بسم الله الرحمن الرحيم

لغة البرمجة جافا

Java Programming Language

الدرس السادس :

تقنيات إعادة استخدام الصفوف

مقدمة :

احدى الميزات الهامة للبرمجة غرضية التوجه هي إعادة استخدام الصفوف المعرفة مسبقا..

أسلوب إعادة الاستخدام يحقق عن طريق التركيب و الوراثة

التركيب :

في البرامج التي كتبناها سابقا قمنا باستخدام التركيب..

التركيب وضع عنوان غرض داخل تعريف الصف الجديد ..

لنفرض أننا نريد بناء صف يحوي مجموعة من السلاسل و مجموعة حقول أولية (primitive)

بالنسبة للحقول الاولية نعرفها بشكل مباشر أما بالنسبة للحقول غير الأولية فإننا نضع عنوان الغرض

مثال :

```
class WaterSource {
      private String s;

      WaterSource() {
            System.out.println("WaterSource()");
            s = new String("Constructed");
      }

      public String toString() { return s; }
}

public class SprinklerSystem {
      private String valve1, valve2, valve3, valve4;
      WaterSource source;
      int i;
      float f;

      void print() {
            System.out.println("valve1 = " + valve1);
            System.out.println("valve2 = " + valve2);
            System.out.println("valve3 = " + valve3);
            System.out.println("valve4 = " + valve4);
            System.out.println("i = " + i);
            System.out.println("f = " + f);
            System.out.println("source = " + source);
      }
```

```
	public static void main(String[] args) {
		SprinklerSystem x = new SprinklerSystem();
		x.print();
	}
}
```

كل صف جاهز أو نقوم بتعريفه يملك طريقة toString() و هذه الطريقة تقوم بتحويل الغرض الذي يقوم باستدعائها إلى سلسلة ...

يمكن إعادة تعريف هذه الطريقة بالنسبة للصفوف التي نقوم بتعريفها

و هذه الطريقة تستدعى بشكل تلقائي عندما تكون هناك حاجة لتحويل الغرض إلى سلسلة

و يمكن أيضا استدعائها بشكل صريح

ففي مثالنا عندما قمنا بكتابة السطر التالي :

```
System.out.println("source = " + source);
```

نلاحظ هنا أننا نحاول دمج سلسلة مع غرض و هنا يتم بشكل تلقائي استدعاء الطريقة toString() الخاصة بالغرض و تتم عملية الدمج بنجاح

كما ذكرنا سابقا إن لغة جافا تضمن أن جميع المتحولات التي تعرف تأخذ قيم بدائية ... و الخرج يوضح ذلك

كما ذكرنا سابقا ... طرق تهيئة الأغراض تتم بثلاث طرق :

- في مكان تعريف الغرض و هذا يعني أن الغرض يبنى بشكل دائم قبل استدعاء الباني
- داخل الباني
- قبل الاستخدام الفعلي للغرض

الطرق الثلاثة موضحة في المثال التالي :

```
class Soap {
	private String s;

	Soap() {
		System.out.println("Soap()");
		s = new String("Constructed");
	}

	public String toString() { return s; }
}

public class Bath {
	private String
	// Initializing at point of definition:
	s1 = new String("Happy"),
	s2 = "Happy",
	s3, s4;
```

```
	Soap castille;
	int i;
	float toy;

	Bath() {
		System.out.println("Inside Bath()");
		s3 = new String("Joy");
		i = 47;
		toy = 3.14f;
		castille = new Soap();
	}

	void print() {
		// Delayed initialization:
		if(s4 == null)
			s4 = new String("Joy");
		System.out.println("s1 = " + s1);
		System.out.println("s2 = " + s2);
		System.out.println("s3 = " + s3);
		System.out.println("s4 = " + s4);
		System.out.println("i = " + i);
		System.out.println("toy = " + toy);
		System.out.println("castille = " + castille);
	}

	public static void main(String[] args) {
		Bath b = new Bath();
		b.print();
	}
}
```

تتم مناقشة الكود و توضيح الفكرة ..

<u>الوراثة :</u>

الوراثة جزء مكمل للغة Java (و جميع لغات البرمجة)

بشكل ضمني جميع الصفوف ترث من الصف Object ..

عندما نقوم بعمل وراثة بين صفين فذلك يعني " الصف الجديد يشبه الصف القديم "

لتحقيق الوراثة بين صفين نضع الكلمة المفتاحية extends بعد اسم الصف الجديد و بعدها اسم الصف الأب و قبل قوس بداية تعريف الصف ..

عندما نقوم بذلك فإن جميع المتحولات الأعضاء و التوابع الأعضاء في الصف الأب تصبح متاحة للصف الابن

<u>مثال :</u>

```
class Cleanser {
      private String s = new String("Cleanser");

      public void append(String a) { s += a; }
      public void dilute() { append(" dilute()"); }
      public void apply() { append(" apply()"); }
      public void scrub() { append(" scrub()"); }
      public void print() { System.out.println(s); }
      public static void main(String[] args) {
            Cleanser x = new Cleanser();
            x.dilute(); x.apply(); x.scrub();
            x.print();
      }
}

public class Detergent extends Cleanser {
      // Change a method:
      public void scrub() {
            append(" Detergent.scrub()");
            super.scrub(); // Call base-class version
      }
      // Add methods to the interface:
      public void foam() { append(" foam()"); }
      // Test the new class:
      public static void main(String[] args) {
            Detergent x = new Detergent();
            x.dilute();
            x.apply();
            x.scrub();
            x.foam();
            x.print();
            System.out.println("Testing base class:");
            Cleanser.main(args);
      }
}
```

معنى كلمة super : تشير إلى الصف الأب ...

تتم قراءة الكود و مناقشته

<u>تهيئة الصف الأب :</u>

الآن لدينا صفين : صف أب و صف ابن ...

الصف الابن يرث ضمنيا كل ما لدى الصف الأب

عندما ننشأ غرض من الصف الابن فإنه يحتوي بداخله غرض جزئي .. هذا الغرض الجزئي من نوع الصف الأب و لكن هذا الغرض الجزئي يكون مغلف داخل الغرض الذي قمنا بإنشاءه .

لذلك من الضروري أن يكون هذا الغرض الجزئي مهيأ بشكل صحيح ..

الطريقة الوحيدة لعمل ذلك هي القيام بالتهيئة في الباني و ذلك باستدعاء باني الصف الأب

في java .. تدخل عمليات استدعاء باني الصف الأب داخل باني الصف الابن بشكل أوتوماتيكي ..

مثال :

```
class Art {
      Art() {
            System.out.println("Art constructor");
      }
}

class Drawing extends Art {
      Drawing() {
            System.out.println("Drawing constructor");
      }
}

public class Cartoon extends Drawing {
      Cartoon() {
            System.out.println("Cartoon constructor");
      }

      public static void main(String[] args) {
            Cartoon x = new Cartoon();
      }
}
```

من الخرج نلاحظ أن البناء يتم من الصف الأب نزولا إلى الأبناء ..

و في حال كان الصف الابن لا يحتوي باني .. فإن المترجم ينشأ له باني افتراضي و هذا الباني الافتراضي يقوم باستدعاء باني الأب بشكل أوتوماتيكي ..

في المثال السابق كانت جميع التوابع البناءة هي توابع بناءة افتراضية (من دون وسطاء)

بفرض أن الصف الأب لا يحتوي باني افتراضي أو أن الصف الأب يحتوي أكثر من باني و أحد هذه البواني افتراضي و الآخر باني بوسطاء ..

فإن المترجم بشكل افتراضي يقوم باستدعاء الباني الافتراضي للصف الأب في باني الابن

و لكن إذا أردنا استدعاء الباني ذو الوسطاء فإن الطريقة الوحيدة لعمل ذلك عن طريق الكلمة المفتاحيةsuper و نمرر لها وسطاء التابع الباني الذي نريد استدعاءه

إذا لم نقم باستدعاء باني الأب و كان الأب لا يحتوي باني افتراضي فإن ذلك يؤدي إلى حصول خطأ في زمن الترجمة

و كذلك يجب أن يكون استدعاء باني الأب هو اول تعليمة في باني الابن

مثال :

```
class Game {
      Game(int i) {
            System.out.println("Game constructor");
      }
}

class BoardGame extends Game {
      BoardGame(int i) {
            super(i);
            System.out.println("BoardGame constructor");
      }
}

public class Chess extends BoardGame {
      Chess() {
            super(11);
            System.out.println("Chess constructor");
      }

      public static void main(String[] args) {
            Chess x = new Chess();
      }
}
```

الدمج بين الوراثة و التركيب :

عادة و الأكثر شيوعا هو استخدام الوراثة و التركيب معا

المثال التالي يبين كيفية إنشاء صف معقد باستخدام الوراثة و التركيب

المثال :

```
class Plate {
      Plate(int i) {
            System.out.println("Plate constructor");
      }
}

class DinnerPlate extends Plate {
      DinnerPlate(int i) {
            super(i);
            System.out.println(
            "DinnerPlate constructor");
      }
}
```

```
class Utensil {
      Utensil(int i) {
            System.out.println("Utensil constructor");
      }
}

class Spoon extends Utensil {
      Spoon(int i) {
            super(i);
            System.out.println("Spoon constructor");
      }
}

class Fork extends Utensil {
      Fork(int i) {
            super(i);
            System.out.println("Fork constructor");
      }
}

class Knife extends Utensil {
      Knife(int i) {
            super(i);
            System.out.println("Knife constructor");
      }
}

class Custom {
      Custom(int i) {
            System.out.println("Custom constructor");
      }
}

public class PlaceSetting extends Custom {
      Spoon sp;
      Fork frk;
      Knife kn;
      DinnerPlate pl;

      PlaceSetting(int i) {
            super(i + 1);
            sp = new Spoon(i + 2);
            frk = new Fork(i + 3);
            kn = new Knife(i + 4);
            pl = new DinnerPlate(i + 5);
            System.out.println(
            "PlaceSetting constructor");
      }

      public static void main(String[] args) {
            PlaceSetting x = new PlaceSetting(9);
```

```
	}
}
```

تتم مناقشة الكود و الشرح خطوة خطوة ..

عملية هدم الأغراض :

لغة Java لا تملك مفهوم الهادم (مثل C++) و عملية الهدم يقوم بها الـgarbage collector كما مر سابقا

و لكن أحيانا ربما نريد أن نقوم بعملية الهدم بشكل صريح .. إلا أننا لا نعلم متى و أين سيتم استدعاء الـ garbage collector .. لذلك علينا أن نكتب طريقة تقوم بعملية الهدم بشكل صريح ..

كما سيمر لاحقا في مناقشة الأخطاء أثناء التنفيذ (الاستثناءات) يجب وضع استدعاء عملية الهدم في كتلةfinally .

مثال :

```
import java.util.*;

class Shape {
	Shape(int i) {
		System.out.println("Shape constructor");
	}

	void cleanup() {
		System.out.println("Shape cleanup");
	}
}

class Circle extends Shape {
	Circle(int i) {
		super(i);
		System.out.println("Drawing a Circle");
	}

	void cleanup() {
		System.out.println("Erasing a Circle");
		super.cleanup();
	}
}
class Triangle extends Shape {
	Triangle(int i) {
		super(i);
		System.out.println("Drawing a Triangle");
	}

	void cleanup() {
		System.out.println("Erasing a Triangle");
```

```
            super.cleanup();
        }
}

class Line extends Shape {
        private int start, end;

        Line(int start, int end) {
            super(start);
            this.start = start;
            this.end = end;
            System.out.println("Drawing a Line: " +
            start + ", " + end);
        }

        void cleanup() {
            System.out.println("Erasing a Line: " +
            start + ", " + end);
            super.cleanup();
        }
}

public class CADSystem extends Shape {
        private Circle c;
        private Triangle t;
        private Line[] lines = new Line[10];

        CADSystem(int i) {
            super(i + 1);
            for(int j = 0; j < 10; j++)
            lines[j] = new Line(j, j*j);
            c = new Circle(1);
            t = new Triangle(1);
            System.out.println("Combined constructor");
        }

        void cleanup() {
            System.out.println("CADSystem.cleanup()");
            // The order of cleanup is the reverse
            // of the order of initialization
            t.cleanup();
            c.cleanup();
            for(int i = lines.length - 1; i >= 0; i--)
                lines[i].cleanup();
            super.cleanup();
        }

        public static void main(String[] args) {
            CADSystem x = new CADSystem(47);
            try {
            // Code and exception handling...
```

```
            } finally {
            x.cleanup();
            }
      }
}
```

كل غرض في المثال السابق هو من نوع shape (و هو بدوره نوع من Object حيث أنه يرث منه بشكل ضمني)

كل صف يرث من الصف shape يعيد تعريف الطريقة cleanup بالاضافة إلى استدعاء الـ cleanup الخاصة بالأب (shape) و ذلك باستخدام الكلمة المفتاحية super ..

في التابع main () نلاحظ وجود الكلمتين المفتاحيتين try و finally .. سيتم شرحها بالتفصيل لاحقا .

و لكن كلمة try تشير أن الكتلة التي بعدها هي عبارة عن منطقة حرجة و التي تعالج بشكل مختلف

إحدى هذه الطرق هي الكتلة finally و هذه الكتلة تنفذ دوما حتى لو كان هناك أخطاء في تنفيذ بعض التعليمات في كتلة try

في مثالنا ... كتلة finally تقوم بعملية الهدم بشكل دائم

بالنسبة لتابع الهدم (cleanup) .. يجب الانتباه لترتيب الهدم ... و هنا يتم اتباع نفس الاسلوب في لغة C++ حيث أنه يجب الهدم بترتيب عكس ترتيب البناء

إخفاء الأسماء :

إذا كان لدينا طريقة في الصف الأب و تمت عملية تحميل بشكل زائد لهذه الطريقة أكثر من مرة و من ثم قمنا بإعادة تعريف الطريقة في الصف الابن فإن النسخة الخاصة بالأبن لا تلغي نسخة الأب ..

مثال :

```
class Homer {
      char doh(char c) {
            System.out.println("doh(char)");
            return 'd';
      }

      float doh(float f) {
            System.out.println("doh(float)");
            return 1.0f;
      }
}

class Milhouse {}

class Bart extends Homer {
      void doh(Milhouse m) {}
}
```

```
public class Hide {
      public static void main(String[] args) {
            Bart b = new Bart();
            b.doh(1); // doh(float) used
            b.doh('x');
            b.doh(1.0f);
            b.doh(new Milhouse());
      }
}
```

اختيار الوراثة أو التركيب :

التركيب يستخدم عادة عندما تريد الاستفادة من ميزات الصف الموجود داخل الصف الجديد ..

و نوع علاقة التركيب هو علاقة "يملك" (has a)

مثال :

```
class Engine {
      public void start() {}
      public void rev() {}
      public void stop() {}
}

class Wheel {
      public void inflate(int psi) {}
}

class Window {
      public void rollup() {}
      public void rolldown() {}
}

class Door {
      public Window window = new Window();

      public void open() {}
      public void close() {}
}

public class Car {
      public Engine engine = new Engine();
      public Wheel[] wheel = new Wheel[4];
      public Door left = new Door(),
      right = new Door(); // 2-door
      public Car() {
            for(int i = 0; i < 4; i++)
```

```
            wheel[i] = new Wheel();
      }

      public static void main(String[] args) {
            Car car = new Car();
            car.left.window.rollup();
            car.wheel[0].inflate(72);
      }
}
```

تتم مناقشة الكود ...

الوراثة تعني أننا نريد تخصيص صف ما .. أي عندما نرث من صف معين فهذا يعني أننا حصلنا على نسخة خاصة من الصف الأب .

و علاقة الوراثة هي من نوع "هو" (is a) .. فمثلا السيارة هي نوع من المركبات و ليس السيارة جزء من المركبات

الكلمة المفتاحية protected :

و تعني أن حقل ما أو طريقة ما غير متاحة إلا في الصف نفسه أو الصفوف التي ترث من الصف

في حين أن private تعني أن حقل ما أو طريقة ما غير متاحة إلا في الصف نفسه

أما public فهي متاحة في جميع الصفوف و يمكن الوصول إليها عن طريق غرض من الصف من أي صف آخر

مثال :

```
class V {
      private int i;
      protected int read() { return i; }
      protected void set(int ii) { i = ii; }
      public V(int ii) { i = ii; }
      public int value(int m) { return m*i; }
}

public class O extends V {
      private int j;
      public O(int jj) { super(jj); j = jj; }
      public void change(int x) { set(x); }
}
```

الطريقة set يمكن رؤيتها لأنها protected .. في حين لا يمكن التعامل مع الحقل i مباشرة لأنه private

عملية الوراثة تشبه عملية التطوير المتتالي .. بمعنى أننا نكتب صف و من ثم نطوره و ذلك بأن نرثه و نضيف عليه و من ثم نطور الصف الجديد بنفس الأسلوب و هكذا ..

ملاحظة عملية : لا ينصح أن تكون شجرة الوراثة كبيرة ... مستويين أو ثلاثة على الأكثر

مفهوم الـ upcasting :

مفهوم الوراثة ليس مجرد أن صف ما يرث من صف آخر و بالتالي فإن جميع الطرائق في الصف الأب متاحة للابن بل هناك علاقة بين الصف الأب و الصف الابن و هذه العلاقة هي " الصف الابن هو نوع من الصف الأب "

فمثلا لنفرض أن لدينا صف يعبر عن الأدوات الموسيقية و صف يعبر عن الناي .. من الواضح أن الناي هو نوع من الأدوات الموسيقية و بالتالي أي رسالة يمكن ارسالها للصف الأب يمكن ارسالها للصف الابن (تنفيذ طريقة)

أي إذا كان لدينا طريقة play في الصف الأب فإنه يمكن استدعاؤها من خلال غرض من الصف الابن

مثال :

```
import java.util.*;

class Instrument {
      public void play() {}

      static void tune(Instrument i) {
      // ...
      i.play();
      }
}

// Wind objects are instruments
class Wind extends Instrument {
      public static void main(String[] args) {
      Wind flute = new Wind();
      Instrument.tune(flute); // Upcasting
      }
}
```

بما أن الطريقة tune تأخذ وسيط من نوع Instrument و الـ Wind يرث من الـ Instrument و بالتالي هي نوع منه .. أي يمكن استدعاء الطريقة على غرض من النوعWind و هذا ما يسمى upcasting .

لا تنسوني من صالح دعائكم

تم بحمد الله