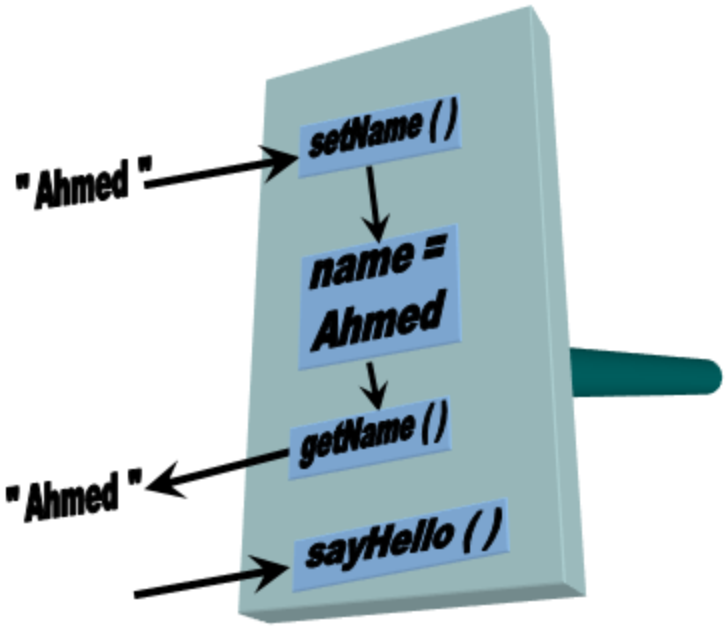
Unit 3
Object Oriented Programming Concepts

Dr. Magdi AMER

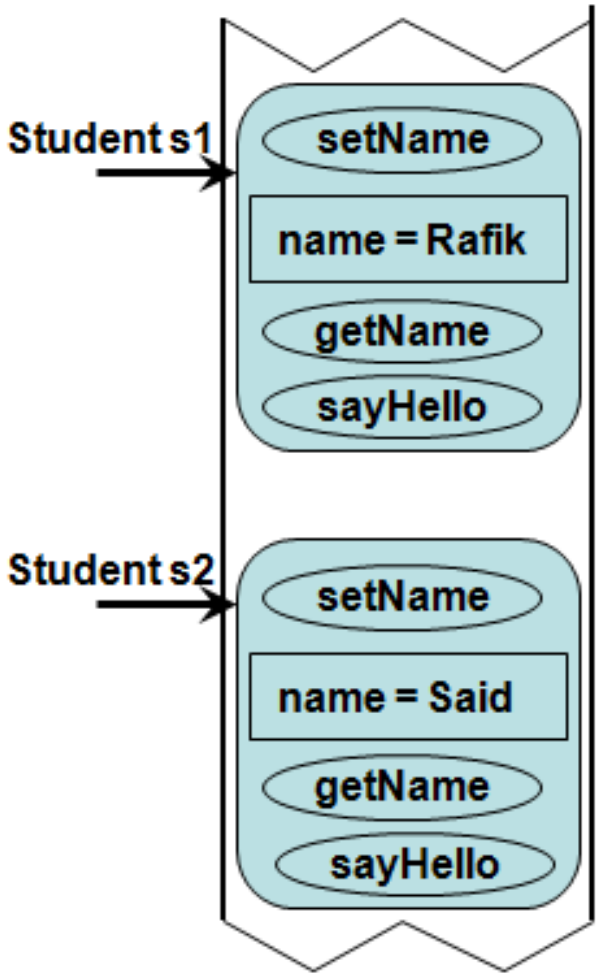
Using Objects

```
public class Student  {
  protected String name;
public Student()
{name = "“;}
  public void setName(String aName)
 {  name = aName;   }
  public String getName( )
 {   return name;   }
public void sayHello()
{ System.out.println("Hello");}
}
```



Using Objects

```
public class MyApp
{
        public static void main(String args[])
        {
        Student s1;
        s1 = new Student();
        s1.setName("Rafik");
        Student s2 = new Student();
        s2.setName("Said");
        }
}
```

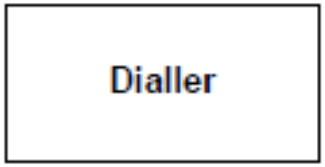


Example

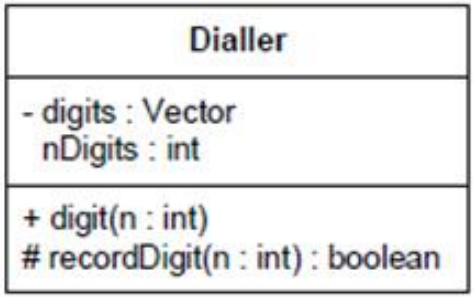
You are required to write a Java application for a bookstore. The application is composed of 3 classes: a category, an author, and a book. The book has a title (String), an id (int), a category and an author. The Category has an id (int), a name (String) and a description (String). The author has a first-name (String), a family-name (String) and a National ID number (int).
Write a method (printInfo) that prints the information about a book.
Create an author (first-name = Tawfeek, last-name= Elhakim, NID =728632) another author (first-name = Nagueeb, last-name= Mahfooz, NID =763212). Create a category (id=1, name= Novel, description= Arabic novel). Create a book (title= sookareya, id=10), its author is Nagueeb Mahfooz and its category is a Novel. Create another book (title= Asfoor Men Elshark, id=11), its author is Tawfeek Elhakeem and its category is a Novel. Call printInfo on each of the books that you have created.

Class Diagram

- **Class**



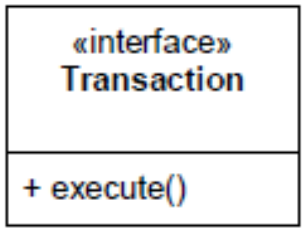

```
public class Dialler
{
}
```



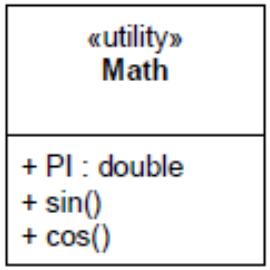

```
public class Dialler
{
  private Vector digits;
  int nDigits;
  public void digit(int n);
  protected boolean recordDigit(int n);
}
```

Class Diagram

- **Class Interface**




```
interface Transaction
{
  public void execute();
}
```

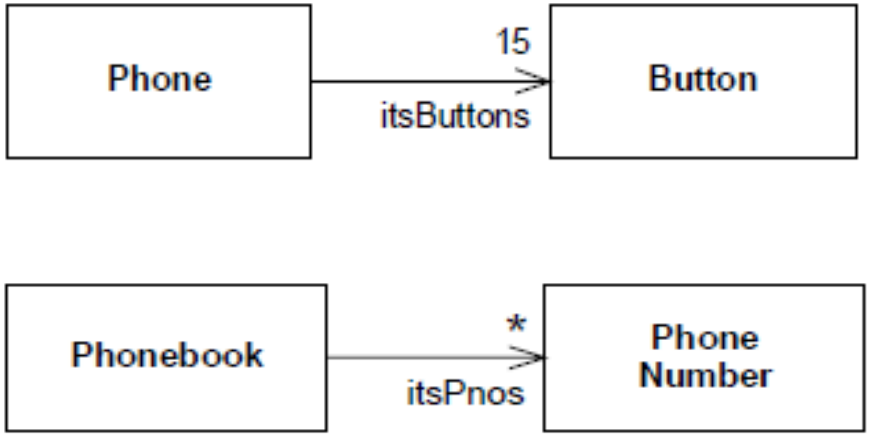



```
public class Math
{
  public static final double PI =
                              3.1415926535897932;

  public static double sin(double theta){...}
  public static double cos(double theta){...}
}
```

All the methods and variables of a «utility» class are static.

Class Diagram

- **Association**




```
public class Phone
{
  private Button itsButtons[15];
}
```

```
public class Phonebook
{
  private Vector itsPnos;
}
```

ArrayList example

```
class Employee {
    protected String id;
    protected String name;
    public Employee(String id, String name) {
        this.id = id;
        this.name = name;
    }
//set and get methods
}
```

ArrayList example

```
public static void main(String args[]) {
        Employee e1 = new Employee("E100","Ahmed");
        Employee e2 = new Employee("E217","Mina");
        Employee e3 = new Employee("E412","Mona");
        ArrayList<Employee> data = new ArrayList<Employee>();
        data.add(e1);
        data.add(e2);
        data.add(e3);
        for(Employee e: data) {
         System.out.println("Employee "+e.getName()+" has an Id "+e.getId());
      }
   }
}
```

Relationship between Objects

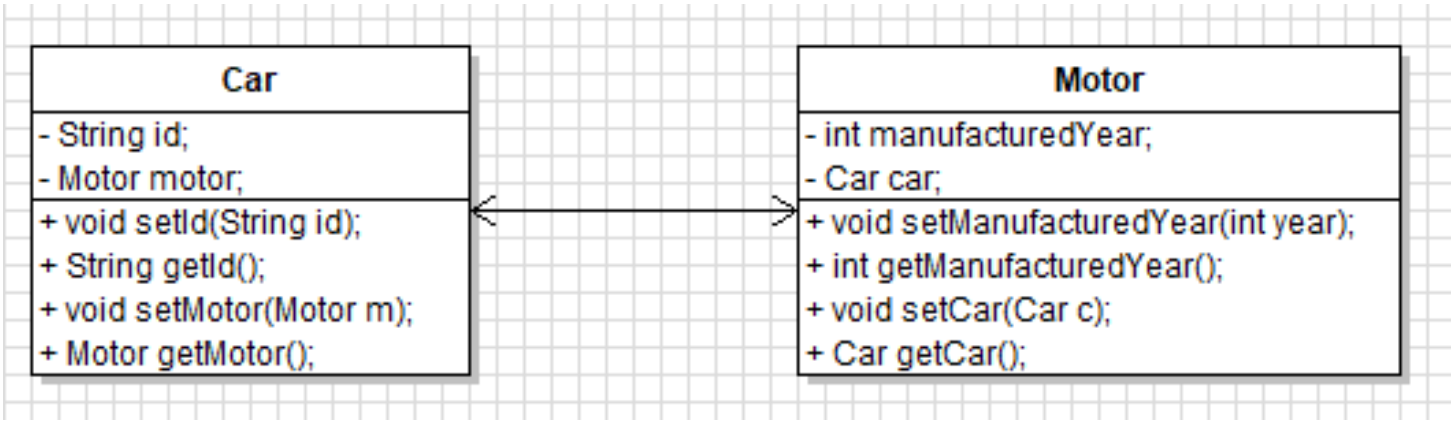
Relationships between objects can be:

- **1-to-1**
  - **Example: A car and a motor.**
    A car contains 1 motor only and a motor is placed inside 1 car only

- **1-to-many**
  - **Example: Car and tires.**
    A car has 4 tires (at least) and each tire is placed in 1 car only

- **Many-to-many**
  - **Example: Student and courses**
    A student has taken many courses, and a course is attended by many students.

Some Design Patterns
One-to-one



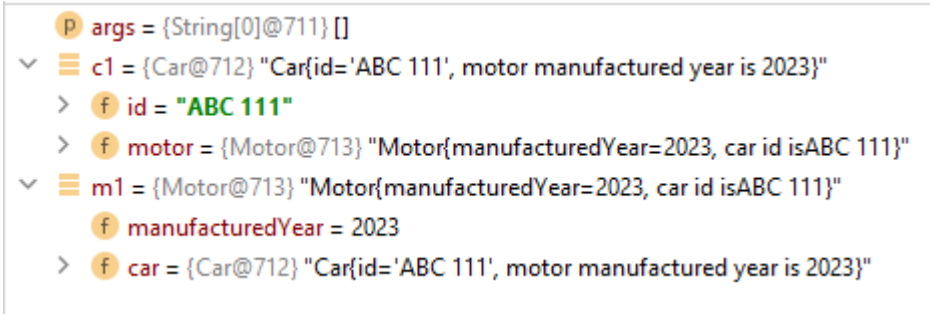
Some Design Patterns
One-to-one

```java
public class Car {
    3 usages
    private String id;
    3 usages
    private Motor motor;

    @Override
    public String toString() {
        return "Car{" +
                "id='" + id + '\'' +
                ", motor manufactured year is " + motor.getManufacturedYear() +
                '}';
    }
    1 usage
    public String getId() {
        return id;
    }

    1 usage
    public void setId(String id) {
        this.id = id;
    }

    no usages
    public Motor getMotor() {
        return motor;
    }

    1 usage
    public void setMotor(Motor motor) {
        this.motor = motor;
    }
}
```

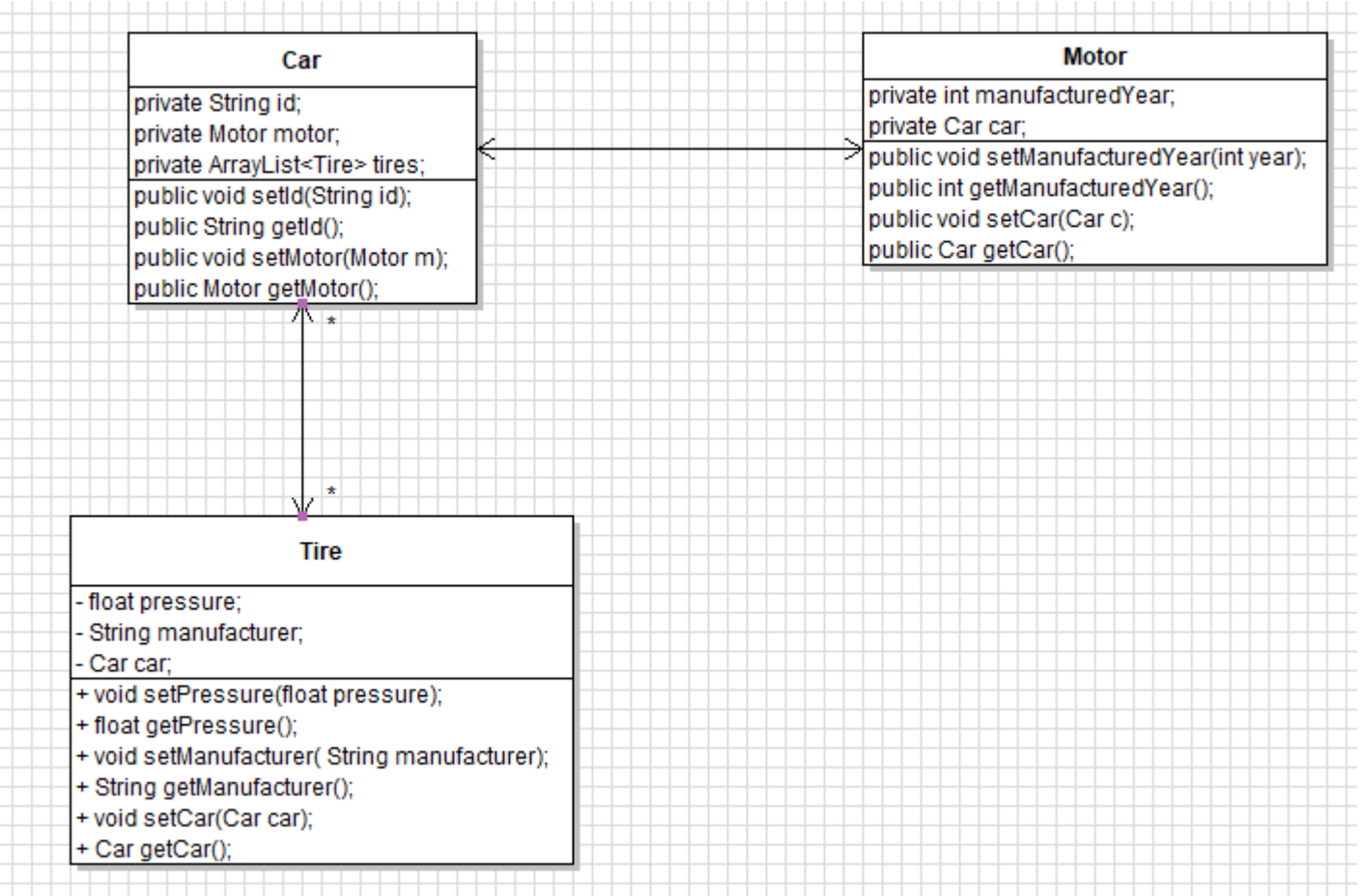
```java
public class Motor {
    3 usages
    private int manufacturedYear;
    3 usages
    private Car car;
    @Override
    public String toString() {
        return "Motor{" +
                "manufacturedYear=" + manufacturedYear +
                ", car id is" + car.getId() +
                '}';
    }
    1 usage
    public int getManufacturedYear() {
        return manufacturedYear;
    }
    1 usage
    public void setManufacturedYear(int manufacturedYear) {
        this.manufacturedYear = manufacturedYear;
    }
    no usages
    public Car getCar() {
        return car;
    }
    1 usage
    public void setCar(Car car) {
        this.car = car;
    }
}
```

Some Design Patterns
One-to-one

```
public class Main {
    public static void main(String[] args) {   args: []
        Car c1 = new Car();   c1: "Car{id='ABC 111', motor manufactured year is 2023}"
        c1.setId("ABC 111");

        Motor m1 = new Motor();   m1: "Motor{manufacturedYear=2023, car id isABC 111}"
        m1.setManufacturedYear(2023);

        m1.setCar(c1);
        c1.setMotor(m1);   c1: "Car{id='ABC 111', motor manufactured year is 2023}"

        //System.out.println("END");

        System.out.println(m1);   m1: "Motor{manufacturedYear=2023, car id isABC 111}"
    }

}
```



Some Design Patterns
1-to-many



Some Design Patterns
1-to-many

- **When using a ArrayList as an instance variable, there are some rules:**
  - **We must create the ArrayList inside the constructor of the class**
  - **We must have a method add*ClassName* (*ClassName* is replaced by the name of the class) that adds an object to the list of objects**

```
public class Car {
    3 usages
    private String id;
    3 usages
    private Motor motor;

    5 usages
    private ArrayList<Tire> tires;

    1 usage
    public Car() {
        this.tires = new ArrayList<>();
    }

    4 usages
    public void addTire(Tire tire){
        this.tires.add(tire);
    }
```

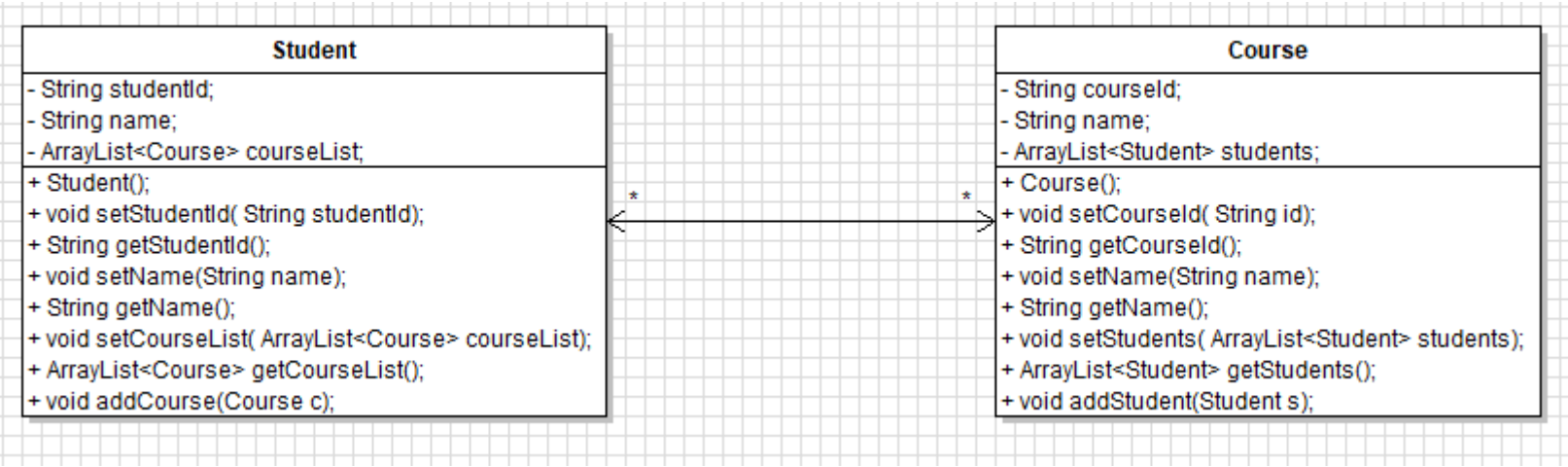
Some Design Patterns
1-to-many

```
Tire t1 = new Tire();
t1.setPressure(32);
t1.setManufacturer("Michelin");

Tire t2 = new Tire();
t2.setPressure(32);
t2.setManufacturer("Michelin");

Tire t3 = new Tire();
t3.setPressure(32);
t3.setManufacturer("BlackBridge");

Tire t4 = new Tire();
t4.setPressure(32);
t4.setManufacturer("BlackBridge");

t1.setCar(c1);
c1.addTire(t1);

t2.setCar(c1);
c1.addTire(t2);

t3.setCar(c1);
c1.addTire(t3);

t4.setCar(c1);
c1.addTire(t4);

System.out.println(c1);
System.out.println(t1);
```

```
Car{id='ABC 111', motor manufactured year is 2023, has 0 tires }
Car{id='ABC 111', motor manufactured year is 2023, has 4 tires
 tire number 0 has manufacturer Michelin
 tire number 1 has manufacturer Michelin
 tire number 2 has manufacturer BlackBridge
 tire number 3 has manufacturer BlackBridge}
```

Many-to-Many

- **A Student can take multiple courses. A course is attended by multiple students. This is an example of Many-to-Many.**



Many-to-Many

```java
public class Course {
    2 usages
    protected String courseId;
    protected String name;
    4 usages
    protected ArrayList<Student> students;

    2 usages
    protected ArrayList<Professor> profList;

    2 usages
    public Course() {
        this.students = new ArrayList<>();
        this.profList = new ArrayList<>();
    }

    3 usages
    public void addStudent(Student s){
        this.students.add(s);
    }

    no usages
    public void addProfessor(Professor p){
        this.profList.add(p);
    }
```

```java
public class Student {
    3 usages
    protected String studentId;
    protected String name;
    7 usages
    protected ArrayList<Course> courseList;

    2 usages
    public Student() {
        this.courseList = new ArrayList<>();
    }

    3 usages
    public void addCourse(Course c){
        this.courseList.add(c);
    }
@Override
public String toString() {
    String result =  "Student{" +
            "studentId='" + studentId + '\'' +
            "name='" + name + '\'' +
            ", has " + courseList.size() +" courses ";
    for(int i=0; i< courseList.size(); i++){
        result = result + "\n course number "+i+
                " has manufacturer "+courseList.get(i).getName();
    }

    result = result + '}';
    return result;
}
```

Many-to-Many

```java
public class MyMain {
    public static void main(String[] args) {
        Course c1 = new Course();
        c1.setName("OOP");

        Course c2 = new Course();
        c2.setName("Database");

        Student s1 = new Student();
        s1.setStudentId("12");
        s1.setName("Ahmed");

        Student s2 = new Student();
        s2.setStudentId("15");
        s2.setName("Morcos");

        c1.addStudent(s1);
        s1.addCourse(c1);

        c1.addStudent(s2);
        s2.addCourse(c1);

        c2.addStudent(s1);
        s1.addCourse(c2);
```

Instance Method vs static method

- A class is a factory, the object is the product of the factory.
- Each object has inside it the variables that are defined by the class as well as the methods.
- If we want to place a counter that count the number of product the factory builds, where is the correct place to put it? Is it inside the products? Of course not. It should be placed inside the factory itself.
- Similarly, if we want to have a counter that counts the number of objects this class has built, the best place to put it is inside the class itself, hence the name class variable.
- All objects of this class will share the same instance of the class variable.

encapsulation

- The word “encapsulate” means to enclose or to place in a protective shell.
- In OOP, encapsulation is used to express that the data of each object must be placed inside this object. Furthermore, the data of each object should not be accessed directly. Instead, the data of an object must be accessible only through the methods that are defined in the class of this object.

```
public class Student  {
  protected String name;
  public void setName(String aName)
 {  name = aName;   }
  public String getName( )
 {   return name;   }
}
```

```
public class Student
{  protected char[] name = null;
  public void setName(String aName)
 {  name = aName.toCharArray(); }
  public String getName( )
 {  if(name = =null)
          return “”;
      else
          return new String(name); }  }
```

Instance variable vs static variable

```
public class Person {
   static int counter =0;
   public static  int getCounter() { return counter; }
   protected int id;
      public Person(int anId) {
      counter++;
      id = anId;
      System.out.println("Person with id "+ id+" is the object nb "+counter+
                         " that this class has constructed");    }}
public class InstanceAndClassVariable {
   public static void main(String[] args) {
      Person p1 = new Person(7);
      Person p2 = new Person(9);
      Person p3 = new Person(12);
      Person p4 = new Person(15);  }  }
```

```
Person with id 7 is the object nb 1 that this class has constructed
Person with id 9 is the object nb 2 that this class has constructed
Person with id 12 is the object nb 3 that this class has constructed
Person with id 15 is the object nb 4 that this class has constructed
```

Over loading

```
public class Person {
  public Person() { name=''; }
  public Person(String name) {this.name = name;}
  protected String name;
  public void setName(String aName) {  name = aName; }
  public void setName(char[]  nameArray)
{name = new String (nameArray); }
  public String getName( ) { return name; }
  public void sayHello( )
 {System.out.println("Person "+name+ "is saying Hello"); }  }
```

inheritance

```
public class Student {
  protected String name;
  protected int id;
  public void setName(String aName)
 {  name = aName; }
  public String getName( )
 { return name; }
  public void setId(int anId)
 {  id = anId; }
 public String getId( )
 { return id; }
}
```

```
public class Professor {
  protected String name;
  protected String jobTitle;
  public void setName(String aName)
 {  name = aName; }
  public String getName( )
 { return name; }
  public void setJobTitle(int aJob)
 {  jobTitle = aJob; }
 public String getJobTitle( )
 { return jobTitle; }
}
```

inheritance

```
public class Person {
  protected String name;
  public void setName(String aName)
 {  name = aName; }
  public String getName( )
 { return name; }   }
```

```
public class Student extends Person {
  protected int id;
  public void setId(int anId)
 {  id = anId; }
 public String getId( )
 { return id; }  }
```

```
public class Professor extends Person {
  protected String jobTitle;
  public void setJobTitle(int aJob)
 {  jobTitle = aJob; }
 public String getJobTitle( )
 { return jobTitle; }  }
```

inheritance

```
public class MyApp
{
  public static void main(String[] args)
  {
    Person p1;
    p1 = new Person();
    Student s1 = new Student();
    s1.setName("Said");
    s1.setId(5);
    Person p2 = new Student();
    p2.setName("ahmed");
    //p2.setId(4); //illegal
    }
}
```

Interface

```
public class Person {
 protected String firstName, lastName;
 protected int sinNumber;
 public Person(String f, Sting l, int num) {
 firstName = f;
 lastName = l;
 sinNumber = num; }
....
}
```

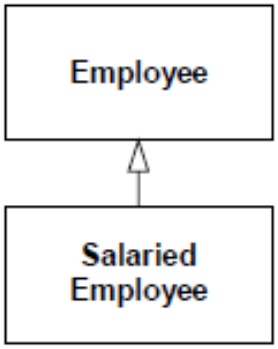
```
public class Teacher extends Person implements Employee
{
 public int calculateTax()
{
 .......
}
 ........
}
```

```
 interface Employee
 {
  int calculateTax();
 }
```

```
public class MyProgram {
 public static void main(String[] args)  {
  Person p = new Person("Winnie", "Pooh", 1313);
  Teacher t = new Teacher("Rabbit", "", 111);
  Employee e = t;
  //Employee e = p; //illegal   } }
```

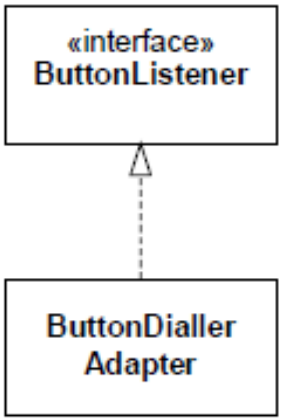
Class Diagram

- **Inheritance**




```
public class Employee
{
   ...
}

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

- **Implementing an interface**




```
interface ButtonListener
{
   ...
}

public class ButtonDiallerAdapter
   implements ButtonListener
{
   ...
}
```

Overriding

```
public class Person {
  protected String name;
  public void setName(String aName) {  name = aName; }
  public String getName( ) { return name; }
  public void sayHello( )
 {System.out.println("Person "+name+ "is saying Hello"); } }
```

```
public class Student extends Person {
  protected int id;
  public void setId(int anId) {  id = anId; }
  public String getId( ) { return id; }
 public void sayHello( )
 {System.out.println("Student "+name+ "is saying Hello"); } }
```

```
public class Professor extends Person {
  protected String jobTitle;
  public void setJobTitle(String aJob) {  jobTitle = aJob; }
  public String getJobTitle( ) { return jobTitle; } }
```

Overriding

```
public class MyApp  {
  public static void main(String[] args)   {
    Person p1 = new Person();
    p1.setName("Mina");
    p1.sayHello();
    Student s1 = new Student();
    s1.setName("Said");
    s1.setId(5);
    s1.sayHello();
    Person p2 = new Student();
    p2.setName("Ahmed");
    p2.sayHello();    //p2.getId();//Error
    Student st = (Student) p2;
    st.setId(5);
    Professor pf = new Professor( );
    pf.setName("Amer");
    pf.sayHello();  }  }
```

Access Modifier

Access Modifiers	*Same Class*	*Same Package*	*Subclass*	*Other packages*
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

Access Modifier

Java has four types of access modifiers:

public:

Public means that the method or attribute can be access from any object in the program.

private:

A private method or attribute cannot be access by any object in the system. Private members can be accessed only by the instance of the class. If a class *B* inherits from a class *A*, B will contain all the attributes and the method that were declared in *A*. If *A* contains a private member, than *B* cannot access it directly, but only through other methods that *B* inherited from *A*.

No access modifier:

When no access modifier is indicated, than the member will react like if it were a *private* member of the class for the instances of the classes defined outside the ***package***, meaning that they will not be able to access these members directly.

For the instance of the classes of the same ***package*** or instance of the classes that were defined in the same physical file as the class, these members will react like they were public, meaning that direct access to them will be allowed.

protected:

A protected member of a parent class can be access normally from instances of the sub-classes.

A protected method or attribute can be accessed from the same package.