بسم الله الرحمن الرحيم

لغة البرمجة جافا

Java Programming Language

الدرس التاسع :

تعددية الأشكال

<u>مقدمة :</u>

تعددية الأشكال هي الصفة الأساسية الثالثة للبرمجة غرضية التوجه بعد التجريد (مفهوم الصف) و الوراثة .

في هذا الفصل سنتعلم مفهوم تعددية الأشكال أو ما يسمى الربط الديناميكي أو الربط أثناء التنفيذ أو التنفيذ الديناميكي

Upcastng :

سابقا تحدثنا كيف أنه يمكن استخدام غرض ما على أنه غرض من نوع الصف الذي ينتمي إليه أو من نوع صف الأب

أي معالجة غرض على أنه غرض من النوع الأب يسمىupcasting .

<u>مثال :</u>

```
class Note {
      private int value;

      private Note(int val) { value = val; }

      public static final Note
                 MIDDLE_C = new Note(0),
                 C_SHARP = new Note(1),
                 B_FLAT = new Note(2);
}

class Instrument {
      public void play(Note n) {
            System.out.println("Instrument.play()");
      }
}

// Wind objects are instruments
// because they have the same interface:
class Wind extends Instrument {
      // Redefine interface method:
      public void play(Note n) {
            System.out.println("Wind.play()");
      }
}

public class Music {
      public static void tune(Instrument i) {
            // ...
            i.play(Note.MIDDLE_C);
      }
```

```
	public static void main(String[] args) {
		Wind flute = new Wind();
		tune(flute); // Upcasting
	}
}
```

إذا فرضنا أننا خصصنا الطريقة ()tune بحيث تأخذ وسيط غرض من نوع Wind ..

فإذا قمنا بإنشاء صف جديد يرث من الصف Instrument فإن الطريقة () tune لا يمكن استدعائها عن طريق غرض من الصف الجديد لأنه لا يمكن القسر بين الأبناء

لذلك نحن بحاجة لتعريف طريقة () tune لكل نوع من أنواع الآلات الموسيقية .. أي سنقوم بكتابة المزيد من الكود

<u>مثال :</u>

```
class Note {
	private int value;

	private Note(int val) { value = val; }

	public static final Note
			MIDDLE_C = new Note(0),
			C_SHARP = new Note(1),
			B_FLAT = new Note(2);
}

class Instrument {
	public void play(Note n) {
		System.out.println("Instrument.play()");
	}
}
class Wind extends Instrument {
	public void play(Note n) {
		System.out.println("Wind.play()");
	}
}

class Stringed extends Instrument {
	public void play(Note n) {
		System.out.println("Stringed.play()");
	}
}

class Brass extends Instrument {
	public void play(Note n) {
		System.out.println("Brass.play()");
	}
```

```
}

public class Music2 {
      public static void tune(Wind i) {
            i.play(Note.MIDDLE_C);
      }
      public static void tune(Stringed i) {
            i.play(Note.MIDDLE_C);
      }
      public static void tune(Brass i) {
            i.play(Note.MIDDLE_C);
      }

      public static void main(String[] args) {
            Wind flute = new Wind();
            Stringed violin = new Stringed();
            Brass frenchHorn = new Brass();
            tune(flute); // No upcasting
            tune(violin);
            tune(frenchHorn);
      }
}
```

نلاحظ في هذا المثال أن ثلاث طرق () tune كل واحدة منها أصبحت تأخذ كوسيط نوع من أنواع الآلات الموسيقية .

و لا يوجد طريقة () tune تأخذ كوسيط غرض من نوع الأب مما أدى لكتابة المزيد من الكود .

فالطريقة السابقة أفضل (كتابة طريقة واحدة يتم استدعائها من قبل جميع الأغراض التي ترث من صف أب واحد)

و بالتالي إذا نسينا تحميل الطريقة بشكل زائد فإن ذلك لا يهم لأنه الوسيط هو من نوع الأب و بالتالي يمكن استدعاء الطريقة من أي غرض ابن

و هذا بشكل عام مبدأ تعددية الأشكال .

<u>عملية ربط الاستدعاءات :</u>

عملية ربط استدعاء الطريقة بجسم الطريقة تسمى binding .

تتم عملية binding إما قبل التنفيذ أو أثناء التنفيذ

عندما تتم عملية binding قبل التنفيذ تسمى عملية ربط مبكرة .

و هناك أسلوب آخر لعملية binding و يحدث أثناء التنفيذ و يسمى ربط متأخر .

في المثال السابق لا يمكن للمترجم أن يحدد أي طريقة سيتم استدعاؤها عندما تتم تنفيذ طريقة()tune أي لا يمكن القيام بعملية ربط مبكر .. لذلك فالحل هو استخدام عملية الربط المتأخر .

الربط المتأخر يعني أن الربط بين استدعاء الطريقة و جسم الطريقة أثناء التنفيذ اعتمادا على نوع الغرض .

الربط المتأخر يسمى أيضا الربط الديناميكي أو الربط أثناء التنفيذ .

جميع عمليات الربط في جافا هي عمليات ربط متأخر باستثناء الطرق التي تكون من نوعfinal

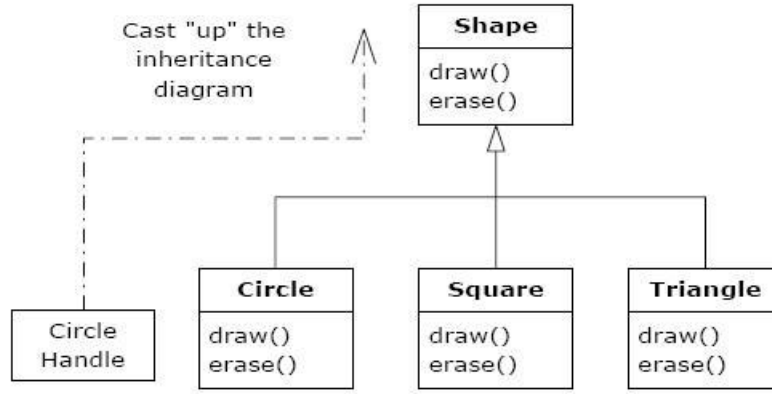
حيث أنه عندما تكون طريقة ما final فأنه لا يمكن إعادة تعريفها عند الأبناء و بالتالي لا يمكن استدعائها إلا من غرض تابع للصف نفسه حصرا .

<u>مثال الأشكال الهندسية :</u>

لدينا صف رئيسي اسمه shape و عدة صفوف مشتقة circle , triangle , square

أي بإمكاننا أن نقول بكل وضوح أن الدائرة هي شكل .. و كذلك بالنسبة لبقية الأشكال

لاحظ الشكل التالي :



عملية الـ upcasting يمكن أن تحدث في التعليمة التالية :

```
Shape s = new Circle();
```

أنشأناغرض من نوع دائرة و أسندناه لغرض من نوع شكل .. و هذه العملية صحيحة لأن الدائرة هي شكل

لنفرض أننا قمنا باستدعاء طريقة موجودة في الصف شكل و التي قمنا بإعادة تعريفها في الصف الابن

```
s.draw();`
```

من المتوقع أنه سيتم استدعاء الطريقة الموجودة في الصف شكل و لكن بالحقيقة سيتم استدعاء الطريقة الخاصة بالصف دائرة بسبب وجود الربط المتأخر (تعدد أشكال)

```
class Shape {
      void draw() {}
      void erase() {}
}
```

```
class Circle extends Shape {
      void draw() {
            System.out.println("Circle.draw()");
      }

      void erase() {
            System.out.println("Circle.erase()");
      }
}

class Square extends Shape {
      void draw() {
            System.out.println("Square.draw()");
      }

      void erase() {
            System.out.println("Square.erase()");
      }
}

class Triangle extends Shape {
      void draw() {
            System.out.println("Triangle.draw()");
      }

      void erase() {
            System.out.println("Triangle.erase()");
      }
}

public class Shapes {
      public static Shape randShape() {
            switch((int)(Math.random() * 3)) {
                  default:
                  case 0: return new Circle();
                  case 1: return new Square();
                  case 2: return new Triangle();
            }
      }

      public static void main(String[] args) {
            Shape[] s = new Shape[9];
            // Fill up the array with shapes:
            for(int i = 0; i < s.length; i++)
                  s[i] = randShape();
            // Make polymorphic method calls:
            for(int i = 0; i < s.length; i++)
                  s[i].draw();
      }
}
```

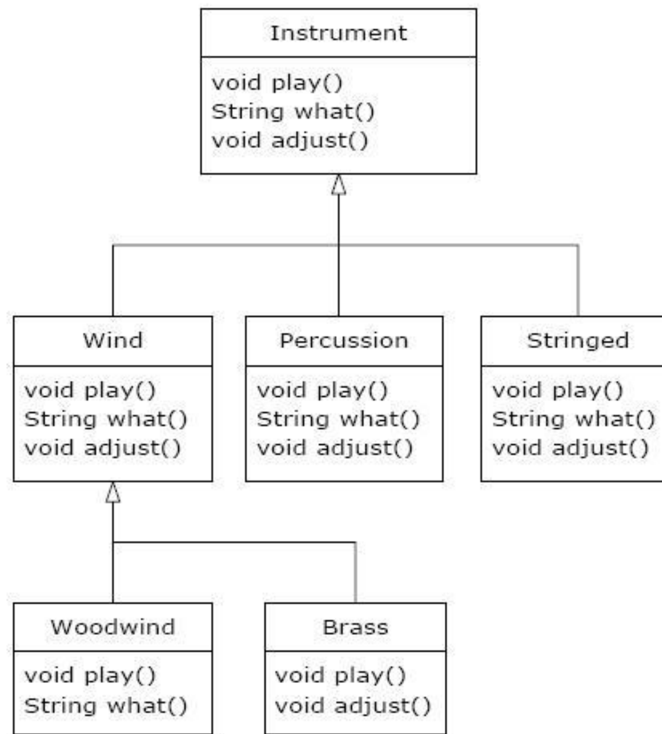
الصف شكل يقوم بتقديم واجهة لكل صف يرث منه . الصفوف المشتقة تقوم بإعادة تعريف طرق الواجهة بشكل يخص كل نوع

تتم مناقشة الكود و تنفيذه عدة مرات خطوة خطوة و ملاحظة كيفية عملية الربط المتأخر ..

التوسعة :

بالعودة لمثال الآلات الموسيقية و نتيجة لتعددية الأشكال يمكننا إضافة صفوف جديدة إلى النظام من دون تغيير الطريقة tune() . لذلك نقول عن البرنامج أنه قابل للتوسعة حيث يمكننا إضافة أشياء جديدة عن طريق الوراثة من الصف الأساسي (الأب)

لنفرض أننا أضفنا للنظام صفوف جديدة ... سيكون المخطط كمايلي :



جميع هذه الصفوف الجديدة تعمل بشكل صحيح من دون تغيير الطريقةtune()

هنا الكود الموافق للمخطط السابق :

```
import java.util.*;
```

نلاحظ أن الطريقة tune () مع كل التغييرات و الاضافات قد تم تجاهلها (أي لم يتم التطرق لجسمها) و هي لا تزال تعمل بشكل صحيح و هذا هو بشكل أساسي ما تؤمنه تعددية الأشكال حيث أن التغييرات التي تتم على الكود لا تؤثر على الأجزاء التي لا نريد لها أن تتأثر .. فـ مبدأ تعددية الأشكال "هو فصل الأشياء التي لا تتغير عن الأشياء التي تتغير"

الطريقة `what()` تعيد سلسلة باسم الصف .

التحميل الزائد و إعادة التعريف :

الفكرة الأساسية في تعددية الأشكال أن طرائق الواجهة الخاصة بالصف الأساسي يتم إعادة تعريفها و ليس تحميلها بشكل زائد .. و عند القيام بعملية التحميل الزائد يلتغي مفهوم تعدد الأشكال

مثال :

```
class NoteX {
      public static final int
                  MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

class InstrumentX {
      public void play(int N) {
            System.out.println("InstrumentX.play()");
      }
}

class WindX extends InstrumentX {
      public void play(NoteX n) {
            System.out.println("WindX.play(NoteX n)");
      }
}

public class WindError {
      public static void tune(InstrumentX i) {
            // ...
            i.play(NoteX.MIDDLE_C);
      }

      public static void main(String[] args) {
            WindX flute = new WindX();
            tune(flute); // Not the desired behavior!
      }
}
```

من الخرج نلاحظ أنه تم استدعاء الطريقة الخاصة بالصف الأب .. و نلاحظ هنا أنه تم الغاء مفهوم تعددية الأشكال .

الصفوف و الطرائق المجردة (Abstract classes and methods) :

في مثال الآلات الموسيقية .. الطرائق الموجودة في الصف الأب هي طرائق زائفة ، حيث أنه إذا قمنا باستدعاء أي منها فإن النتائج التي نحصل عليها ستكون خاطئة و ذلك بسبب ان الصف الأب مهمته فقط تأمين واجهة للصفوف المشتقة

لتجنب الأخطاء التي يمكن الحصول عليها في البرنامج السابق نقوم بجعل الصف الأب هو صف مجرد .

نقوم بإنشاء صف مجرد فقط عندما نريد معالجة مجموعة من الصفوف من خلال واجهة مشتركة . تكون هذه الواجهة المشتركة معرفة في الصف المجرد .

إذا كان لدينا صف مجرد فإن إنشاء أغراض من هذا الصف هي عملية بلا معنى و ذلك كون الصف المجرد يعبر فقط عن واجهة .. و لذلك المترجم يمنع عملية تهيئة أغراض من نوع الصف المجرد ..

يمكن التصريح عن غرض و لكن لا يمكن تهيئته بنفس نوع الصف المجرد .. إلا أنه يمكن القيام بعملية التهيئة عن طريق صف ابن للصف المجرد (upcasting).

تؤمن لغة جافا طريقة لجعل طريقة ما مجردة أي غير مكتملة .. فقط نقوم بالتصريح عن الطريقة و لكن لا نكتب جسم الطريقة .

أي أن الصف المجرد يمكن أن يحتوي طرائق مجردة و غير مجردة

إذا كان لدينا صف يحوي طريقة مجردة فيجب أن يكون الصف مجرد (المترجم يجبرنا على ذلك)

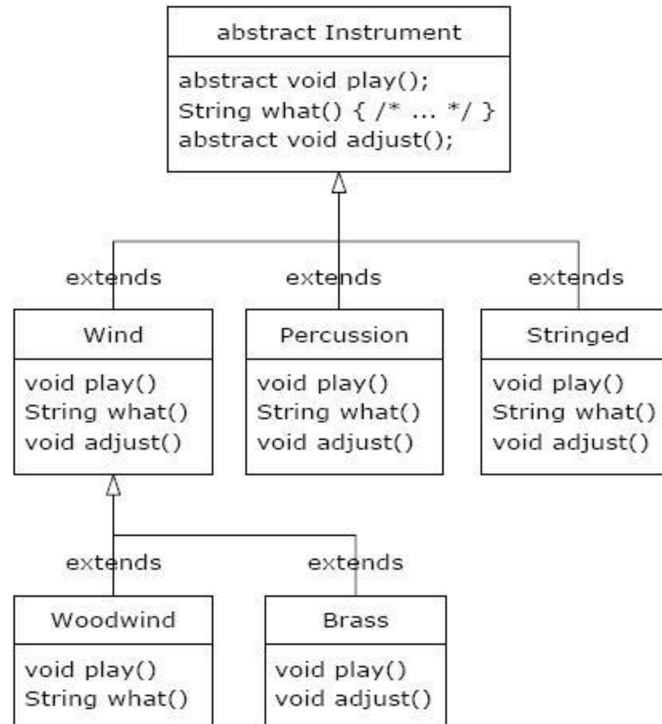
الآن .. عندما نقوم بالوراثة من صف مجرد يجب علينا إعادة تعريف جميع الطرق المجردة الموجودة في الصف الأب

و المترجم يجبرنا على القيام بذلك

نعود لمثال الآلات الموسيقية ... يمكن تحويل الصف Instrument إلى صف مجرد و نقوم بجعل بعض الطرق فيه مجردة

و الصفوف الأخرى ترث منه و تقوم بإعادة تعريف الطرائق المجردة

و بالتالي لدينا الشكل التالي :



و هنا الكود الموافق للشكل التالي :

```
abstract class Instrument {
      public abstract void play();

      public String what() {
            return "Instrument";
      }

      public abstract void adjust();
}

class Wind extends Instrument {
      public void play() {
            System.out.println("Wind.play()");
      }

      public String what() { return "Wind"; }

      public void adjust() {}
}

class Percussion extends Instrument {
      public void play() {
            System.out.println("Percussion.play()");
      }
```

```
	public String what() { return "Percussion"; }

	public void adjust() {}
}

class Stringed extends Instrument {
	public void play() {
		System.out.println("Stringed.play()");
	}

	public String what() { return "Stringed"; }

	public void adjust() {}
}

class Brass extends Wind {
	public void play() {
		System.out.println("Brass.play()");
	}

	public void adjust() {
		System.out.println("Brass.adjust()");
	}
}

class Woodwind extends Wind {
	public void play() {
		System.out.println("Woodwind.play()");
	}

	public String what() { return "Woodwind"; }
}

public class Music4 {
	// Doesn't care about type, so new types
	// added to the system still work right:

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

	static void tuneAll(Instrument[] e) {
		for(int i = 0; i < e.length; i++)
			tune(e[i]);

	}

	public static void main(String[] args) {
		Instrument[] orchestra = new Instrument[5];
		int i = 0;
```

```
            // Upcasting during addition to the array:
            orchestra[i++] = new Wind();
            orchestra[i++] = new Percussion();
            orchestra[i++] = new Stringed();
            orchestra[i++] = new Brass();
            orchestra[i++] = new Woodwind();
            tuneAll(orchestra);
      }
}
```

بإمكاننا أن نرى بوضوح أن الصف الأب لم يحدث به أي تغيير ..

من المفيد جدا إنشاء صفوف و طرائق مجردة لأنها تجعل تجريد الصف أكثر وضوحا و تخبر كلا من المستخدم و المترجم ما هو الهدف من إنشاء هكذا صفوف و طرائق (فقط واجهات للأبناء) ..

<u>البواني و تعددية الأشكال :</u>

سنناقشها من خلال الأمور التالية :

<u>أولا – ترتيب استدعاء البواني :</u>

ترتيب استدعاء البواني تمت مناقشته سابقا .. باني الأب يتم استدعاؤه دائما في بواني الأبناء و ذلك حسب ترتيب الشجرة الوراثية للصفوف ..

السبب في ذلك هو أن الصف الابن يمكنه الوصول لحقوله الخاصة و لكن لا يمكنه الوصول لحقول الصف الأب (إذا كانت private) ... فقط باني الأب يمكنه الوصول و تهيئة حقوله الخاصة..

و كما ذكرنا .. إذا لم يتم استدعاء الباني صراحة فإن المترجم يقوم باستدعاء الباني الافتراضي .. و إذا لم يكن هناك باني افتراضي فإن المترحم يجبرنا على استدعاء أحد البواني الموجودة ..

المثال التالي يوضح تأثير الوراثة و التركيب و تعددية الأشكال على ترتيب استدعاء البواني .

<u>مثال :</u>

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

class Bread {
      Bread() { System.out.println("Bread()"); }
}

class Cheese {
      Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
```

```
      Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
      Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
      PortableLunch() {
            System.out.println("PortableLunch()");
      }
}

public class Sandwich extends PortableLunch {
      Bread b = new Bread();
      Cheese c = new Cheese();
      Lettuce l = new Lettuce();

      Sandwich() {
            System.out.println("Sandwich()");
      }

      public static void main(String[] args) {
            new Sandwich();
      }
}
```

هذا المثال يقوم بإنشاء صف معقد من صفوف أخرى ..

الصف الأهم هو الصف Sandwich و الذي يعكس ثلاث مستويات وراثة و يحتوي على ثلاث حقول .

من خرج البرنامج نلاحظ أن عملية استدعاء البواني تتم كمايلي :

- **يتم استدعاء باني الأب .. و هذه العملية عودية .. أي يتم الصعود لأعلى حتى يتم أولا استدعاء باني الأب الأول**
- **يتم تهيئة الحقول الاعضاء بحسب ترتيب تعريفهم**
- **يتم تنفيذ جسم باني الصف الابن**

ثانيا – سلوك الطرائق متعددة الأشكال داخل البواني :

لنفرض أننا قمنا باستدعاء طريقة متعددة الشكل داخل الباني .. ما الذي يمكن أن يحدث ؟؟؟؟!!!

بالنسبة للطرائق العادية .. عملية استدعاء طريقة متعددة الشكل يتم تحديدها في وقت التنفيذ

أما بالنسبة للبواني فإنه يتم استدعاء الطريقة التي تم إعادة تعريفها في الابن .. و لكن ذلك يبدو غريبا

مهمة الباني هي جلب الغرض للحياة (جعله قابل للاستخدام) ..

داخل باني الابن يتم استدعاء كأول تعليمة باني الأب .. و بذلك فإنه فقط غرض الأب تمت تهيئته .

إذا قمنا باستدعاء طريقة متعددة الأشكال داخل باني الابن (او باني الأب) فإنه يتم استدعاء الطريقة التي تم إعادة تعريفها مع أنه من الممكن أن يكون غرض الابن ليس مهيئا بعد (ممكن أن يتم استدعاء طريقة متعددة الأشكال بعد استدعاء باني الأب مباشرة أو قبل انتهاء جسم باني الابن)

فكيف سيكون ترتيب التهيئة ؟؟؟؟

مثال :

```
abstract class Glyph {
      abstract void draw();

      Glyph() {
            System.out.println("Glyph() before draw()");
            draw();
            System.out.println("Glyph() after draw()");
      }
}

class RoundGlyph extends Glyph {
      int radius = 1;

      RoundGlyph(int r) {
            radius = r;
            System.out.println(
                  "RoundGlyph.RoundGlyph(), radius = "
                  + radius);
      }

      void draw() {
            System.out.println(
                  "RoundGlyph.draw(), radius = " + radius);
      }
}

public class PolyConstructors {
      public static void main(String[] args) {
            new RoundGlyph(5);
      }
}
```

في الصف Glyph الطريقة draw() مجردة .. أي أنها مصممة لكي تتم عملية إعادة تعريف لها

و لكن داخل الباني يتم استدعاء الطريقة draw() ..

من الخرج نلاحظ أن ترتيب التهيئة الذي ذكرناه في الفقرة الماضية ليس صحيحا بشكل كامل

و الترتيب الصحيح هو :

- **يتم تهيئة جميع الحقول بالقيمة الثنائية 0**

- يتم استدعاء باني الأب بالطريقة التي ذكرناها سابقا
- تهيئة الحقول في الصفوف المشتقة
- تنفيذ أجسام البواني للصفوف المشتقة

: DownCasting

عملية الـ upcasting هي الانتقال إلى الأعلى في الشجرة الوراثية و هي دوما آمنة

في حين عملية الـ downcasting هي الانتقال إلى الأسفل في الشجرة الوراثية عملية ليست آمنة

حيث اننا على سبيل المثال نعلم أن كل دائرة هي شكل و يمكن لغرض من نوع دائرة أن يقوم بكل ما يقوم به غرض شكل

إلا أنه ليس كل شكل هو دائرة .. فمن الممكن أن يكون مربع أو مستطيل أو أو ... الخ

لحل هذه المشكلة لدينا مجموعة إجراءات تضمن ان يكون القسر صحيح

ففي لغة جافا جميع عمليات الفسر يتم فحصها في زمن التنفيذ في حال كان النمط خاطئ فإنه يحصل استثناء أثناء التنفيذ من نوع ClassCastException ..

مثال :

```
import java.util.*;

class Useful {
      public void f() {}
      public void g() {}
}

class MoreUseful extends Useful {
      public void f() {}
      public void g() {}
      public void u() {}
      public void v() {}
      public void w() {}
}

public class R {
      public static void main(String[] args) {
            Useful[] x = {
                        new Useful(),
                        new MoreUseful()
            };
            x[0].f();
            x[1].g();
            // Compile-time: method not found in Useful:
            //! x[1].u();
            ((MoreUseful)x[1]).u(); // Downcast
```

```
            ((MoreUseful)x[0]).u(); // Exception thrown
        }
}
```

يتم شرح كيف تمت توسعة الواجهة و بالتالي عملية القسر خاطئة

ساهم بنشر الكتاب ولك الأجر والثواب إن شاء الله

لا تنسوني من صالح دعائكم

تم بحمد الله