مراجعة للبرمجة الشيئية بلغة جافا

Objects & Classes

الكائن (Object)

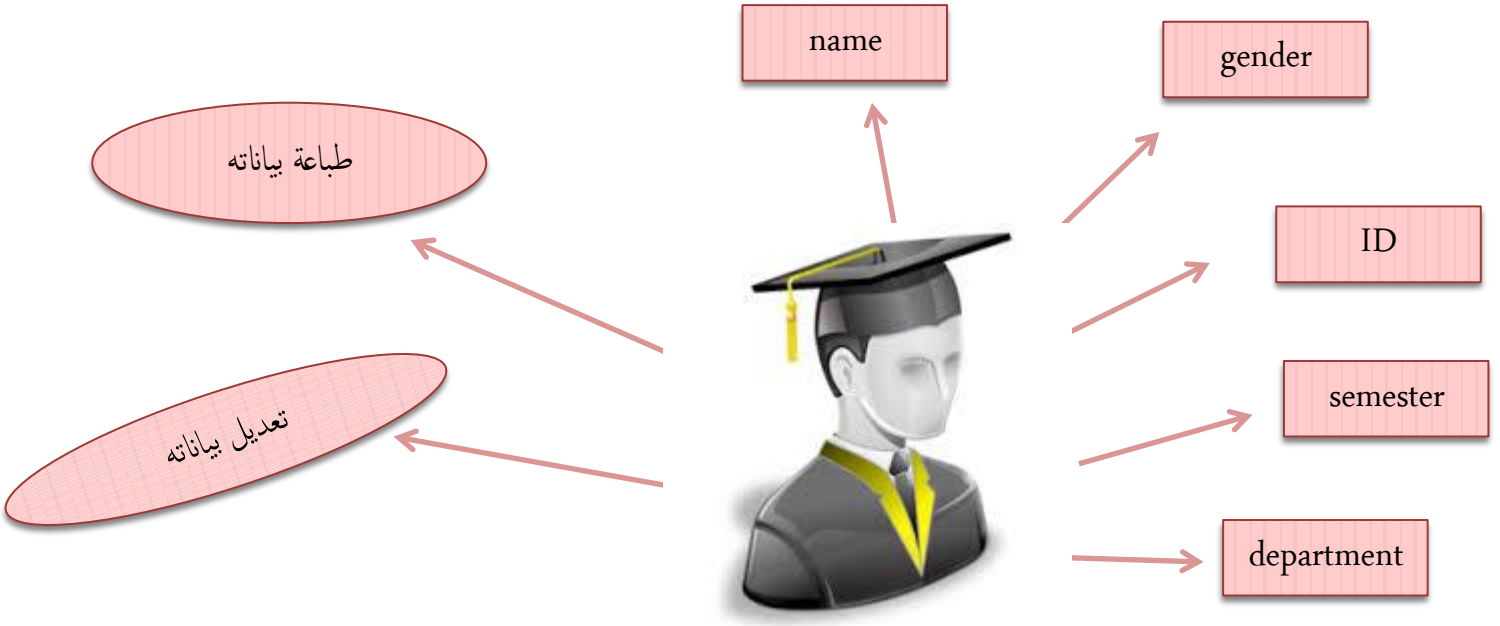
Objects يتم أستخدامها في تمثيل الاشياء فمثلا يمكن أستخدام Object في تمثيل طالب في مدرسة أو موظف في شركة.

كل Object له:

- **Properties** تقوم بوصف خصائصه.
- **Behavior** يحدد ماذا يستطيع أن يفعل أو يفعل به.

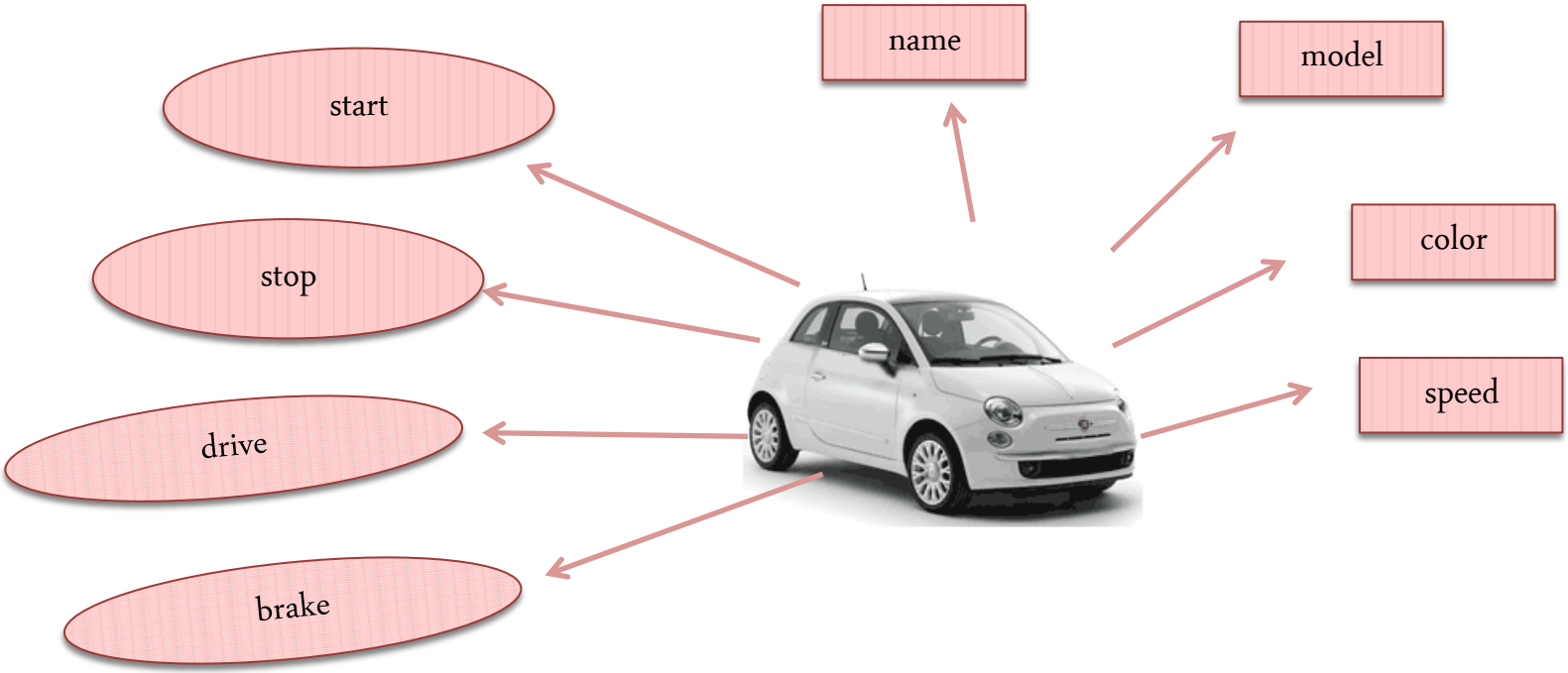
الكائن (Object)

يمكن أستخدام Object في تمثيل طالب في مدرسة حسب الصفات والسلوك التالية.



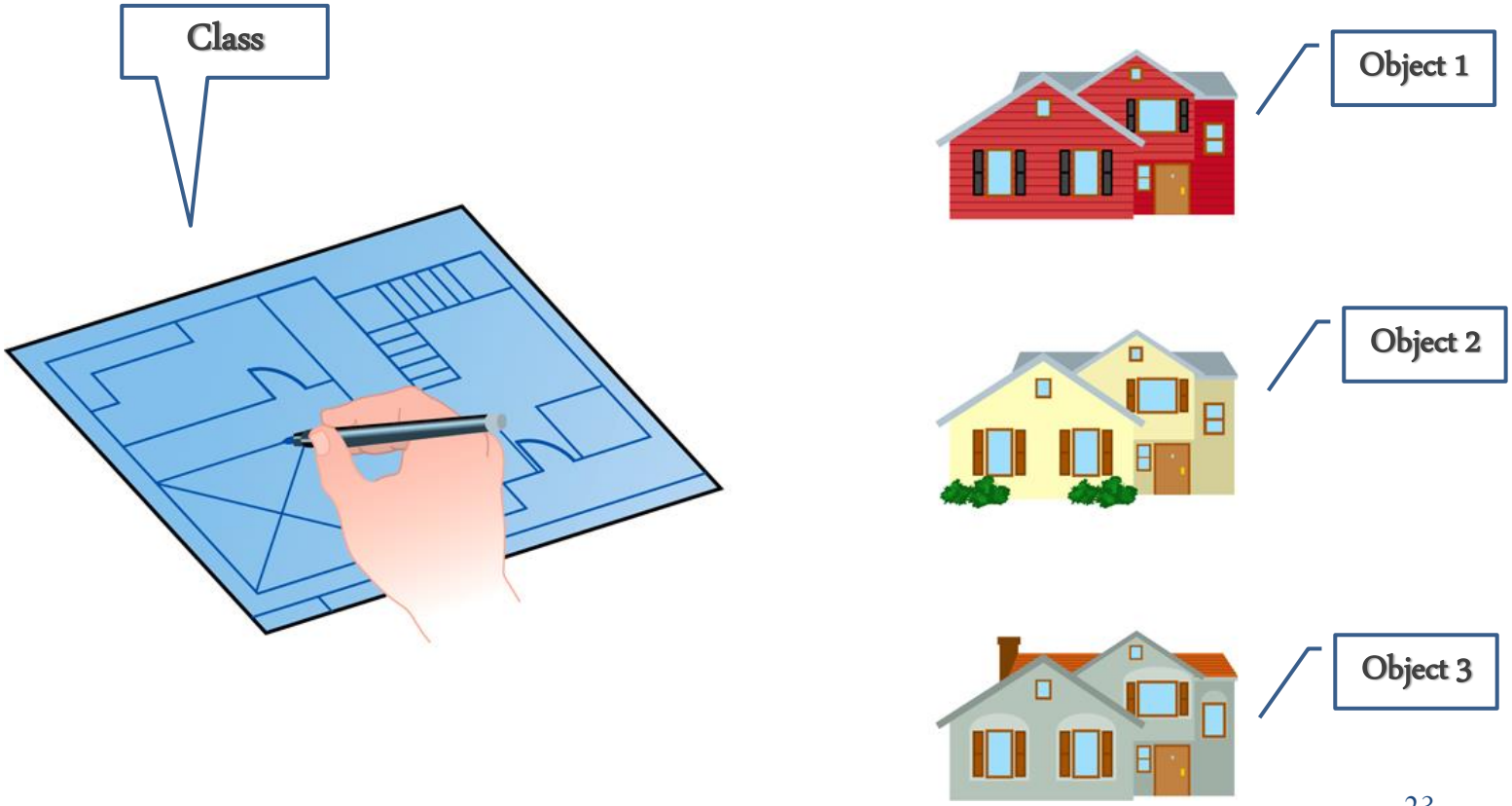
الكائن (Object)

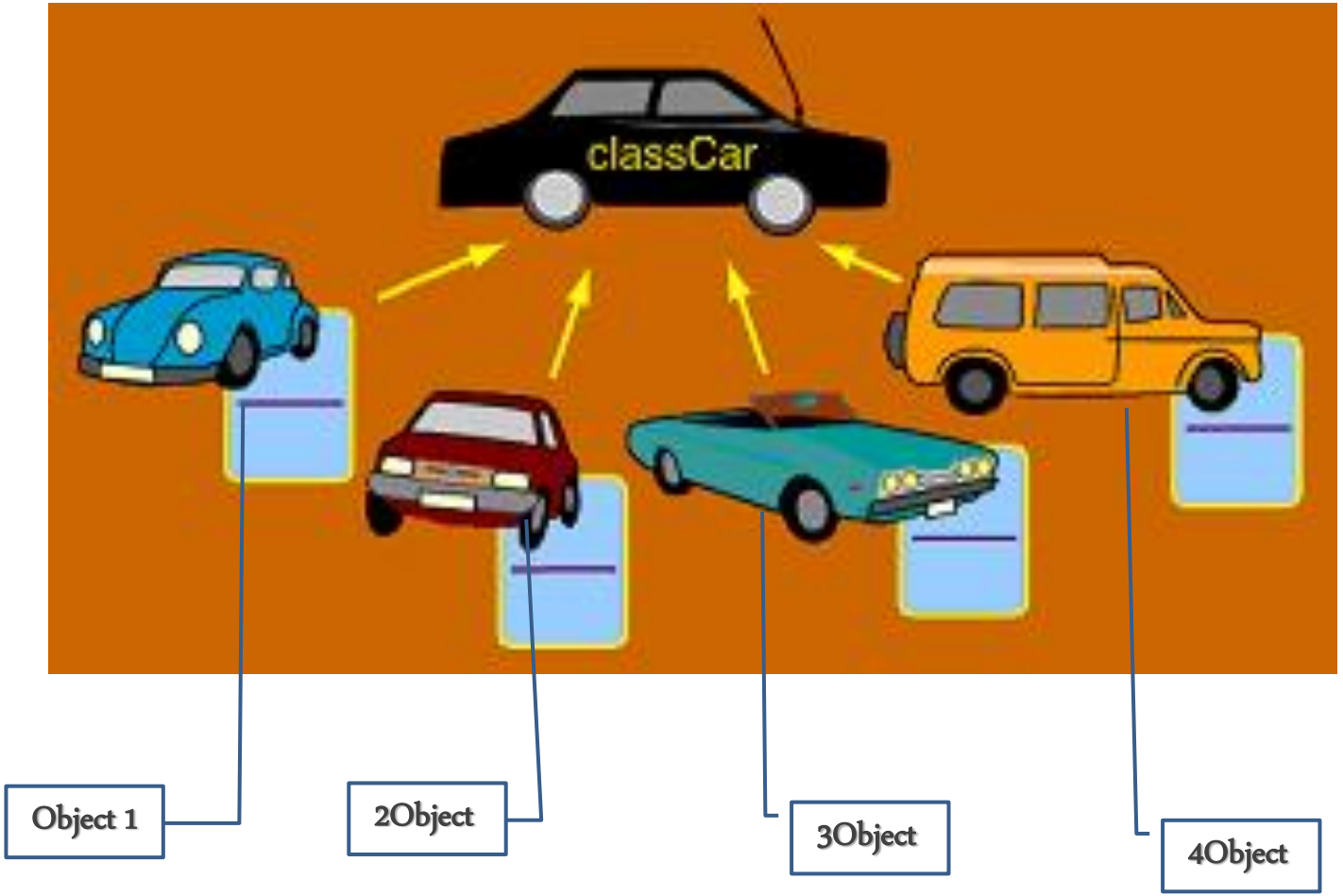
يمكن أستخدام Object في تمثيل سيارة حسب الصفات والسلوك التالية.

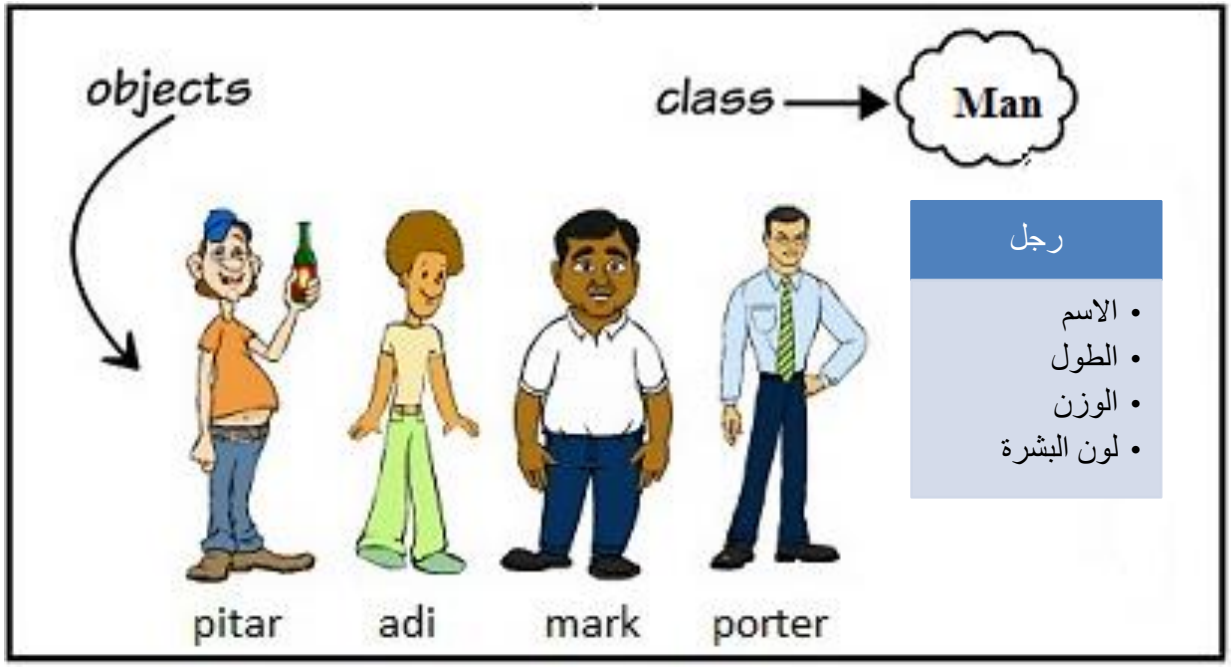


الفئة (Class)

Class هي القالب الذي عن طريقه يتم أنشاء Objects وهو يستخدم المتغيرات في تعريف الصفات (properties) والدوال في تحديد السلوك (behavior).



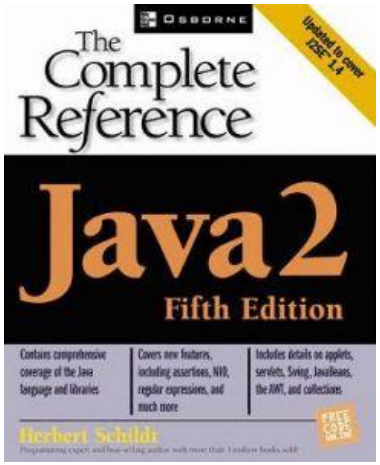

classCar
Object 1
2Object
3Object
4Object


objects
class
Man
رجل
الاسم
الطول
الوزن
لون البشرة
pitar
adi
mark
porter
Object 1
Object 2
Object 3
Object 4

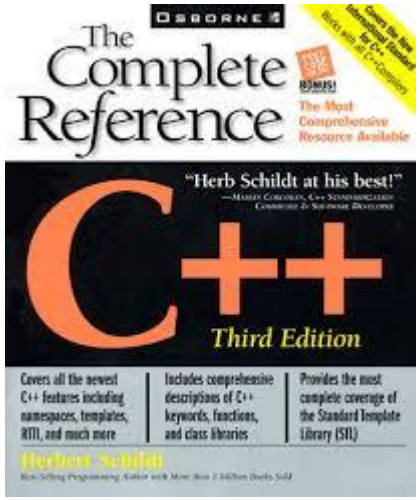
Class

الكتاب

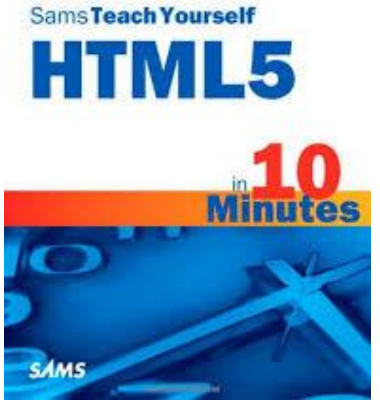
- العنوان
- المؤلف
- عدد الصفحات
- تاريخ النشر



Object 1



Object 2



Object 3

مثال:

Class

- name
- ID
- Semester
- department
- gender

Object 1

- أحمد
- 3245435
- 5
- الحاسوب
- ذكر

Object 2

- سامية
- 686656
- 3
- الاحياء
- أنثى

Object 3

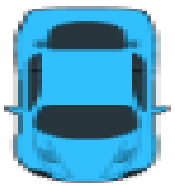
- خالد
- 66753276
- كيمياء
- 7
- ذكر

مثال:

Class
• name • model • color • speed

Object 1

Object 2

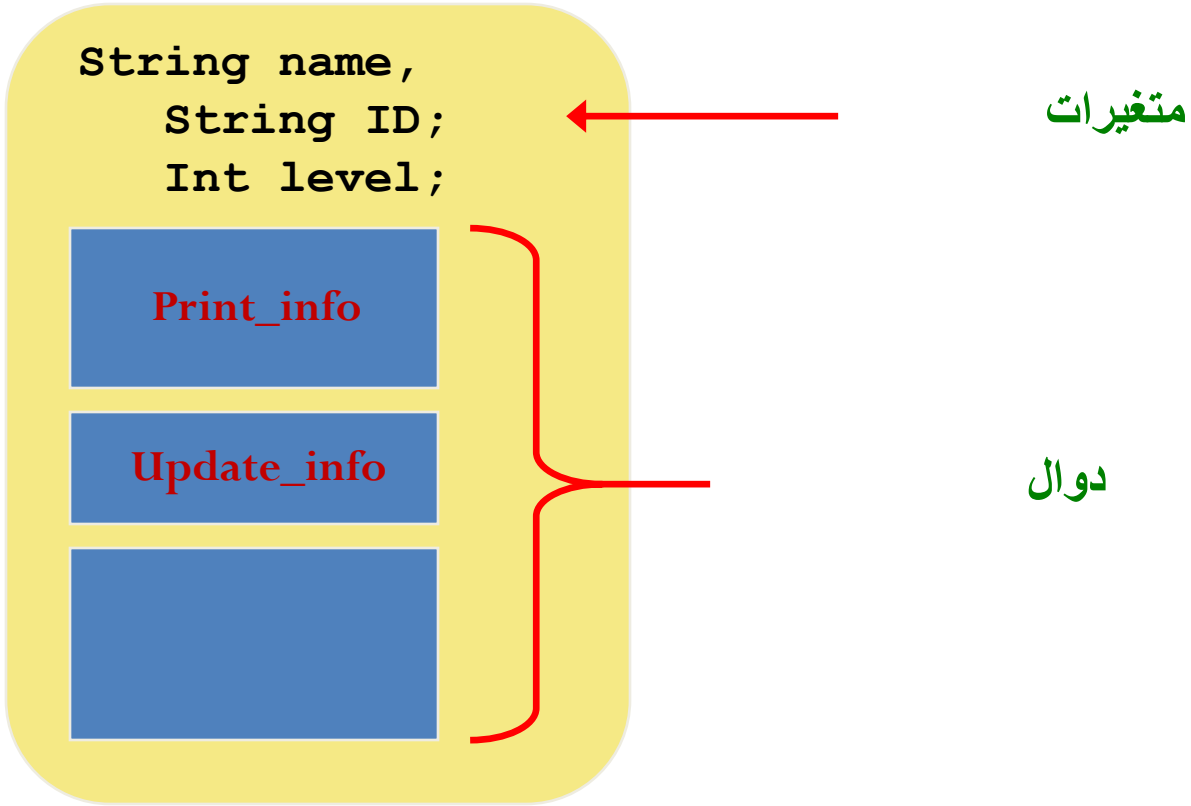


Object 3

Object 1	Object 2	Object 3
Fiat • 1968 • yellow • 160 •	Kia • 2010 • Orange • 220 •	Ferari • 2016 • blue • 320 •

كتابة Class

- Class تحتوي عل تعريف المتغيرات والدوال.



مثال:

```
public class Student {
    long ID;
    String name;
    short semester;
    short department;

    void printDetails(){
        System.out.println("ID: "+ID);
        System.out.println("name: "+name);
        System.out.println("semester: "+semester);
        System.out.println("department: "+department);
     }

    void updateDetails(long id, String na, short sm, short dep){
        ID =id;
        name =na;
        semester = sm;
        department= dep;
     }

}
```

الوصول إلى عناصر Object

- للوصول إلى المتغيرات الموجودة في Object نستخدم

objectName.variableName

- للوصول إلى الدوال الموجودة في Object نستخدم

objectName.methodName

مثال:

```
public class Circle {

    double radius;
    double getArea(){
        double area = 3.14* radius*radius;
        return area;
    }

}
```

```
public class circleExample {
    public static void main(String[] args) {
        Circle c1 = new Circle();
        c1.radius = 5.5;
        System.out.println("Area1 = " +    c1.getArea());
    }
}
```

Constructors

هي عبارة عن دالة تحمل نفس أسم Class يتم أستدعائها عند أنشاء Object لتستخدم في تمرير البيانات إلى Object . Class ممكن أن تحتوي على أكثر من Constructor .

Constructor

```
public class Circle {
   Circle(double r){
      radius = r;
   }
   double radius;
   double getArea(){
      double area = 3.14* radius*radius;
      return area;
   }

}
```

التحكم في الوصول لعناصر Class

تستخدم جافا ثلاث مستويات في الوصول لعناصر Class وهي موضحة في الجدول التالي:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
private	Y	N	N	N

مثال:

لايمكن الوصول إليه الا من داخل Class

يمكن الوصول إليهم من خارج Class

```
public class Circle_A {

   private double radius;

   public void setRadius(double radius) {
      this.radius = radius;
   }

   public double getRadius() {
      return radius;
   }

   double getArea() {
      double area = 3.14 * radius * radius;
      return area;
   }

}
```

مقدمة في البرمجة الشيئية

البرمجة الشيئية

Object Oriented Programming

نشأت فكرة البرمجة الشيئية من الحاجة إلي جعل لغات البرمجة أكثر كفأة فجأءت فكرة محاكاة الواقع من خلال التعامل مع الأشياء (objects) ومن هنا ظهرت لغات البرمجة الشيئية.

فوائد البرمجة الشيئية :

- القدرة على معرفة مكان الأخطاء في البرنامج بسهولة .
- القدرة على تطوير البرنامج بسهولة مع الوقت .
- القدرة على إعادة استخدام الكثير من أجزاء البرنامج لتطوير برامج أخرى .
- القدرة على توزيع العمل في برنامج واحد ضخم على أكثر من مبرمج.

مصطلحات خاصة بالبرمجة الشيئية

- **Class**

عبارة عن Template تستخدم في أنشاء Objects .

- **Object**

عبارة عن Entity لها properties تستخدم في التعريف بحالتها و Methods للتعبير عن وظائفها بالاضافة إلى Events توضح التغير في حالتها.

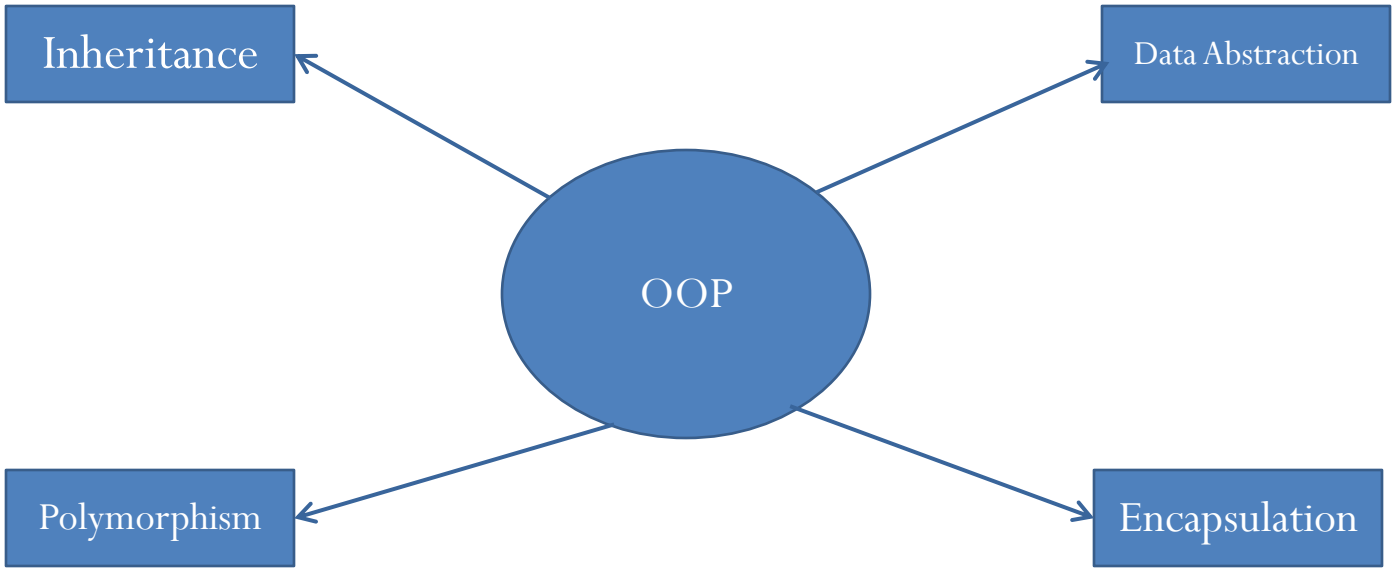
- **Object Oriented Application**

عبارة عن مجموعة من Objects التي تتعامل مع بعضها البعض من خلال تبادل الرسائل في بيئة تتحكم في هذا التعاون وتنظمه.

- **Components**

عبارة عن مكون برمجي جاهز يمكن أستخدامه في تطوير البرامج ويمكن أستبداله بمكون برمجي أخر دون أن يوثر في البرنامج.

مميزات البرمجة الشيئية



فوائد البرمجة الشيئية

1. **التغليف (Encapsulation)**

هي عملية تجميع كل الخصائص properties و العمليات Methods في وحدة واحدة لا يمكن الوصول إليها إلاّ عن طريق الكائن .

Encapsulation يستخدم في وضع البيانات والدوال التي ستشتغل عليها في مكان واحد.

مثال:

اذا كان لديك في البيت شئ تستعمله باستمرار مثل التلفزيون والذي تستعمله بشكل يومي ، اوصف بأختصار كيف يعمل ، ثم قم بوصف مكوناته الداخلية بكل تفاصيلها التقنية.

وصف عمل التلفزيون أسهل بكثير من وصف مكوناته الداخلية وطريقة عملها. معظم الناس لايعرفون المكونات الداخلية للاجهزة وكيفية عملها لكن عذا لم يمنعهم من أستخدامها يومياً. (مثال أخر المفتاح الكهربائي)

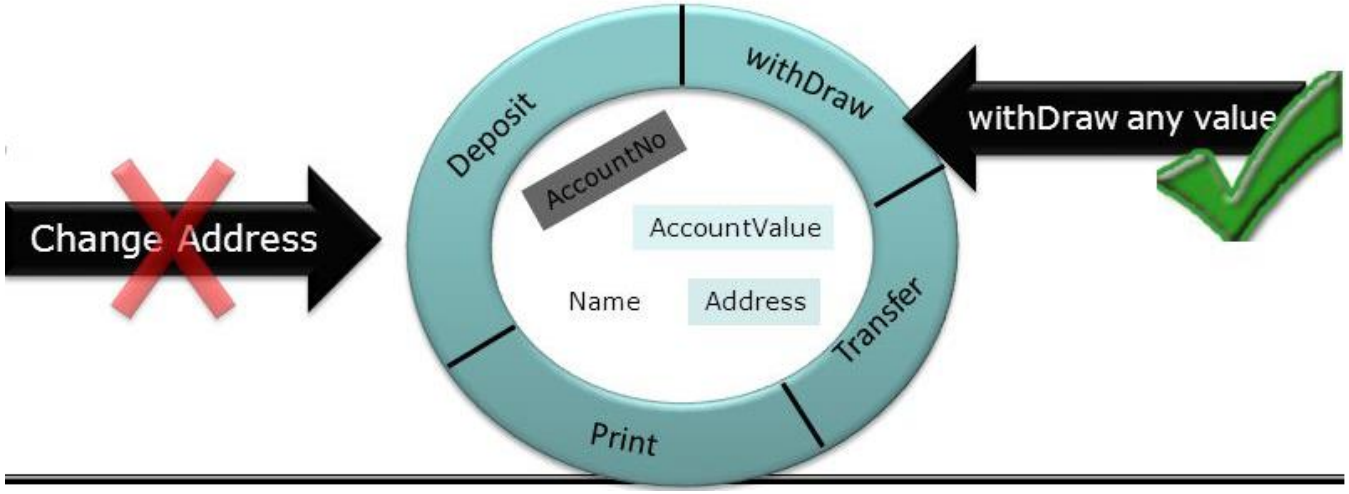
2. إخفاء البيانات Data Hiding

وهي خاصية ناتجة عن تغليف البيانات . وتعني إضافة مستوى حماية معين إلى البيانات حتى نمنع الوصول الخطأ إليها .

مثال:

في هذا المثال لايمكن الوصول إلى ماموجود في الحساب وتغيره الا عن طريق الدوال withDraw, Deposit, Transfer.

بينما لايمكن الوصول إلى العنوان و تغيير لانه لاتوجد دالة تقوم بذلك وهذا مايعرف ب data hiding .



3. التجريد (Data Abstraction)

وهي عملية كشف المعلومات الضرورية ب Object للعالم الخارجي وأخفاء الغير ضروري معرفته للاخرين.

مثال:

تخيل ان لديك منزل وأردت ان تقوم بإعطائه لمدة أسبوع لصديق، اذكر ثلاث أشياء رئيسية يجب ان تذكرها لصديقك ومهم ان يعرفها. ثم اذكر ثلاث أمور اخرى غير مهمة وليس من الضروري ان تذكرها له؟

الأشياء الضرورية الثلاث ممكن ان تكون :	الأشياء الثلاث الغير ضرورية ممكن ان تكون:
1- العنوان	1- لون المنزل
2- عدد الغرف ونوعها	2- عدد التلفزيونات
3- مكان مفتاح تشغيل الكهرباء والماء	3- عدد المقاعد

- *Data Abstraction* يسمح لنا بالتركيز على الأشياء المهمة مثل العنوان في المنزل بدون الدخول في التفاصيل الاخرى الغير مهمة مثل لون المنزل.

4. الوراثة Inheritance

وهي أن يرث صنف ما الخصائص والعمليات الموجودة في الصنف الآخر مما يساعد على إعادة الاستخدام الأصناف التي تم إنشاؤها من قبل المستخدم .

Inheritance تسمح لنا بتحديد بعض الصفات والعمليات الخاصة ب object بالاضافة إلى أنشاء نوع خاص من هذا object له بعض الصفات الخاصة ويورث كل صفات object الذي انشأ منه.

مثال:

أذكر ثلاث وظائف للناس التي تشتغل في المستشفى ثم أذكر ثلاث او أثنين من أصناف الموظفين الذين يشغلون كل وظيفة.

ممكن ذكر ثلاث وظائف بالمستشفى وهي طبيب ممرض وعامل النظافة. وهذه الوظائف الرئيسية ممكن أن تقسم إلى عدة وظائف فرعية مثل:

• طبيب • طبيب قلب • طبيب جراح • طبيب تخدير	• ممرض • ممرض للتخدير • ممرض عام	• عامل النظافة • مشرف عمال • عامل نظافة

مثال:

أذكر شي يقوم به كل الاطباء وشيء لايقوم به الا الطبيب الجراح.

- الاطباء كلهم لديه المعرفة بالادوية وكيفية الكشف عن المرض ووصف الادوية المناسبة للمريض.
- الطبيب الجراح يعرف كيف يتعامل مع أدوات الجراحة وكيفية أستخدامها في أجراء العمليات الجراحية.

طبقا لما ذكر كل الجراحين أطباء لانهم لديهم المعرفة بالادوية وكيفية الكشف عن المرض ووصف الادوية المناسبة للمريض. ولكن ليس كل الاطباء جراحين لان ليس كل الاطباء يستطيعون أجرا العمليات الجراحية.

نستنتج من هذ أن الطبيب الجراح قد أخد بعض الصفات الخاصة بالطبيب بلاضافة إلى بعض الصفات الخاصة به.

5. تعدد الأشكال polymorphisms

تسمح خاصية تعدد الأشكال لل Object أن تكون له أشكال مختلفة تقدم نفس الوظيفة ولكن تنفيذها يتم بطرق مختلفة.

مثال:

الاستاذ في المدرسة ممكن أن يكون بأكثر من صورة أستاذ رياضة أو أستاذ لغة أنجليزية وفي كلا الحالتين له نفس الوظيفة وهي التدريس ولكنه يقوم بها بطريقة مختلفة عندما يكون أستاذ رياضة عن الطريقة عندما يكون أستاذ لغة أنجليزية .

مثال:

مجموعة الحيونات في الصورة جميعها لها نفس الوظيفة هنا "Speak" ولكن كل منها تقوم بها بطريقة مختلفة حسب نوعها (كلب، قطة، بطة) مع أنها جميعها تحت التصنيف **حيوان**.

الوراثة
Inheritance

مفهوم الوراثة

- مفهوم الوراثة في الحياة هو أن يستمد نوع معين صفات و سلوك من نوع آخر, فمثلا يستمد الطفل من صفات أبيه اللون , الطول و من سلوك أبيه مهارة الرسم و هكذا ... و بالطبع قد يتمتع الابن بصفات جديدة لم تكن موجودة في أبيه.

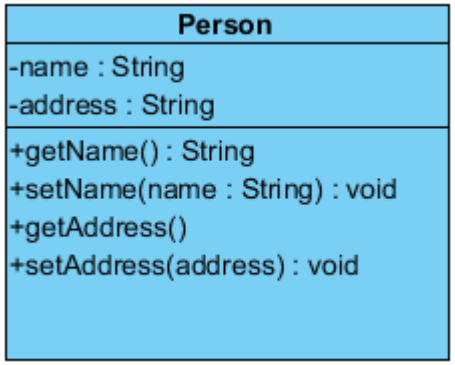
- و في البرمجة يوجد المفهوم ذاته فال class A قد يستمد صفات class B و صفات آخرى إضافية.

مميزات الوراثة

- إعادة استخدام الأكواد. ((Reusability
- تقليل الوقت.
- تحقيق لمبدأ تجزئة المشكلة ((Modularization

مثال:

- أذا رغبنا في تصميم class لتمثيل Person يمكن تصميمها على الشكل التالي

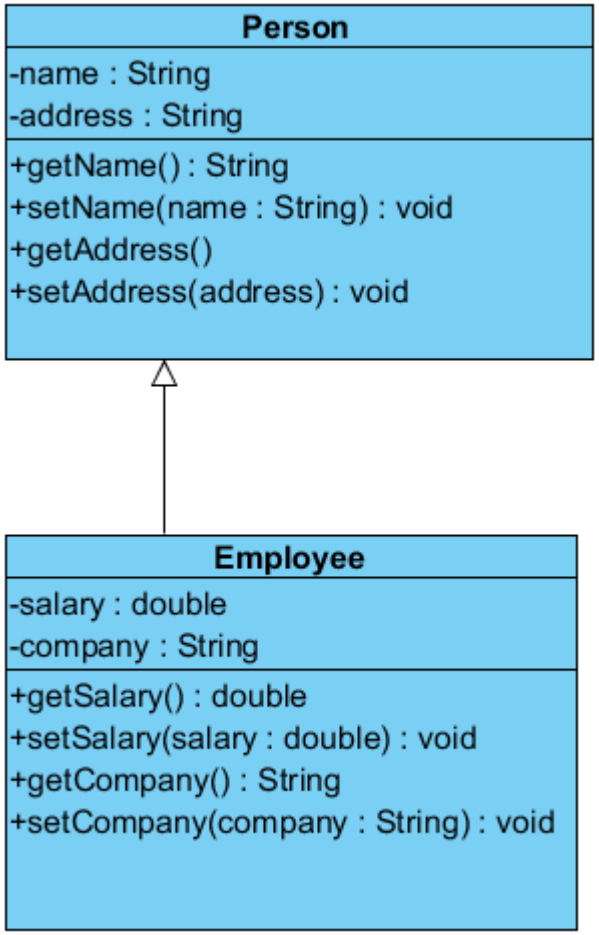


- وهي تحتوي على كل من خاصيتي name و address لتمثل أسم الشخص وعنوانه والتي يمكن الوصول إليها عن طريق مجموعة دوال set و get.

- أذا مارغبنا في تصميم class جديدة لتمثل موظف والذي لديه الخصائص التالي:
  - الاسم
  - العنوان
  - المرتب
  - أسم الشركة التي يعمل بها

- بدلا من تصميم Employee class من الصفر يمكن الاستعانة ب Person class للحصول على الخصائص المشتركة مع Employee class .
- يتحقق ذلك من خلال أستخدام ميزة الوراثة (Inheritance) التي تقدمها البرمجة الشيئية

- الشكل التالي يبين لنا تمثيل مبدأ الوراثة بأستخدام class diagram .
- هنا Employee class ورث ماموجود في Person class .
- ثم أضاف بعض الخصائص والعمليات الخاصة به.



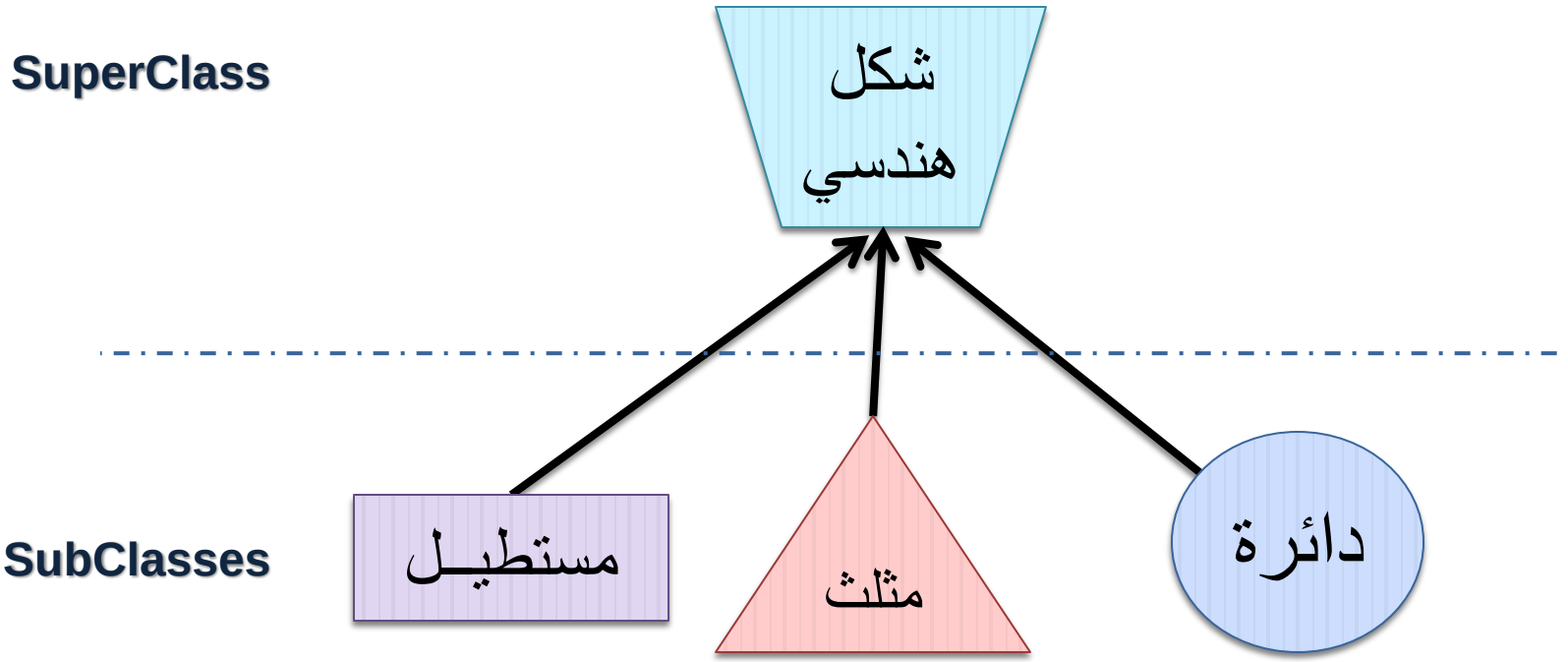
مفاهيم متعلقة بالوراثة

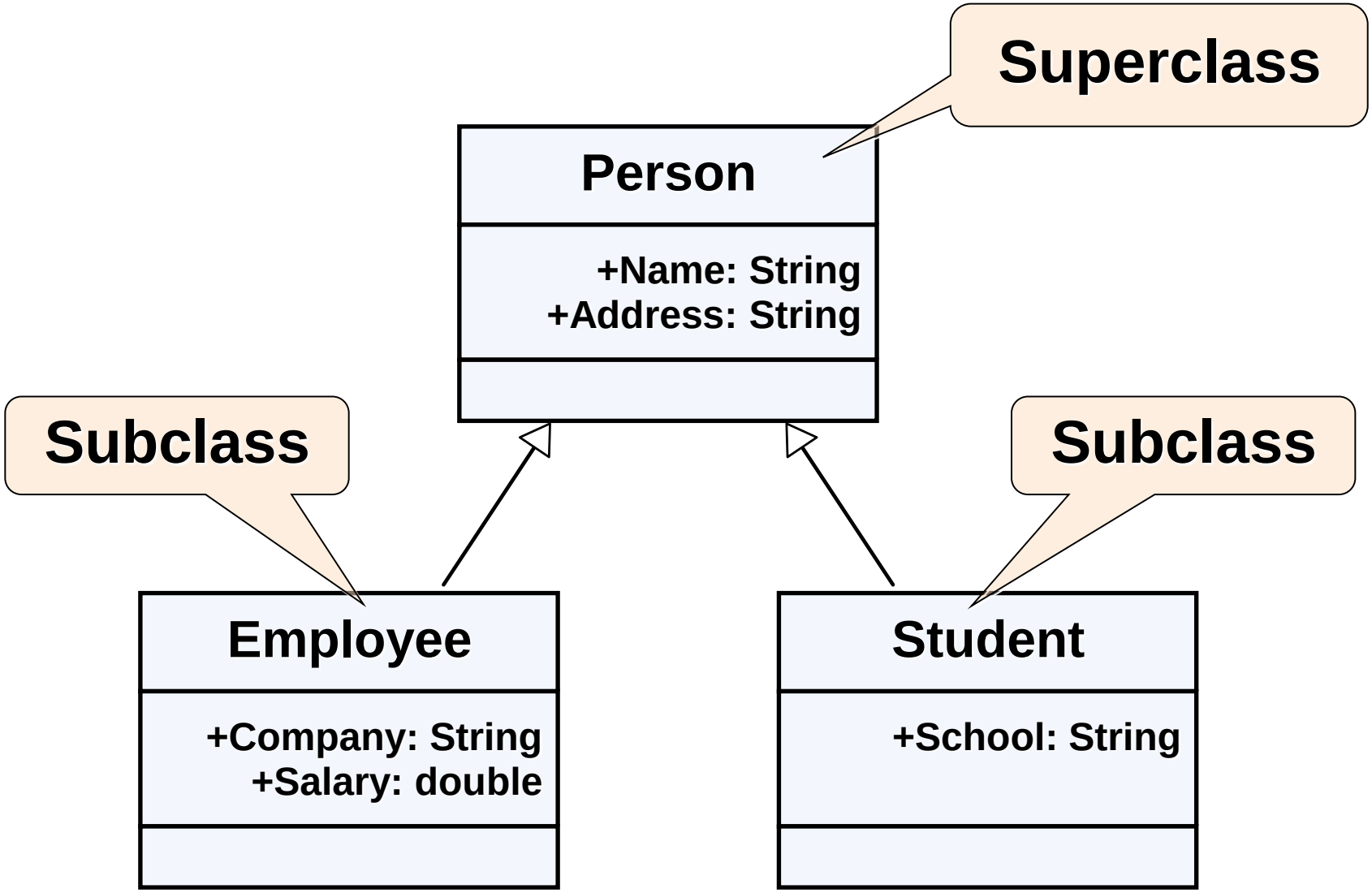
- class الذي يقوم بتوريث صفاته للآخر نقوم بتسميته **superclass** بينما class الذي يرث صفات و سلوك من الآخرين نقوم بتسميته **subclass** .

- subclass يمتلك صفات و سلوك الـ superclass و يزيد عليها صفات و سلوك آخرى.

- في لغة جافا يتم الوراثة من class واحدة فقط وهو مايعرف ب **Single Inheritance** .

مثال:




Superclass
Person
+Name: String
+Address: String
Subclass
Subclass
Employee
+Company: String
+Salary: double
Student
+School: String

الشكل العام لاستخدام الوراثة

لكي تجعل class جديدا يرث class موجودًا بالفعل كل ما عليك أن تكتب وقت التعريف بعد اسم class الجديد كلمة extends و بعدها اسم class الذي تريد وراثته

```
modifier(s) class ClassName extends ExistingClassName modifier(s)
{
    memberList
}
```

```
public class Employee extends Person {
 .....
 .....
 .....
}
```

مثال:

```
public class Manager extends Employee {
 .....
 .....
 .....
}
```

Super class

Sub class

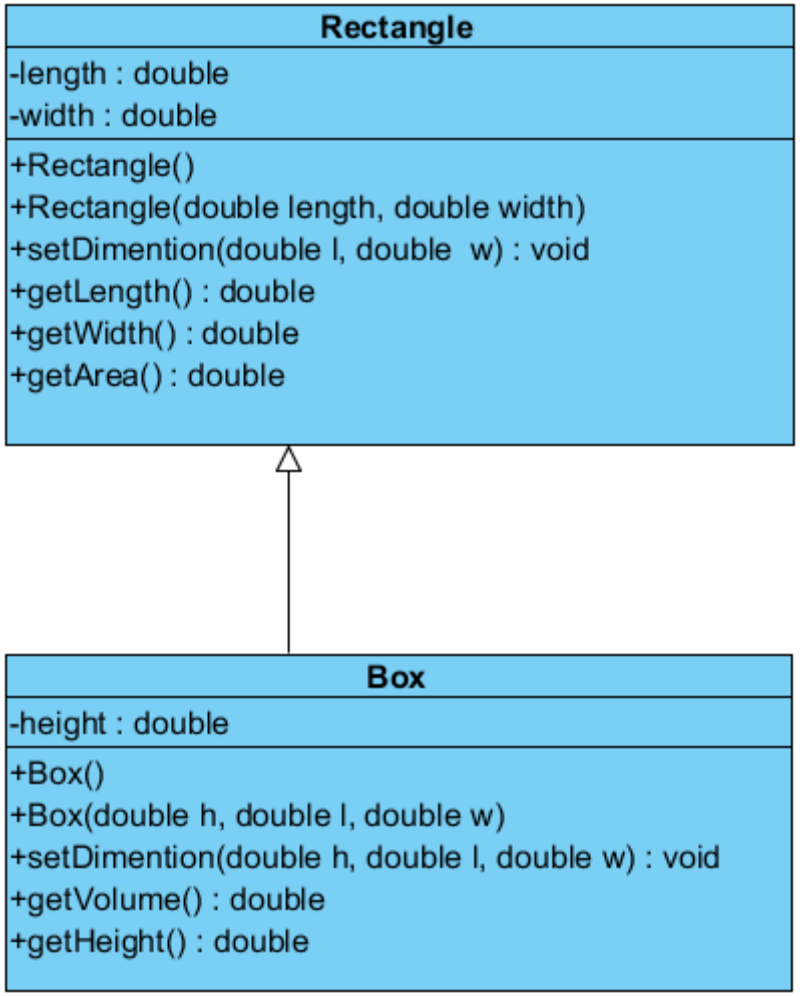
```
public class Employee {
      String      name;
      Address     address;
      String      phoneNumber;
      int         employeeNumber;
      float       hourlyPay;
      …
}

public class Manager extends Employee {
      String[]    duties;
      Employee[]  subordinates;
      …
}
```

عندما يرثclass جديدا class آخرا فإنه يرث كل ما هو public, protected باستثناء Constructor إلا أن subclass يقوم ضمنيًا باستدعاء الـ Constructor الخاص بالصنف superclass.

كما يمكن أن يستدعيه بشكل صريح من داخل الـ Constructor الخاص به بجملة برمجية واحدة كالتالي:

```
super();
```

و هي تعني استدعاء لـ Constructor الخاص بالأب و هي خطوة تتم بشكل ضمني و بالتالي لا داعي لكتابتها بشكل صريح


Rectangle
-length : double
-width : double
+Rectangle()
+Rectangle(double length, double width)
+setDimention(double l, double w) : void
+getLength() : double
+getWidth() : double
+getArea() : double
Box
-height : double
+Box()
+Box(double h, double l, double w)
+setDimention(double h, double l, double w) : void
+getVolume() : double
+getHeight() : double

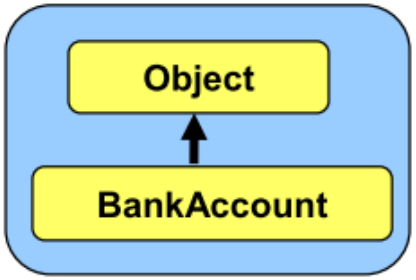
- في هذا المثال نقوم بأشتقاق Box class بأستخدام Rectangle class وذلك بأستخدام مبدأ الوراثة.
- لاستداعاء constructor الخاص ب Rectangle class نستخدم **super()**
- لاستدعاء الدالة getLength نستدعيها بأستخدام أسمها مباشرة **getLength()**
- لاستدعاء الدالة setDimention الموجودة في Rectangle class نستخدم **super.setDimention()** وذلك لوجود دالة في Box class تحمل نفس الاسم.

Protected data members

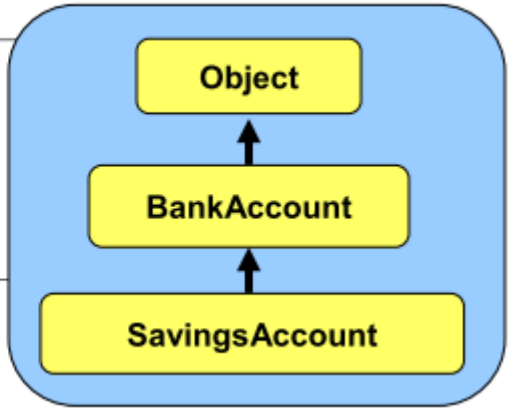
- عند أستخدام الوراثة data members من النوع private لايمكن الوصول إليهم مباشرة
- أذا أستخدم محدد الوصول public فيكون الوصول إليهم من الجميع (من أي class أخرى)
- لجعل أمكانية الوصول data members من superclass و subclass فقط يتم أستخدام محدد الوصول protected .

مثال:

```
public class BankAccount {
      String      owner;           // person who owns the account
      int         accountNumber;   // the account number
      float       balance;         // amount of money currently in the account

      // Some constructors
      public BankAccount() {
         this.owner = "";
         this.accountNumber = 0;
         this.balance = 0;
      }
      public BankAccount(String ownerName) {
         this.owner = ownerName;
         this.accountNumber = 0;
         this.balance = 0;
      }

      // Deposit money into the account
      public void deposit(float amount) {
         this.balance += amount;
      }

      // Withdraw money from the account
      public void withdraw(float amount) {
         if (this.balance >= amount)
            this.balance -= amount;
      }
}
```



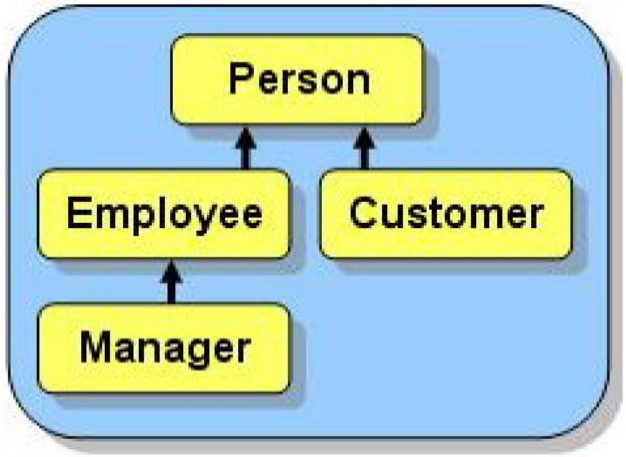
```
public class SavingsAccount extends BankAccount {
    public SavingsAccount() {
        super();
    }
}
```




```
SavingsAccount   s = new SavingsAccount();

System.out.println(s.balance);          // displays 0.0

s.deposit(120);
System.out.println(s.balance);          // displays 120.0

s.withdraw(20);
System.out.println(s.balance);          // displays 100.0
```

مثال:

```
public class Person {
      String      name;
      Address     address;
      String      phoneNumber;
}

public class Employee extends Person {
      int         employeeNumber;
      float       hourlyPay;
}

public class Customer extends Person {
      String[]    itemsPurchased;
      Date[]      purchaseHistory;
}

public class Manager extends Employee {
      String[]    duties;
      Employee[]  subordinates;
}
```



```
Person     p = new Person();
Employee   e = new Employee();
Customer   c = new Customer();
Manager    m = new Manager();

p.name = "Hank Urchiff";                // own attribute
p.address = new Address();              // own attribute
p.phoneNumber = "1-613-555-2328";       // own attribute

e.name = "Minnie Mumwage";              // attribute inherited from Person
e.address = new Address();              // attribute inherited from Person
e.phoneNumber = "1-613-555-1231";       // attribute inherited from Person
e.employeeNumber = 232867;              // own attribute
e.hourlyPay = 8.75f;                    // own attribute

c.name = "Jim Clothes";                 // attribute inherited from Person
c.address = new Address();              // attribute inherited from Person
c.phoneNumber = "1-613-555-5675";       // attribute inherited from Person
c.itemsPurchased[0] = "Pencil Case";    // own attribute
c.purchaseHistory[0] = Date.today();    // own attribute

m.name = "Max E. Mumwage";              // attribute inherited from Person
m.address = new Address();              // attribute inherited from Person
m.phoneNumber = "1-613-555-8732";       // attribute inherited from Person
m.employeeNumber = 232867;              // attribute inherited from Employee
m.hourlyPay = 8.75f;                    // attribute inherited from Employee
m.duties[0] = "Phone Clients";          // own attribute
m.subordinates[0] = e;                  // own attribute
```

تعدد الاشكال

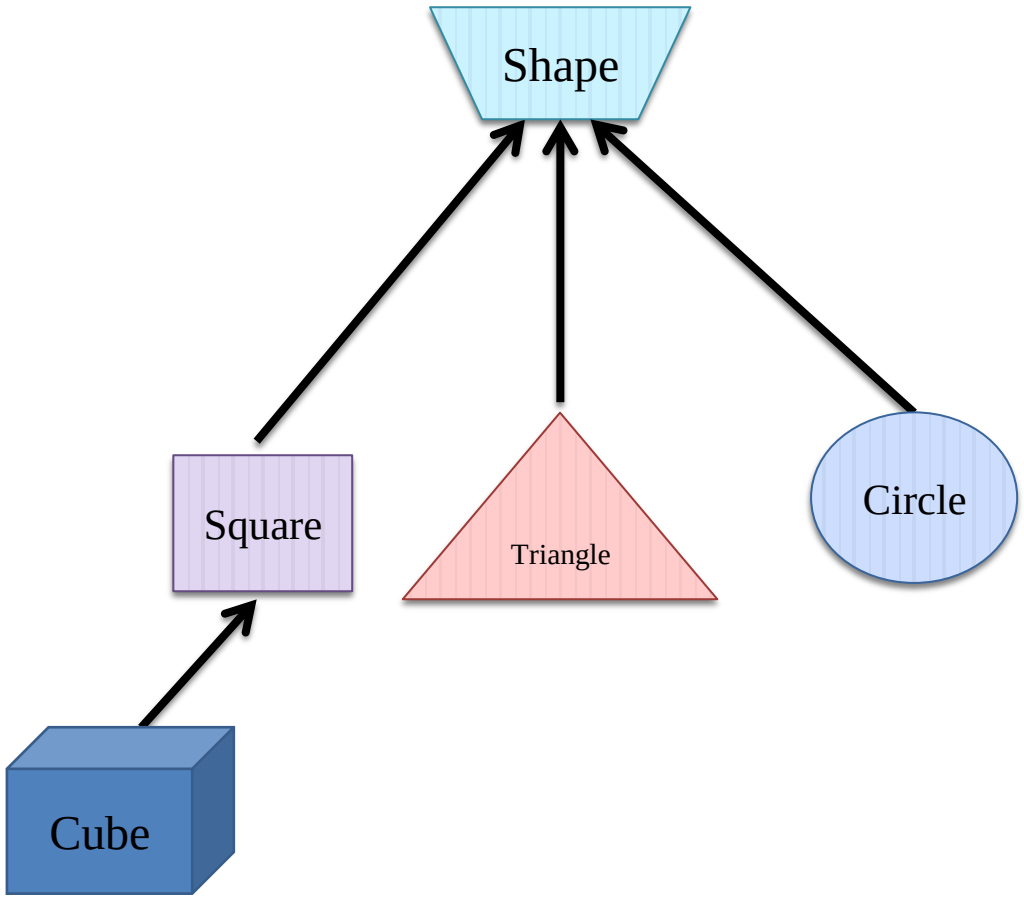
Polymorphism

تعدد الأشكال polymorphisms

- تسمح خاصية تعدد الأشكال لل Object أن تكون له أشكال مختلفة تقدم نفس الوظيفة ولكن تنفيذها يتم بطرق مختلفة.
- Polymorphism هي القدرة على تنفيذ دالة في superclass بعدة أشكال وذلك بحسب subclass الوارث لهذه الدالة.
- Polymorphismيسمح بعمل تغيرات على الدوال التي تم وراثتها.
- في object Polymorphism من super class يمكن أن يؤشر على object من sub class.

مثال:

حسب الرسم التالي أي Shape Object ممكن أن يأخد صورة Square Object أو Cube Object وكذلك أي Square Object يستطيع أن يأخد Cube Object



مثال:

```
Cube c = new Cube();

Square s = c;

Shape e = c;

Shape  e = new Square();
```

مثال:

الاستاذ في المدرسة ممكن أن يكون بأكثر من صورة أستاذ رياضة أو أستاذ لغة أنجليزية وفي كلا الحالتين له نفس الوظيفة وهي التدريس ولكنه يقوم بها بطريقة مختلفة في كلاهما.

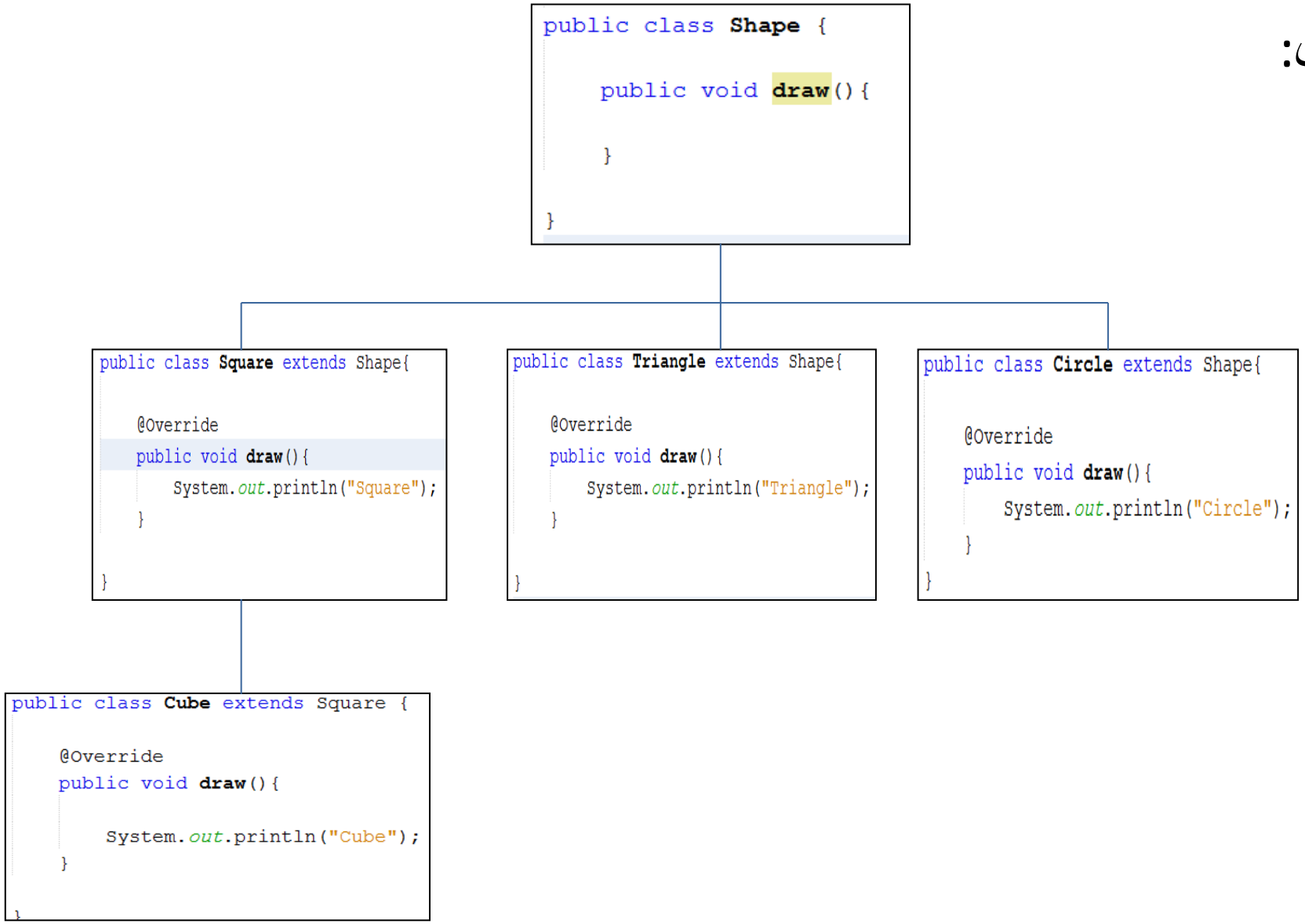
مثال:

مجموعة الحيونات في الصورة جميعها لها نفس الوظيفة هنا "Speak" ولكن كل منها تقوم بها بطريقة مختلفة حسب نوعها (كلب، قطة، بطة) مع أنها جميعها تحت التصنيف **حيوان**.

مفاهيم متعلقة Polymorphism

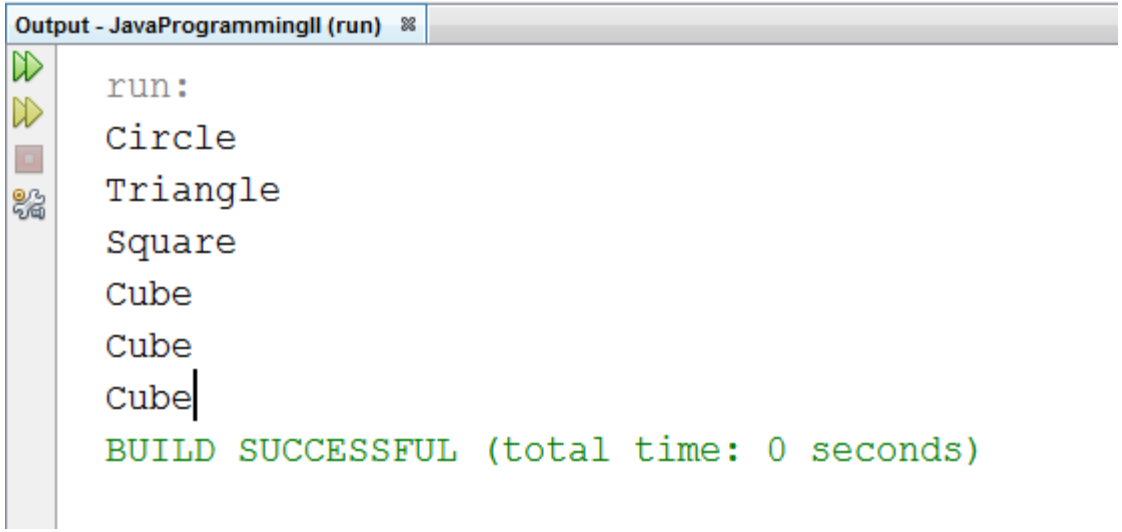
- **Method Override** وهي عملية تنفيذ الدالة الموجودة في subclass بدلا من تلك الموجودة في superclass.

مثال:


public class Shape {
public void draw(){
}
}
public class Square extends Shape{
@Override
public void draw(){
System.out.println("Square");
}
}
public class Triangle extends Shape{
@Override
public void draw(){
System.out.println("Triangle");
}
}
public class Circle extends Shape{
@Override
public void draw(){
System.out.println("Circle");
}
}
public class Cube extends Square {
@Override
public void draw(){
System.out.println("Cube");
}
}

مانتيجة تنفيذ البرنامج التالي

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

        Shape e;
        Square s;

        e = new Circle();
        e.draw();

        e= new Triangle();
        e.draw();

        e = new Square();
        e.draw();

        e = new Cube();
        e.draw();

        Cube c = new Cube();

        s=c;
        s.draw();

        e = c;
        e.draw();

    }
```


Output - JavaProgrammingII (run)
run:
Circle
Triangle
Square
Cube
Cube
Cube
BUILD SUCCESSFUL (total time: 0 seconds)

Abstract class & Interface

Abstract Method

- Abstract method هي الدالة التي يتم الاعلان عنها بدون تحديد بنيتها الداخلية.
- تستخدم الكلمة abstract للاعلان عن دالة من النوع Abstract .

مثال :

```
public abstract double calculateArea( );
```

Abstract Class

- إي Class تحتوي على abstract method تعرف ب Abstract Class .
- للاعلان على Abstract Class تستخدم الكلمة Abstract عند أنشاء class .

```
abstract class MyClass {...}
```

مثال:

```
public abstract class Shape {
    private String color;


    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public  abstract  double calculateArea();

}
```

```java
public class Circle extends Shape {
    private double radius;
    private final double PI =3.14;

    public Circle(double radius, String color) {
        this.radius = radius;
        super.setColor(color);
    }


    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }


    @Override
    public double calculateArea() {
       return this.getRadius()*this.getRadius()*PI;
    }

}
```

خصائص Abstract Class

- لايمكن أنشاء Object من Abstract Class.
- لمنع الحصول على Object من أي class يتم تعريفها على أنها Abstract class وذلك بأستخدام الكلمة Abstract عند أنشائها.
- Abstract Classتستخدم في أنشاء تصنيف جديد يحتوي على عدة classes.
- يمكن أستخدام Abstract Class للحصول على subclass .
- أذا لم يتم كتابة الاوامر المكونة لل abstract methods الموروثة من superclass تعتبر subclass أيضاً abstract class.

مثال:

- يمكن أنشاء abstract class بأسم Shape ويتم أستخدامه في أنشاء مجموعة من subclasses (Square, Rectangle, Circle, Triangle) فيصبح لدينا الصنف Shape يتضمن هذه المجموعة من Classes .

Interface

- Interface يستخدم في الاعلان عن مجموعة من الدوال بدون كتابة بنيتها الداخلية.

- مثال:

```
public interface Shape {
public String getName();
public double getArea();
}
```

```
public interface Relation {
public boolean isGreater( Object a, Object b);
public boolean isLess( Object a, Object b);
public boolean isEqual( Object a, Object b);
}
```

خصائص Interface

- كل الدوال في Interface تكون Abstract و Public.
- لايمكن أنشاء Object من Interface .
- Interface ممكن أن يحتوي على ثوابت (Final Variables).
- أي class تريد أن تستخدم Interface تقوم بأستخدام الكلمة Implements .

```
class Square  implements  Shape {
...
}
```

```java
public class Square implements Shape {

    private String name;
    private double length;

    public Square(String name, double length) {
        this.name = name;
        this.length = length;
    }

    public double getLength() {
        return length;
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public double getArea() {
        return getLength() * getLength();
    }

}
```

- يمكن لـ Class أن تقوم بعمل implement لأكثر من Interface .

```
public class Circle implements Shape, Relation{
    private double radius;
    private String name;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea(){
        return this.radius*this.radius*3.14;
    }

    @Override
    public String getName() {
        return this.name;
    }

    @Override
    public boolean isGreater(Object a) {
       return  getArea() > ((Circle)a).getArea();
    }

    @Override
    public boolean isLess(Object a) {
         return  getArea() < ((Circle)a).getArea();
    }

    @Override
    public boolean isEqual(Object a) {
          return getArea() == ((Circle)a).getArea();
    }
}
```

- يمكن أن نقوم بعمل extends للInterface .

```
public interface ThreeDimentionShape extends Shape {

    public double  getVolume();

}
```

- يمكن أستخدام interfaceلعدة أنواع من classes ليس لها علاقة ببعض

```
public class Account implements Relation {

    private double balance;

    public Account(double balance) {
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }

    @Override
    public boolean isGreater(Object a) {
        return getBalance() > ((Account)a).getBalance();
    }

    @Override
    public boolean isLess(Object a) {
       return getBalance() < ((Account)a).getBalance();
    }

    @Override
    public boolean isEqual(Object a) {
        return getBalance() == ((Account)a).getBalance();
    }

}
```

```
public class Circle implements Relation{
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getArea(){
        return this.radius*this.radius*3.14;
    }



    @Override
    public boolean isGreater(Object a) {
       return  getArea() > ((Circle)a).getArea();
    }

    @Override
    public boolean isLess(Object a) {
        return  getArea() < ((Circle)a).getArea();
    }

    @Override
    public boolean isEqual(Object a) {
         return getArea() == ((Circle)a).getArea();
    }
}
```

Interface as a Type

- عندما نقوم بتعريف interface هذا يعني أننا قمنا بتعريف reference type جديد.
- يمكن أستخدامه في أي مكان يتم فيه أستخدام reference types الاخرى.
- عند أستخدامه لتعريف متغير يجب ان يكون Object الذي سيتم تخصيصه لهذا المتغير من class تقوم بعمل implement لهذا interface .

```
public class interfaceTypeExample {
    public static void main(String[] args) {
        Square s = new Square("square1", 10);
         display(s);
        Triangle t = new Triangle("triangle1", 10, 5);
        display(t);

    }

    public static  void display(Shape s){
        System.out.println("Shape Name: "+s.getName());
        System.out.println("Shape Area: "+s.getArea());
    }
}
```

مقارنة بين abstract class و interface

- كل methods في interface تكون public و abstract بينما في abstract class ليس من الضروري أن تكون كلها abstract.
- في Interface يمكن أن نعرف فقط constants بينما في abstract class يمكن أن نعرف attributes و constants .
- كل abstract classes تشترك في العلاقة مع Object class بينما interfaces ليس لديها هذه العلاقة مع Object class .

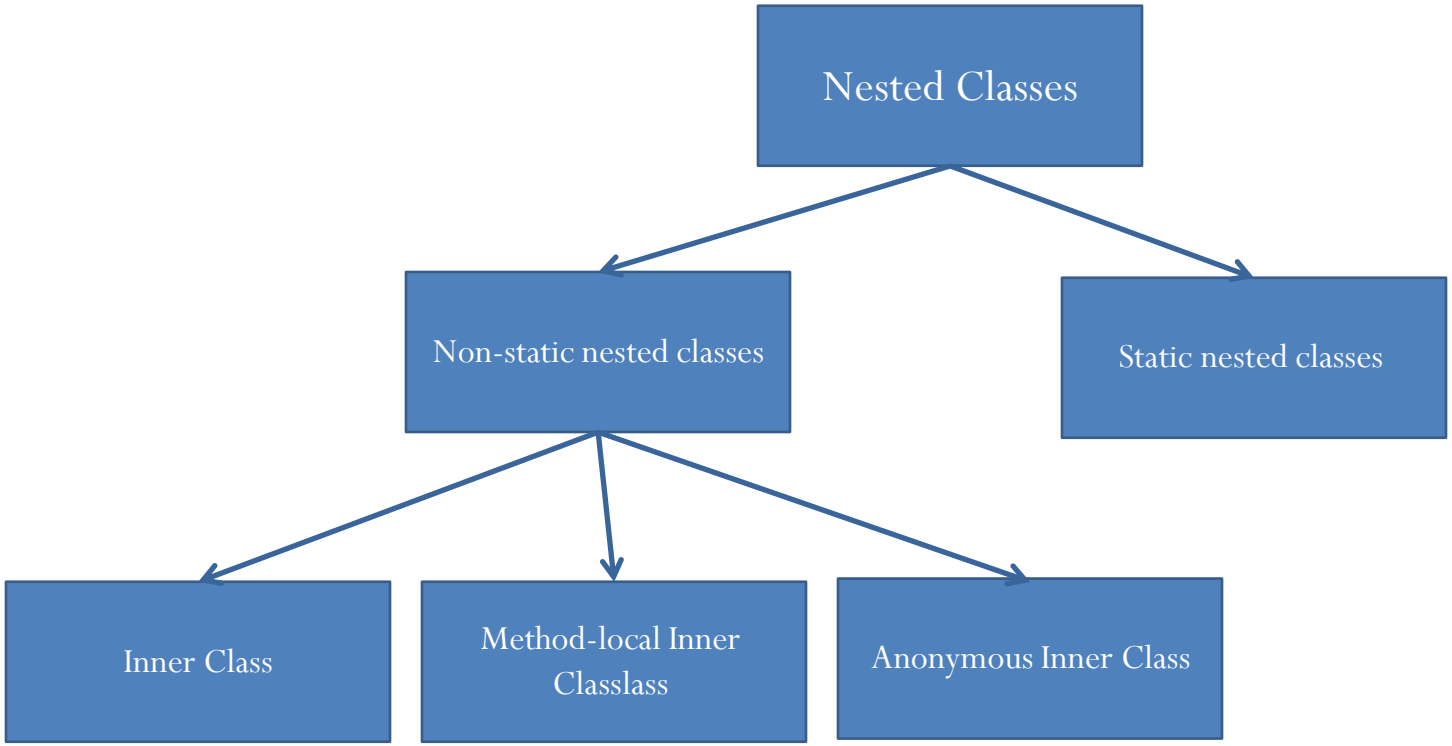
Nested Class

Nested Classes

- تسمح لغة جاف بأنشاء class داخل class أخرى وهو مايعرف ب Nested Classes:
  - Class الموجودة داخل class أخرى تعرف ب **nested class** .
  - Class الموجودة بها nested class تعرف ب **outer class** .
- يوجد نوعين من nested class:
  - **Static nested classes**
  - **Non-static nested classes**

وهي تنقسم إلى ثلاثة أنوع:

- Inner Class
- Method-local Inner Classlass
- Anonymous Inner Class


Nested Classes
Non-static nested classes
Static nested classes
Inner Class
Method-local Inner Classlass
Anonymous Inner Class

Inner Class

- ترتبط inner class مع outer class object ويمكنها الوصول لكل attributes و methods الموجودة بها.
- للحصول على object من inner class يجب أولا الحصول على object من outer class .

مثال:

```
public class OuterClass {

    private class InnerClass{
        public void print(){
            System.out.println("This is the the inner class");
        }
    }

    public void printInner(){
        InnerClass inner = new InnerClass();
        inner.print();
    }

}
```

```java
public class InnerClassExample1 {
    public static void main(String[] args) {
        OuterClass outter = new OuterClass();
        outter.printInner();
    }

}
```

```
run:
This is the the inner class
BUILD SUCCESSFUL (total time: 0 seconds)
```

مثال:

```
public class OuterClass2 {

    private int num = 175;

    public class InnerClass {
        public int getNum() {
            return num;
        }
    }
}
```

```
public class InnerClassExample2 {

    public static void main(String[] args) {
        OuterClass2 outer = new OuterClass2();
        OuterClass2.InnerClass inner = outer.new InnerClass();
        System.out.println(inner.getNum());
    }

}
```

Method-local Inner Classlass

- يمكن في لغة جافا أنشاء class بداخل method وفي هذه الحالة يكون عملها محدود بهذه method فقط.
- للحصول على object منها يتم ذلك بداخل method المعرفة بها.

مثال:

```
public class OuterClass3 {
    public void myMethod(){
        int num = 175;
         class MethodInnerClass{
            public void print(){
                System.out.println(num);
            }

        }
        MethodInnerClass inner = new MethodInnerClass();
        inner.print();
    }
}
```

```
public class InnerClassExample3 {
    public static void main(String[] args) {
        OuterClass3 outer = new OuterClass3();
        outer.myMethod();
    }
}
```

Anonymous Inner Class

- هي inner class التي يتم أنشائها بدون أعطائها أسم.
- تستخدم عادة عندما نرغب في عمل override ل method موجودة ب interface أو class .

In case of anonymous inner classes, we declare and instantiate them at the same time.

مثال:

```
public abstract class MyAbstractClass {
    public abstract void myMethod();
}
```

```
public class InnerClassExample4 {
    public static void main(String[] args) {
        MyAbstractClass inner = new MyAbstractClass() {
            @Override
            public void myMethod() {
                System.out.println("This is an example of an  anonymous inner class");
            }
        };

        inner.myMethod();
    }
}
```

Static Nested Class

- هي static inner class وتكون أحدى static memebers الخاصة ب outer class .
- يمكن الوصول إليها دون الحاجة للحصول على object من outer class .
- لايمكنها الوصول إلى object variables و object methods الخاصة ب outer class .

مثال:

```
public class OuterClass4 {

    static class Nested_Demo {
        public void myMethod() {
            System.out.println("This is my static nested class");
        }
    }

    public static void main(String args[]) {
        OuterClass4.Nested_Demo nested = new OuterClass4.Nested_Demo();
        nested.myMethod();
    }
}
```

شكراً