# The toString() method in Object

toString()

إرجاع تمثيل سلسلة من الكائن. يعيد التنفيذ الافتراضي سلسلة تتكون من اسم الصنف لتمثيل الكائن، علامة @، ورقم يمثل هذا الكائن

```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

الكود اعلاه سينتج : Loan@15037e5



# The equals Method

`equals()`

تحتوي على كائنين: التنفيذ الافتراضي لطريقة  
`equals` في صنف كائن كما يلي:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

For example, the  
`equals` method is  
overridden in  
the `Circle`  
class.

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle)o).radius;  
    }  
    else  
        return false;  
}
```

# أسلوب منع التوريث

```
final public class A {  
    // ...  
}
```

// The following class is **illegal**.

```
public class B extends A { // ERROR!  
Can't subclass A  
    // ...  
}
```

# Overriding vs. Overloading

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

# أسئلة ذاتية

- ما الفرق بين مفهوم زيادة تحميل الطرق **overloading** ومفهوم استبدال الطرق **overriding**؟ وضح الإجابة باستخدام مثال.
- ما الفرق بين المعرفين **this** و **super**؟ ما الدور الذي تقوم به عند استدعائها من داخل الطريقة المنشئة للكائن؟

## أسئلة ذاتية

- نفرض أن الصنف SportCar هو صنف مشتق من الصنف Automobile. ولنفرض أن الصنف Automobile يقوم بتعريف متغير الحالة الخاص private instance variable التالي: manufacturer.
- هل سيحتوي الكائن mySportCar المعرف من النوع SportCar على المتغير manufacturer؟ لماذا؟

# أسئلة ذاتية

■ نفرض أن الصنف SportCar هو صنف مشتق من الصنف Automobile. ولنفرض أن الصنف Automobile يقوم بتعريف طريقة الحالة العامة public instance method التالية:

increaseSpeed.

هل سيحتوي الكائن mySportCar المعروف من النوع SportCar على تعريف للطريقة increaseSpeed؟ لماذا؟

# أسئلة ذاتية

حدد الجمل الصحيحة legal والجمل غير الصحيحة  
illegal:

```
Person p1 = new Trainee();  
Person p2 = new InService();  
Trainee t1 = new Person();  
Trainee t2 = new InService();  
Object ob = new Trainee();  
PreService p1 = new Object();
```

## تدريب عملي (٥-٦):

■ اكتب الصنف `TitledPerson` والمشتق من الصنف العلوي `Person` الظاهر في القائمة (٦-٣)، بحيث يحتوي على متغير حالة واحد: `title` لتخزين القيم: `Mr, Ms, Mrs, Miss, Dr`. (انظر الحقبة لتفاصيل العمليات).

■ اكتب الصنف `TitledPersonDemo` لتجربة الصنف `...TitledPerson`



## تدريب عملي (٦-٦)

■ أكتب برنامج ShapeDemo.java والذي يقوم بعرض قائمة إختيار ومن ثم إحتساب قيمة المساحة والمحيط للشكل الذي تم إختياره، على أن يتم إستدعاء الطريقة المناسبة من أحد الأصناف الظاهرة في الشكل. تظهر عينه من التنفيذ (انظر الحقيبة).

# الجلسة الثانية

■ مبادئ البرمجة الكائنية

■ التعددية polymorphism.

# التعددية Polymorphism

تحديد بعض السلوكيات المشتركة، ومن ثم تحديد أسلوباً لتنفيذ من خلال تعريف أصناف فرعية متخصصة.

← "واجهة واحدة وعدة طرق , one interface ,  
multiple methods ."

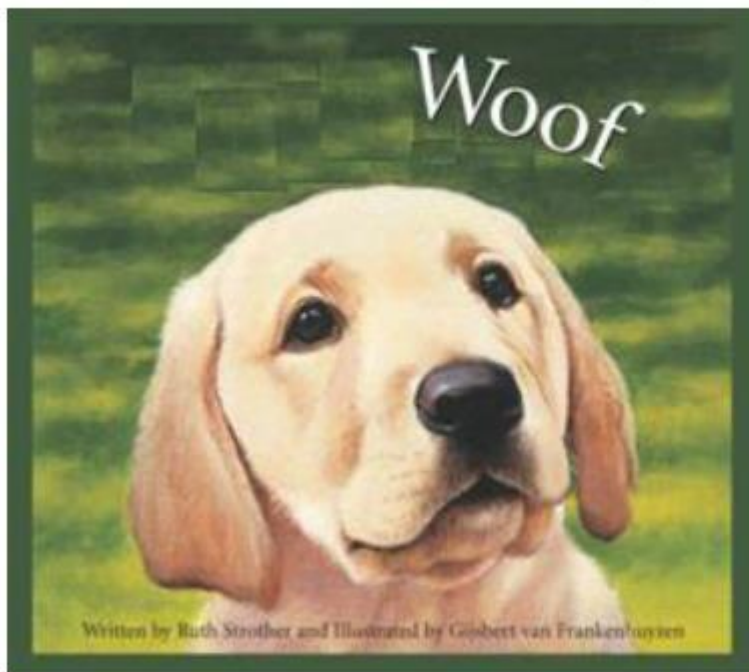
## تطبيق التعددية باستخدام أسلوب الواجهات.

```
public interface Animal {  
    public String talk();  
}  
  
public class Cat implements Animal {  
    public String talk() {  
        return "Meow!";  
    } // talk  
} // Cat  
  
public class Dog implements Animal {  
    public String talk() {  
        return "Woof!";  
    } // talk  
} // Dog  
  
public class PolymorphismEx1 {  
    public static void write(Animal a) {  
        // POLYMORPHIC METHOD  
        System.out.println(a.talk());  
    }  
    public static void main() {  
        write(new Cat());  
        write(new Dog());  
    } // main  
} // PolymorphismEx1
```

## تطبيق التعددية باستخدام أسلوب الأصناف الموزعة.

```
public abstract class Animal {  
    public abstract String talk();  
}  
  
public class Cat extends Animal {  
    public String talk() {  
        return "Meow!";  
    } // talk  
} // Cat  
  
public class Dog extends Animal {  
    public String talk() {  
        return "Woof!";  
    } // talk  
} // Dog  
  
public class PolymorphismEx2 {  
    public static void write(Animal a) {  
        // POLYMORPHIC METHOD  
        System.out.println(a.talk());  
    }  
    public static void main() {  
        write(new Cat());  
        write(new Dog());  
    } // main  
} // PolymorphismEx2
```

ناتج تنفيذ الطريقة write على كائن Dog



ناتج تنفيذ الطريقة write على كائن Cat



بالتالي، نقول عن هذه الطريقة بأنها متعددة الأشكال *polymorphic*.

# التعددية Polymorphism

■ تعريف الطرق methods القادرة على العمل مع أصناف متعددة بسلوكيات مختلفة.

← يعتمد السلوك على نوع الكائن الذي يتم تنفيذ الطريقة عليه.

■ توفر لغة الجافا إمكانية تحديد السلوك من قبل الأصناف العلوية، وتأجيل التنفيذ ليتم بعد ذلك من خلال الأصناف الفرعية.

# التعددية Polymorphism

■ يتم ذلك من خلال:

- تحديد السلوكيات من خلال تعريف واجهة interface لتقوم الأصناف المشتقة بكتابة التطبيق implementation للطرق المحددة في الواجهة.
- تحديد السلوكيات من خلال تعريف طرق موجزة abstract classes في الصنف العلوي لتقوم الأصناف المشتقة بعد ذلك بكتابة التطبيق implementation لتلك الطرق الموجزة.

# واجهات الأصناف interfaces

■ عن مكون برمجي يحتوي عناوين لطرق method heading.

← لا يحتوي تعريف الواجهة على التطبيق البرمجي لتلك الطرق methods.

■ عادة ما يحتوي تعريف الواجهة على تعليقات لتوضيح الغرض من كل طريقة يحتويها وذلك لمساعدة المبرمج الذي سيقوم بكتابة الأجزاء البرمجية لتلك الطرق.



```
/**
 * An interface for methods that return
 * the perimeter and area of an object
 */
public interface Measurable {
    /** Returns the perimeter */
    public double getPerimeter();

    /** Returns the area */
    public double getArea();
}
```

```
class Square implements Measurable {  
    private double side;  
    public Square(double side) {  
        this.side = side; }  
    public double getSide() {  
        return side; }  
    public double getArea() {  
        return side * side;  
    }  
    public double getPerimeter() {  
        return 4 * side;  
    }  
}
```

```
class Circle implements Measurable {  
    private double radius;  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
    public double getArea() {  
        return radius*radius*Math.PI;  
    }  
    public double getPerimeter() {  
        return 2*radius*Math.PI;  
    }  
}
```

```
class Rectangle implements Measurable {  
    private double width;  
    private double height;  
    public Rectangle(double w, double h) {  
        width=w; height=h  
    }  
    public double getArea() {  
        return 2 * (width+height);  
    }  
    public double getPerimeter() {  
        return width * height;  
    }  
}
```

## قائمة (٦-٤١)

يستخدم البرنامج InterfaceDemo.java واجهة  
الصنف Measurable لتطبيق مبدأ التعددية  
polymorphism

Welcome to Area & Perimeter Calculating Program

1. Circle.
2. Square.
3. Rectangle.
4. Quit.

--> choice <1, 2, 3 or 4> : 2

enter side: 3  
Perimeter = 12.0; area = 9.0

press ENTER to continue...

1. Circle.
2. Square.
3. Rectangle.
4. Quit.

--> choice <1, 2, 3 or 4> : 3

enter width: 6  
enter height: 8  
Perimeter = 28.0; area = 48.0

press ENTER to continue...

# الأصناف الموجزة abstract classes

■ توفر إمكانية تحديد الطرق على أن تقوم الأصناف الفرعية بتنفيذها.

← الطرق التي يتم توصيفها بشكل موجز في الصنف العلوي تستخدم المحدد abstract والذي يعني ضرورة قيام الأصناف المشتقة بتعريف تلك الطرق.

■ إذا لم يقم الصنف المشتق بتعريف الطرق الموجزة في الصنف الأب، فإن المترجم سيعرض خطأ أثناء الترجمة لتنبيه المبرمج لذلك.

```
public abstract class ShapeAbstract {  
    private String color;  
    public ShapeAbstract() {  
        color = "black";  
    }  
    public void setColor(String color) {  
        this.color = color;  
    }  
    public String getColor() {  
        return color;  
    }  
    public abstract void display();  
}
```



```
class Square extends ShapeAbstract {  
    private double side;  
    public Square(double side) {  
        this.side = side; }  
    public void display() {  
        System.out.println("\n\t A square  
        with " + side + " side dimension: ");  
        System.out.println(" area of square = "  
        + super.round(side*side));  
        System.out.println("\t\t perimeter of  
        square = " + (4*side));  
    } // display  
} // Square
```

```

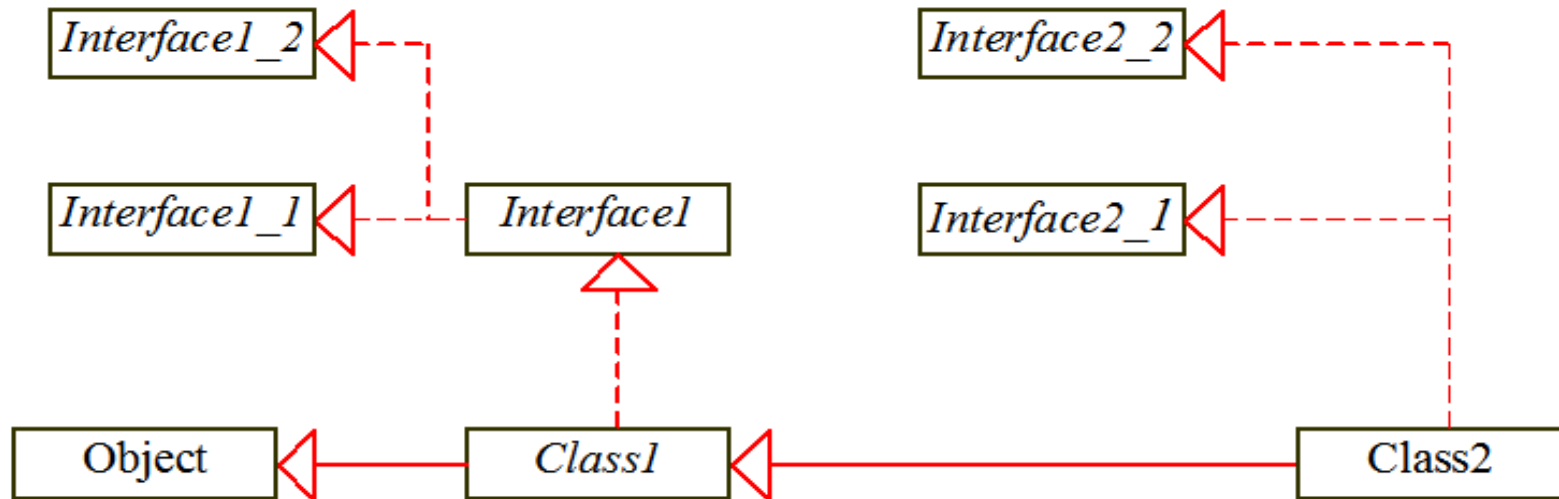
class Circle extends ShapeAbstract {
    private double radius;
    public Circle(double radius) {
        this.radius = radius; }
    public void display() {
        System.out.println("\n\t A circle "
            + radius + " radius dimension: ");
        System.out.println("\t\t area = "
            + radius*radius*Math.PI);
        System.out.println("\t perimeter = "
            + (2*radius*Math.PI);
    }
} // Circle

```

```
class Rectangle extends ShapeAbstract {  
    private double width;  
    private double height;  
    public Rectangle(double w, double h) {  
        width=w; height=h; }  
    public void display() {  
        System.out.println("\t\t area of  
            rectangle = " + (width*height));  
        System.out.println("\t\t perimeter = "  
            + (2* (width+height)) );  
    } // display  
} // Rectangle
```

# Interfaces vs. Abstract Classes, cont.

- All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.



## قائمة (٦-٩)

يستخدم البرنامج ShapeAbstractDemo.java  
الأصناف الموجزة abstract classes لتطبيق  
التعددية polymorphism

Welcome to Area & Perimeter Calculating Program

1. Circle.
2. Square.
3. Rectangle.
4. Quit.

--> choice <1, 2, 3 or 4> : 2

enter side: 3

A square with 3.0 side dimension:  
 area of square = 9.0  
 perimeter of square = 12.0

press ENTER to continue...

1. Circle.
2. Square.
3. Rectangle.
4. Quit.

--> choice <1, 2, 3 or 4> : 3

enter width: 6  
 enter height: 8

A rectangle with width: 6.0 and height: 8.0 dimensions:  
 area of rectangle = 48.0  
 perimeter of rectangle = 28.0

press ENTER to continue...

## تدريب عملي (٦-٧)

■ نقوم في هذا التدريب بتنفيذ الواجهة **Measurable** من خلال الصنف الموجز **ShapeAbstract**.

■ إحصل على نسخة من البرنامج **ShapeAbstractDemo.java** وعدل الصنف الموجز **ShapeAbstract** على النحو التالي (انظر الحقيقة).