1. Introduction

System

- A system is an arrangement where all its component work according to the specific defined rules. It is a method of organizing, working, or performing one or more tasks according to a fixed plan.

Embedded System

- **Embedded System:** which carries out a defined function and is embedded in a physical environment, is optionally surrounded by other subsystems and has an optional user interface.

Embedded System

- It is mostly designed for a specific function or functions within a larger system. For example, a fire alarm is a common example of an embedded system which can sense only smoke.

Embedded Software

- **Embedded Software:** is a piece of software that is embedded in hardware or non-PC devices. It is written specifically for the particular hardware that it runs on and usually has processing and memory constraints because of the device's limited computing capabilities.

Embedded Software

- Examples of embedded software include those found in dedicated GPS devices, factory robots, some calculators and even modern smartwatches

Design (activity)

- The activity that defines how a system is built from several components (architecture elements).

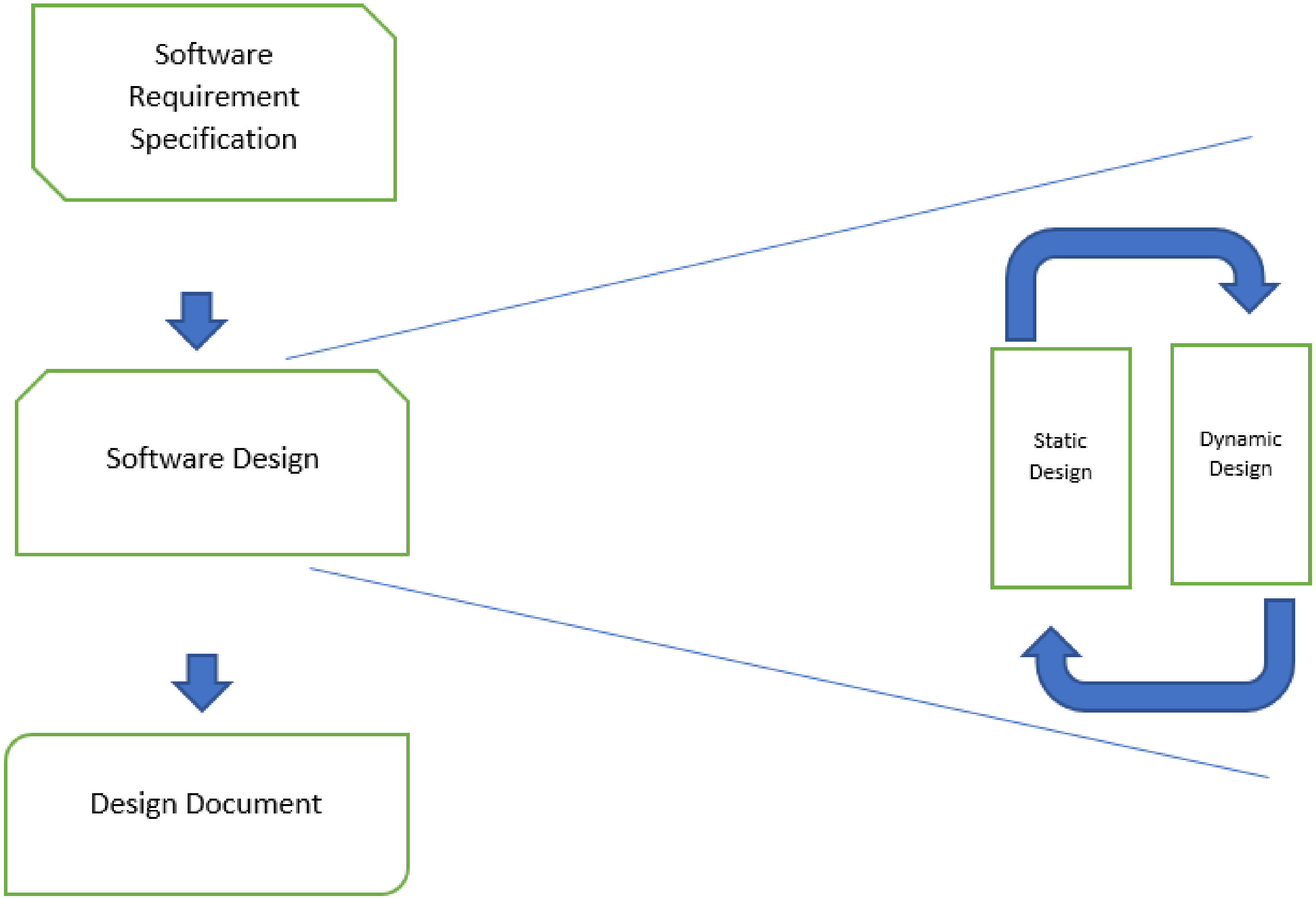
System Design

- with the specification that it is the design of a system, i.e. various elements such as hardware, software, mechanics.

Software Design

- with the specification that it is the design of a software (software elements).

Design scope



Design scope

- In the static design, you define the structure, the structure includes modules and how they connected. On the other hand, the dynamic design defines how the modules interact together.

Design scope

- Static design is responsible for executing the correct function, on the other hand, the dynamic design enables the function to be executed at the correct time.

Objective of the course

1. To build and analyze models for embedded application using the concept of UML.

2. To work with UML tools and represent the model using suitable diagrams.

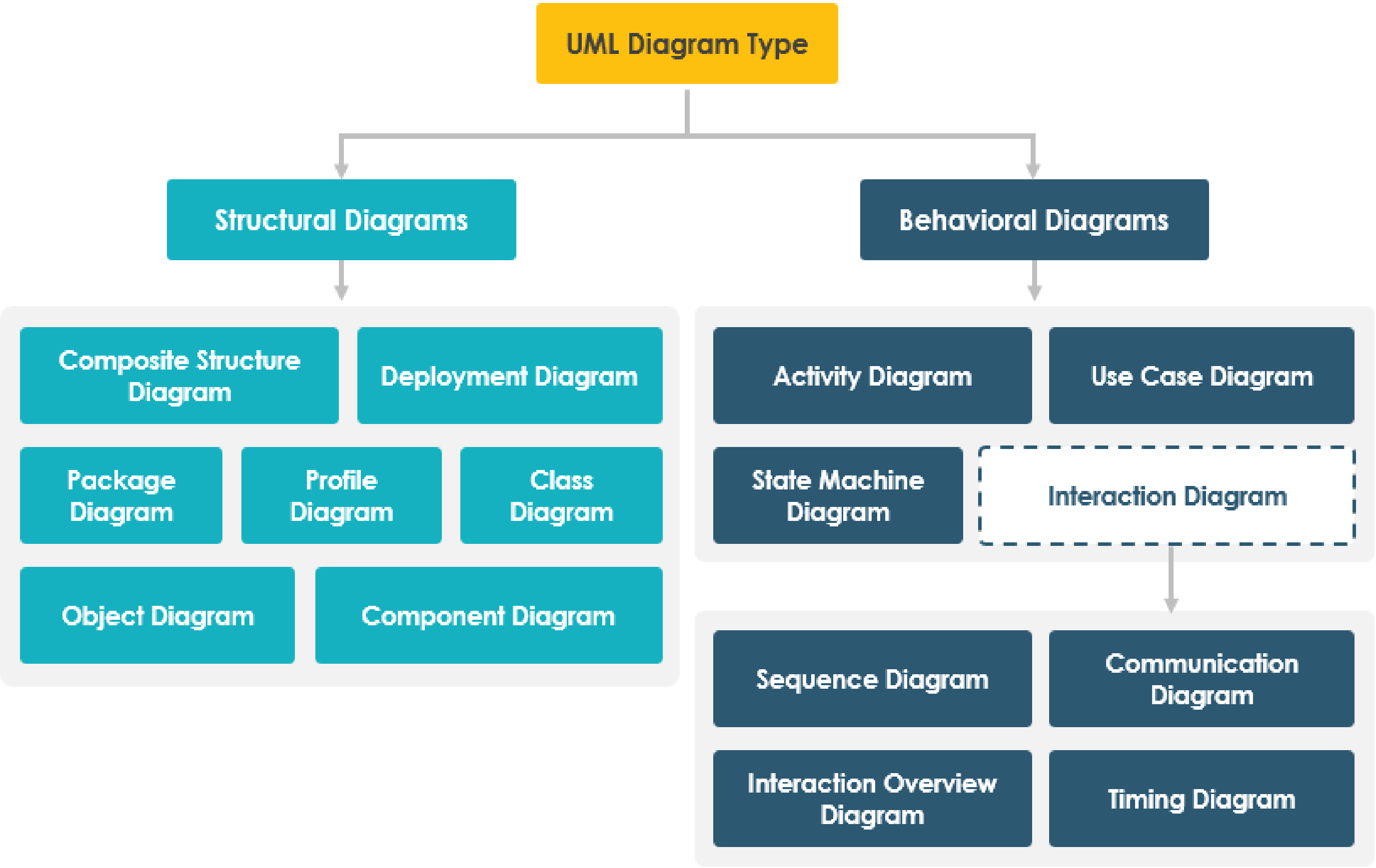
3. To write applications using the OOP concepts

4. To write applications using JAVA constructs for general purpose and embedded systems

UML

- It is a generic developmental modelling language used for analysis, design and implementation of software systems. The purpose of UML is to provide a simple and common method to visualize a software system's inherent architectural properties.

UML



UML Diagram Type

Structural Diagrams

Composite Structure Diagram

Deployment Diagram

Package Diagram

Profile Diagram

Class Diagram

Object Diagram

Component Diagram

Behavioral Diagrams

Activity Diagram

Use Case Diagram

State Machine Diagram

Interaction Diagram

Sequence Diagram

Communication Diagram

Interaction Overview Diagram

Timing Diagram

UML

- **Structural (Static) view:**

emphasizes the static structure of the system using objects, attributes, operations and relationships. It includes class diagrams and composite structure diagrams. Its specify the structure of the object.

UML

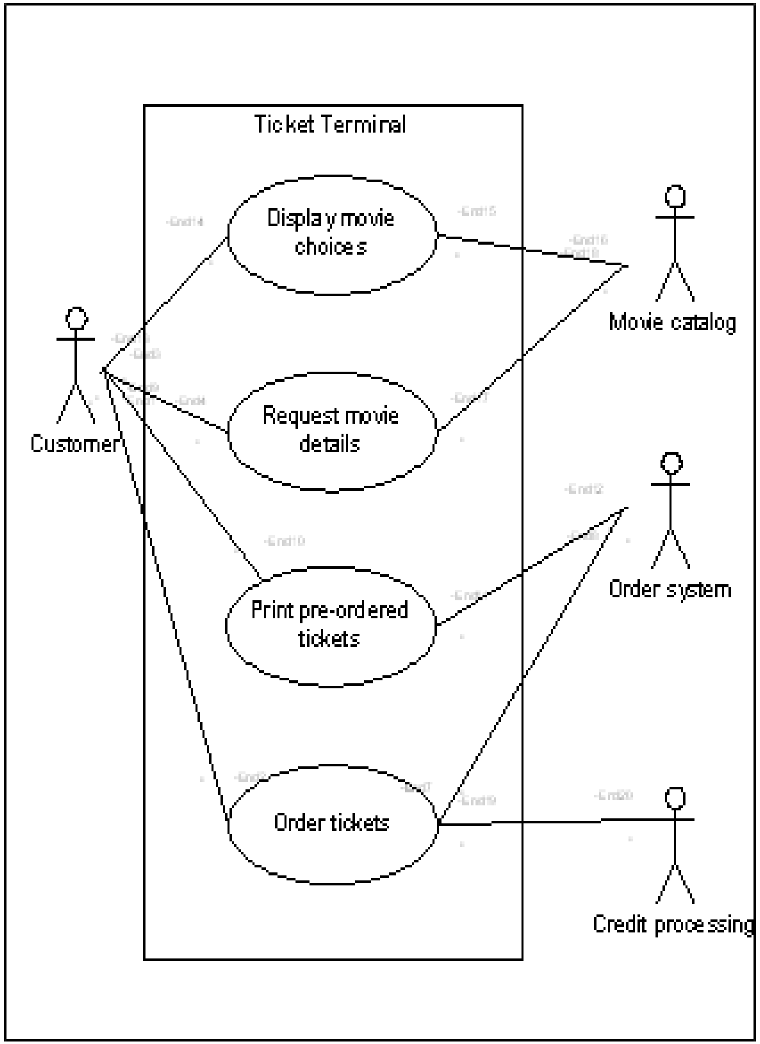
- **Behavioral(Dynamic) view:**

emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams, and state machine diagrams. Its represent the object interaction during runtime.

UML

- If you are describing what the program is able to *do*, you might write user stories, then draw out **use case diagrams** to elaborate on them.

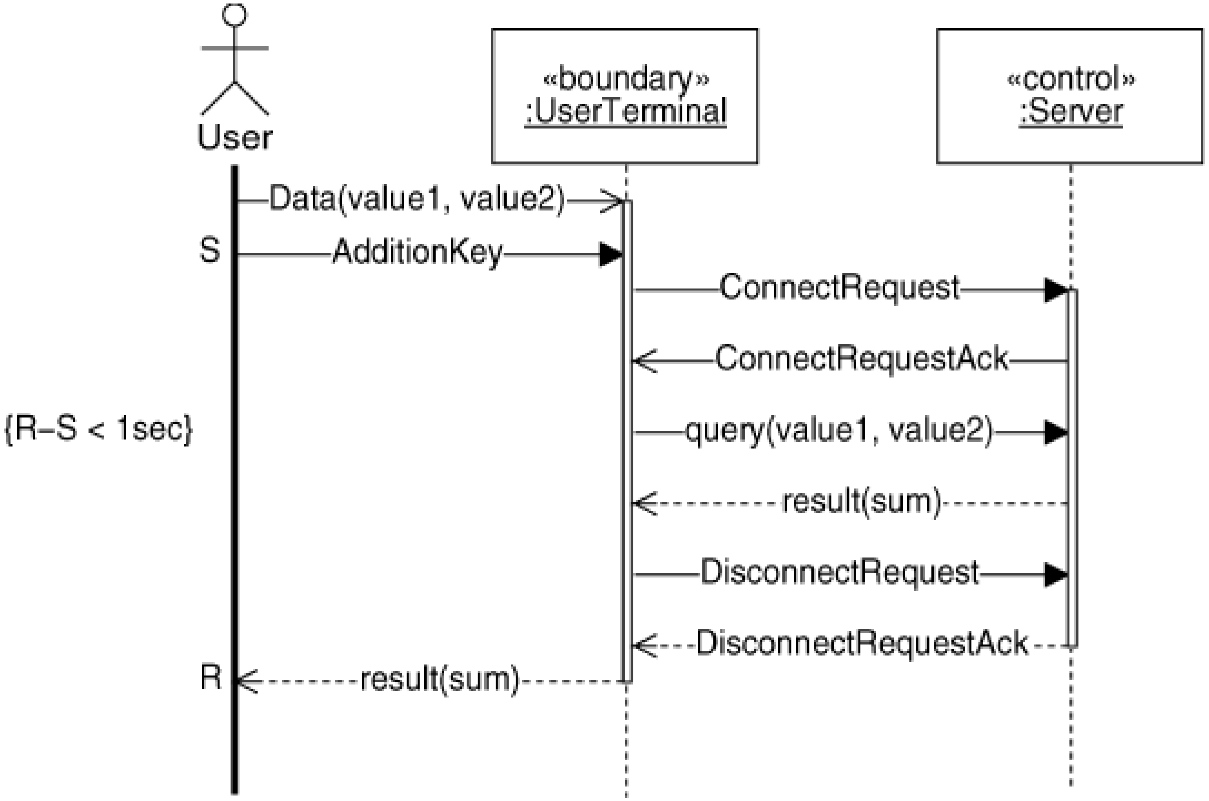
UML


Ticket Terminal
Display movie choices
Request movie details
Print pre-ordered tickets
Order tickets
Customer
Movie catalog
Order system
Credit processing

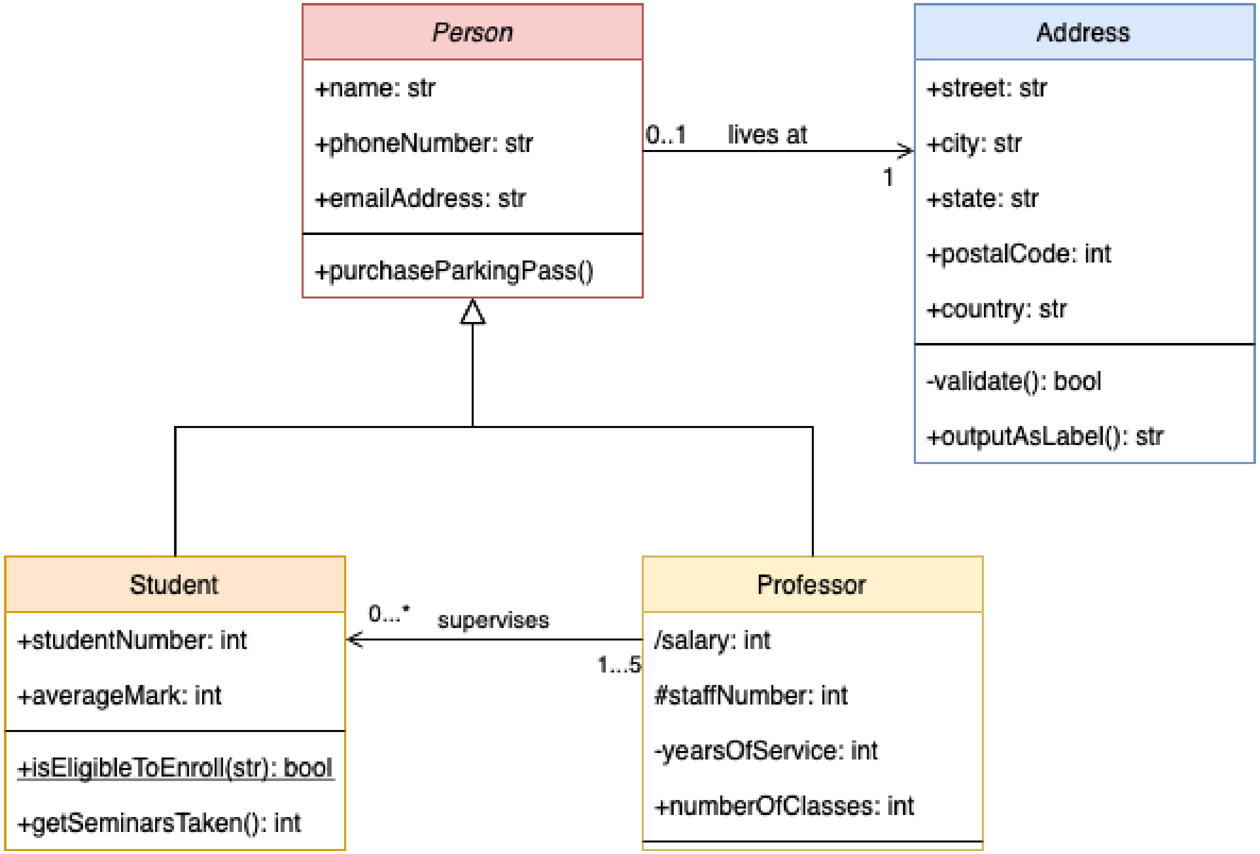
UML

- Once you have an idea what the program is able to do, you might design the structure of the program. Things like **sequence diagrams** and **class diagrams** help structure your program.

UML


User
«boundary»
:UserTerminal
«control»
:Server
Data(value1, value2)
S
AdditionKey
ConnectRequest
ConnectRequestAck
{R–S < 1sec}
query(value1, value2)
result(sum)
DisconnectRequest
DisconnectRequestAck
R
result(sum)

UML



Person

+name: str

+phoneNumber: str

+emailAddress: str

+purchaseParkingPass()

0..1 lives at 1

Address

+street: str

+city: str

+state: str

+postalCode: int

+country: str

-validate(): bool

+outputAsLabel(): str

Student

+studentNumber: int

+averageMark: int

+isEligibleToEnroll(str): bool

+getSeminarsTaken(): int

0...* supervises 1...5

Professor

/salary: int

#staffNumber: int

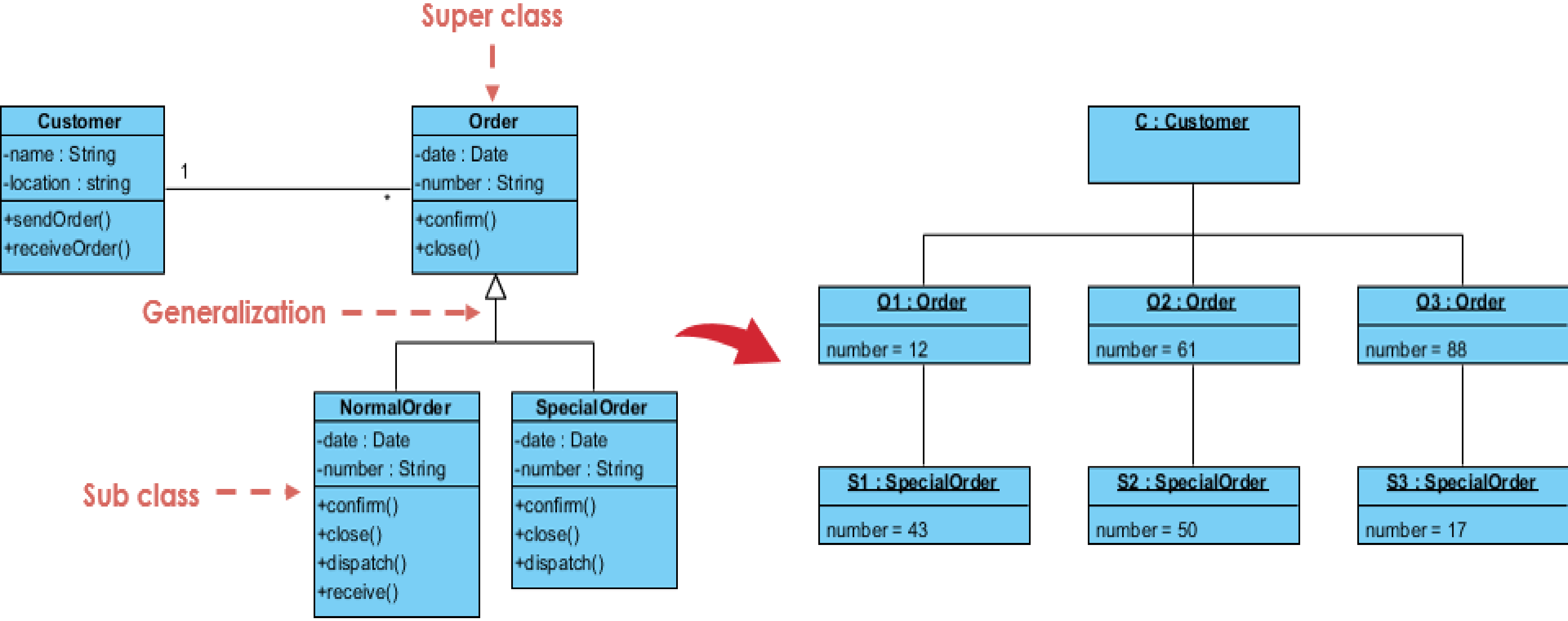
-yearsOfService: int

+numberOfClasses: int
)

UML

- Describing the state of a program's execution might help explain a program's implementation of a feature, so **object diagrams** help in this by giving a snapshot of a program's state.

UML



Super class

Customer
-name : String
-location : string
+sendOrder()
+receiveOrder()

1

*

Order
-date : Date
-number : String
+confirm()
+close()

Generalization

NormalOrder
-date : Date
-number : String
+confirm()
+close()
+dispatch()
+receive()

SpecialOrder
-date : Date
-number : String
+confirm()
+close()
+dispatch()

Sub class

C : Customer

O1 : Order
number = 12

O2 : Order
number = 61

O3 : Order
number = 88

S1 : SpecialOrder
number = 43

S2 : SpecialOrder
number = 50

S3 : SpecialOrder
number = 17

2. OOP/Class Diagram

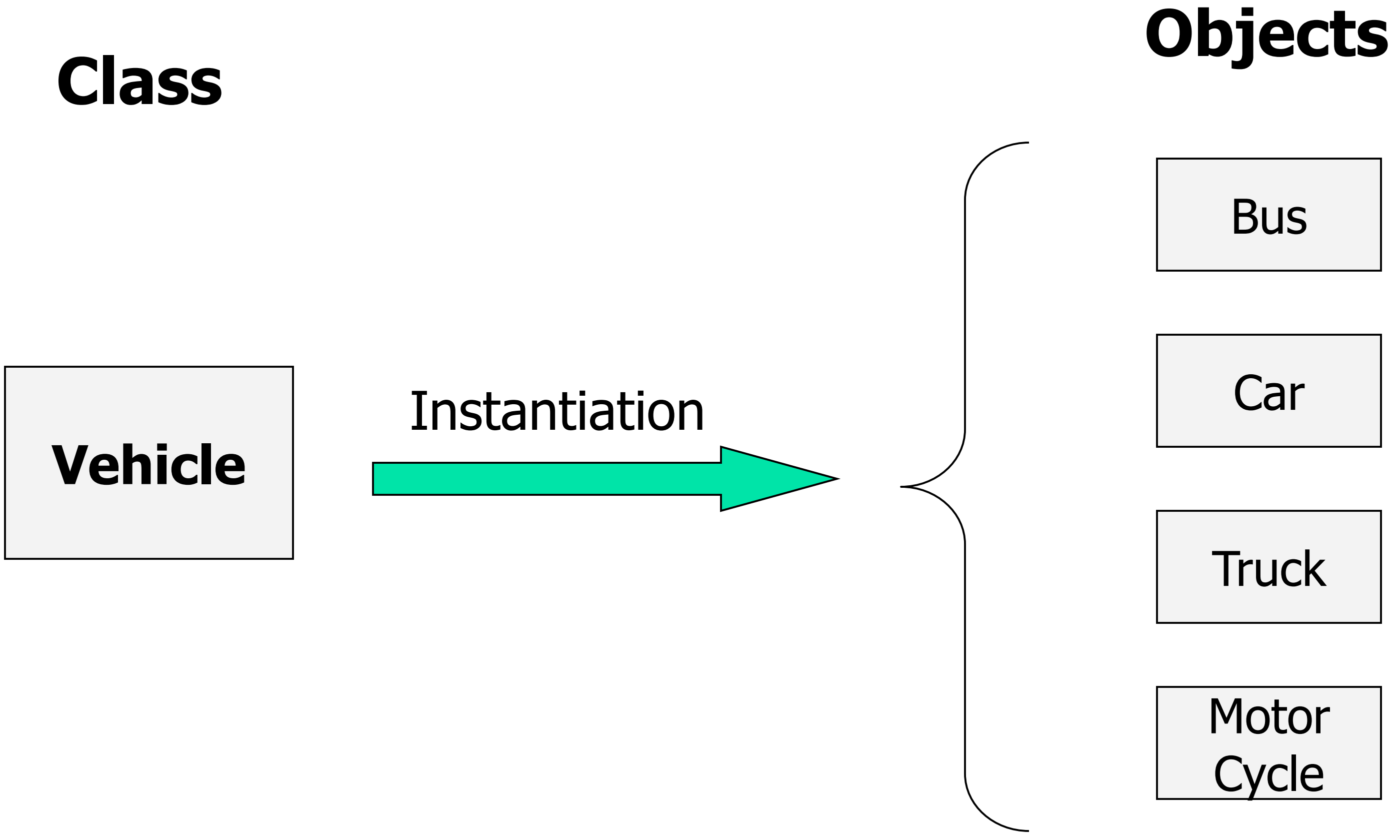
Major Concepts of OOP

1. Class/Objects
2. Abstraction
3. Encapsulation
4. Inheritance
5. Polymorphism

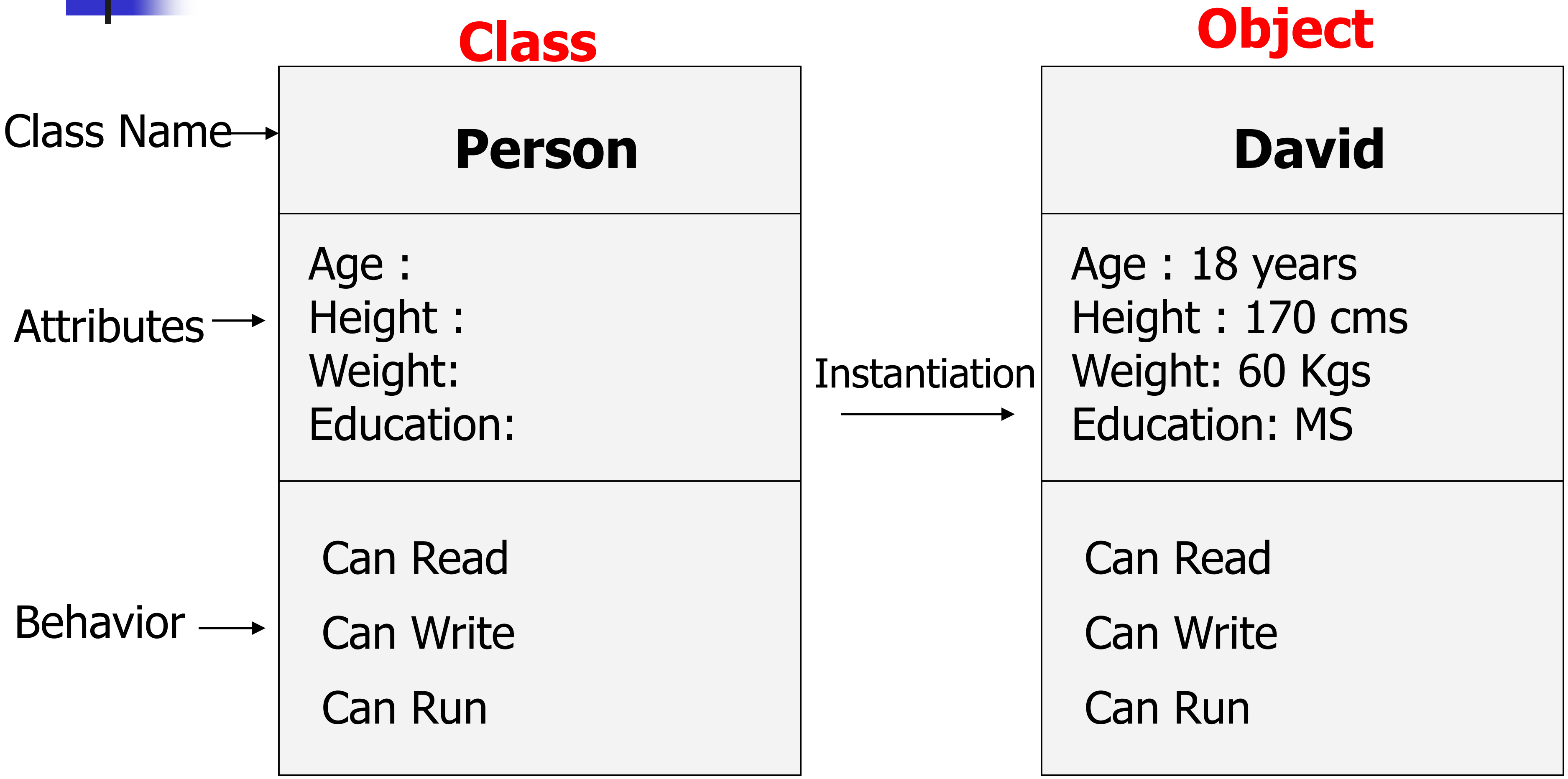
Objects and Classes

- Object is an instance of a Class. That is, every Object has a Class.
- Object has a unique identity. Two objects of a same class are distinguishable.
- A Class describes a group of Objects with similar properties (attributes), common behavior (operation).

Objects and Classes



A Class of an Object



Class

Object

Class Name

Person

David

Attributes

Age :
Height :
Weight:
Education:

Instantiation

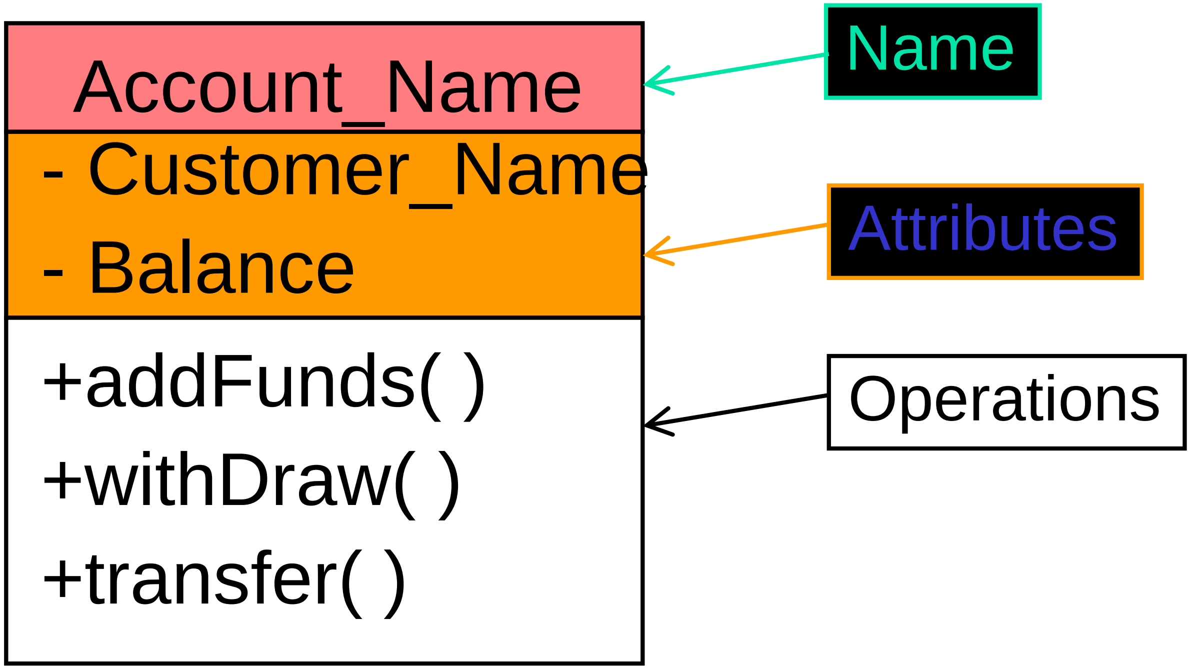
Age : 18 years
Height : 170 cms
Weight: 60 Kgs
Education: MS

Behavior

Can Read
Can Write
Can Run

Can Read
Can Write
Can Run

Class diagram



Class diagram

- A class diagram depicts classes and their interrelationships
- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

Class diagram

- Each class is represented by a rectangle subdivided into three compartments
  - Name
  - Attributes
  - Operations

- Modifiers are used to indicate visibility of attributes and operations.
  - ‘+’ is used to denote *Public* visibility (everyone)
  - ‘#’ is used to denote *Protected* visibility (friends and derived)
  - ‘-’ is used to denote *Private* visibility (no one)

- By default, attributes are hidden and operations are visible.

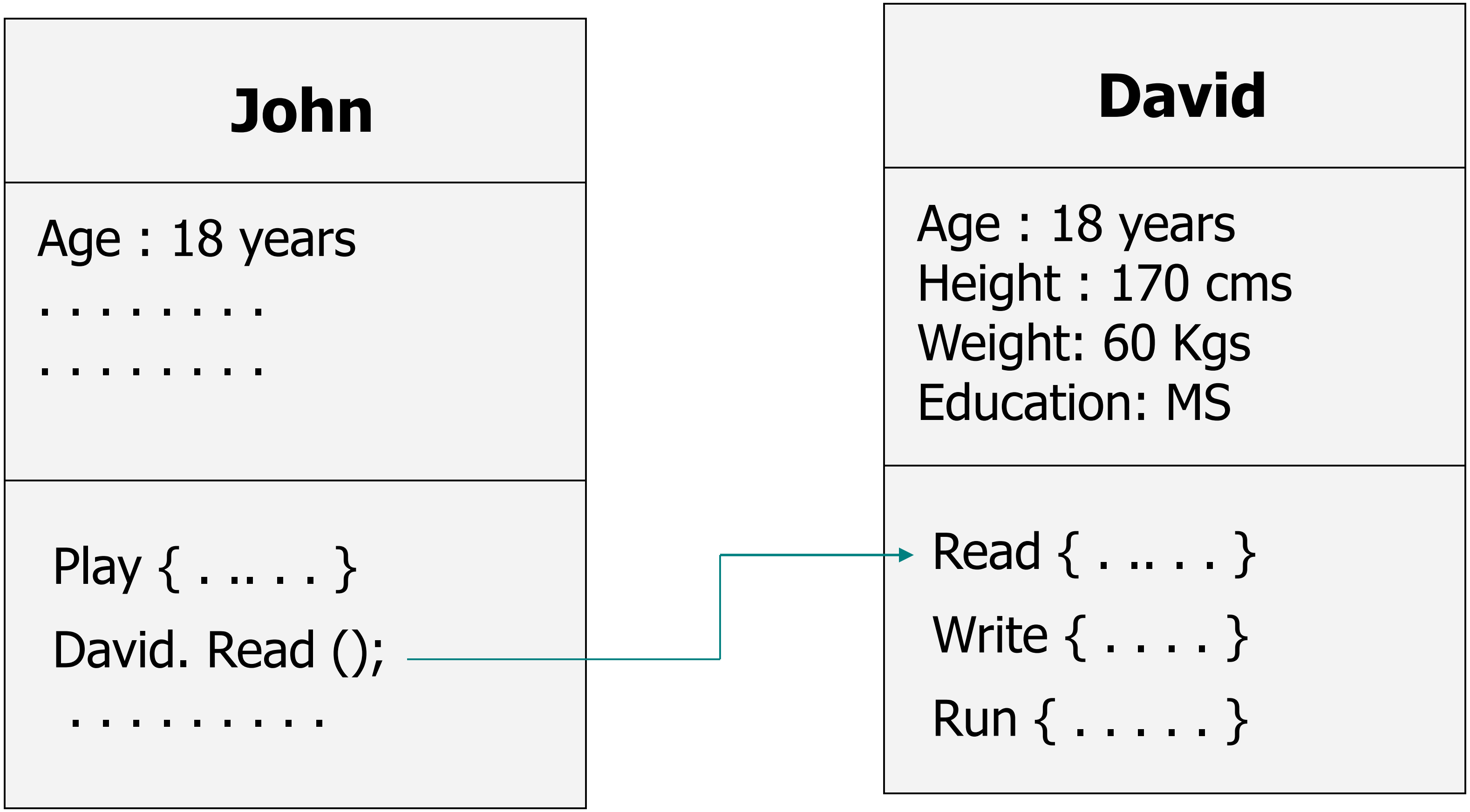
Abstraction

- In OOP, you can abstract the implementation details of a class and present a clean, easy-to-use interface through the class member functions.
- Abstract classes and interfaces are used to hide the internal details and show the functionality. It focuses on ideas rather than events, the user will get to understand of “what” than “how”.

Encapsulation

- Also known as ***Data Hiding***.
- Separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects.
- Since data and behavior are combined in a single entity, this makes encapsulation cleaner and more powerful.

Encapsulation



John

Age : 18 years

. . . .

. . . .

Play { . .. . . }

David. Read ();

. . . .

David

Age : 18 years
Height : 170 cms
Weight: 60 Kgs
Education: MS

Read { . .. . . }

Write { . . . . }

Run { . . . . . }

Abstraction Vs Encapsulation

- In the popular programming text Object-Oriented Analysis and Design, Grady Booch writes that:

"Abstraction and encapsulation are complementary concepts: ***abstraction focuses on the observable behavior of an object...encapsulation focuses on the implementation that gives rise to this behavior****"*

Abstraction Vs Encapsulation

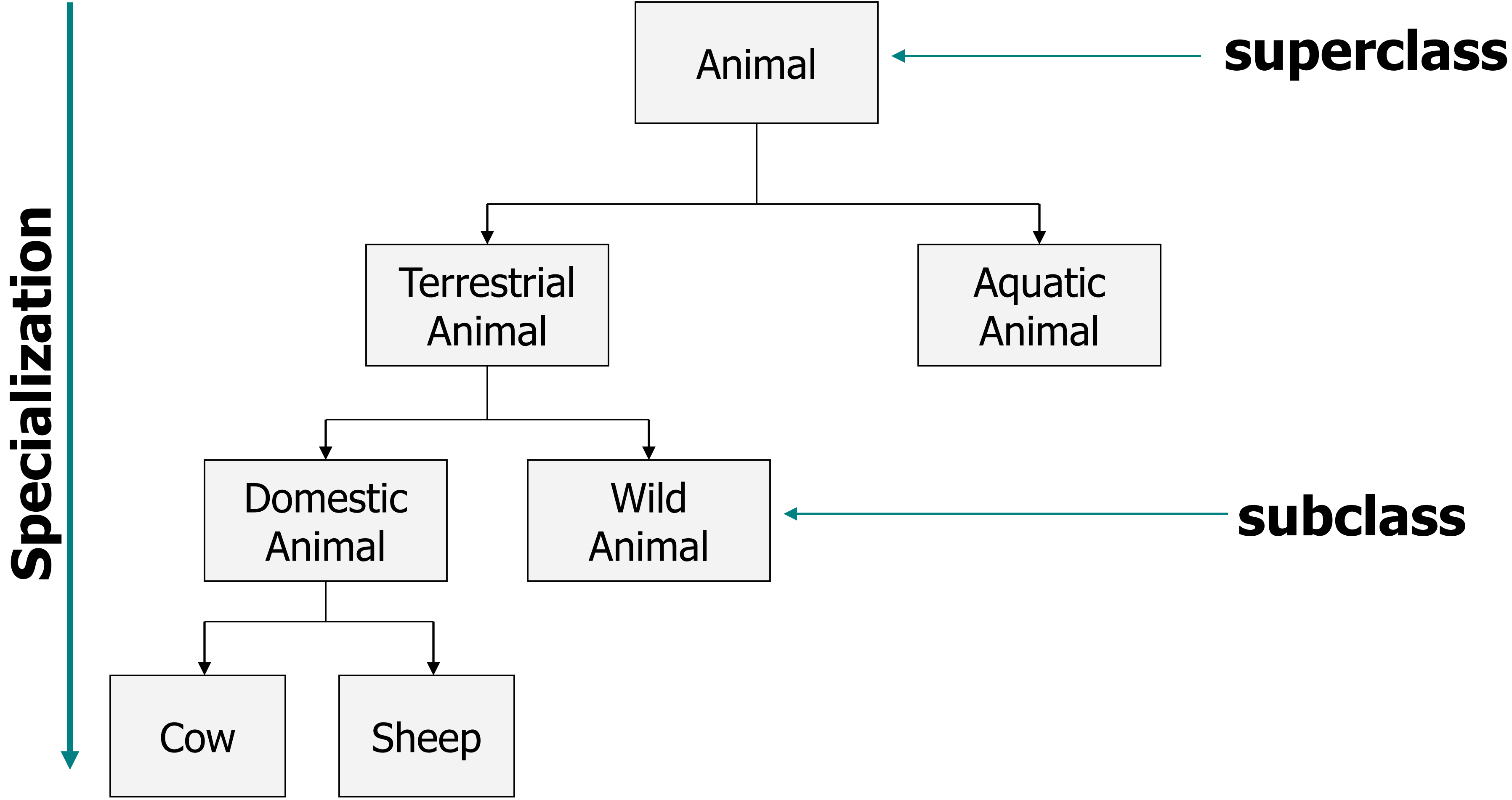
- Stated differently, an abstraction relates to how an object and its behaviors are presented to the user and encapsulation is a methodology that helps create that experience.

Inheritance

- Inheritance is a powerful abstraction for sharing similarities among classes while preserving their differences.

- This is the relationship between a class and one or more refined versions of it.
  - The class being refined is called the ***superclass***
  - Each refined version is called ***subclass***

Inheritance

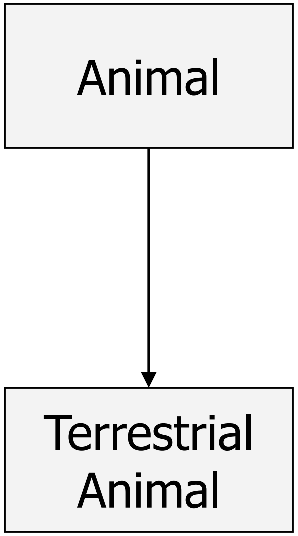


Inheritance

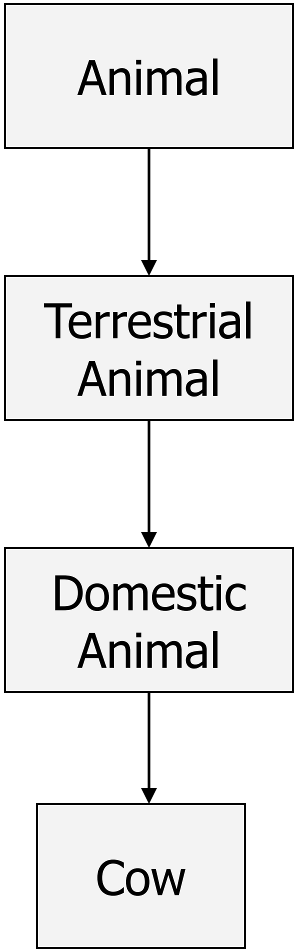
Types of Inheritance:

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance

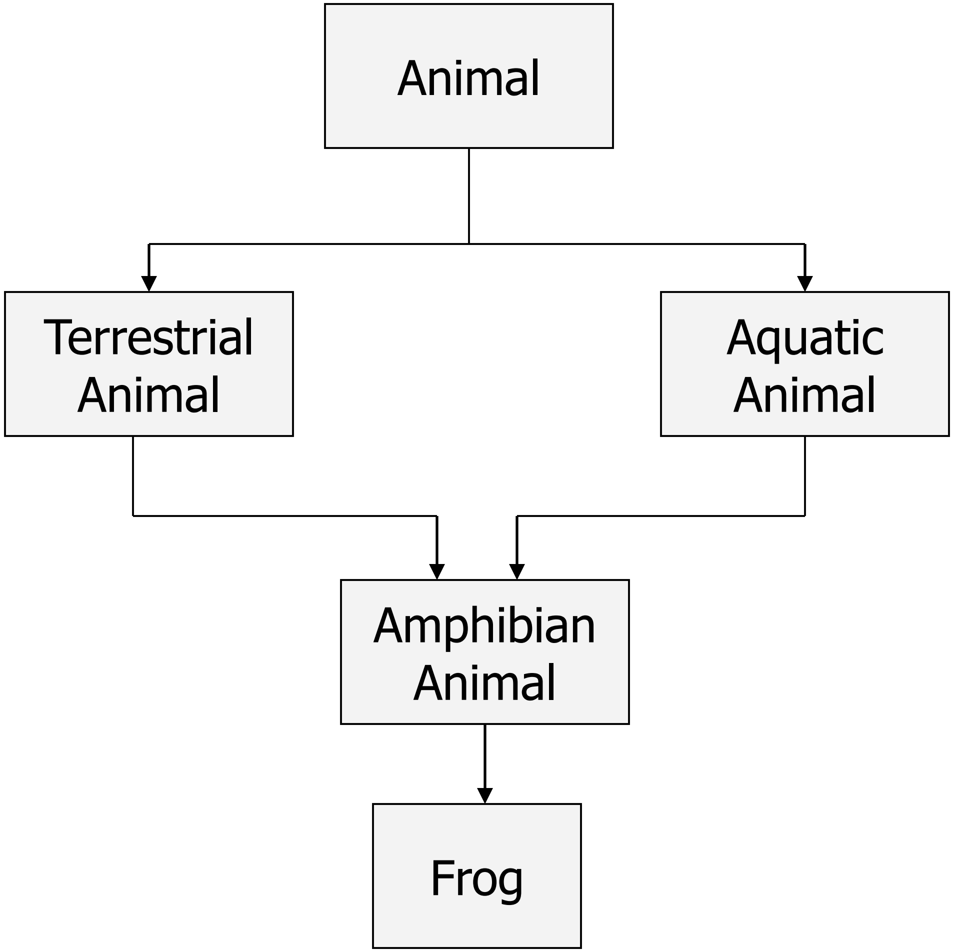
Single Inheritance



Multilevel Inheritance



Multiple Inheritance



Polymorphism

- *Polymorphism* means the ability to take more than one form.
- An operation may exhibit different behaviors in different instances. The behavior depends on the data types used in the operation.
- Polymorphism is extensively used in implementing Inheritance.

03. Class concepts – (Java)

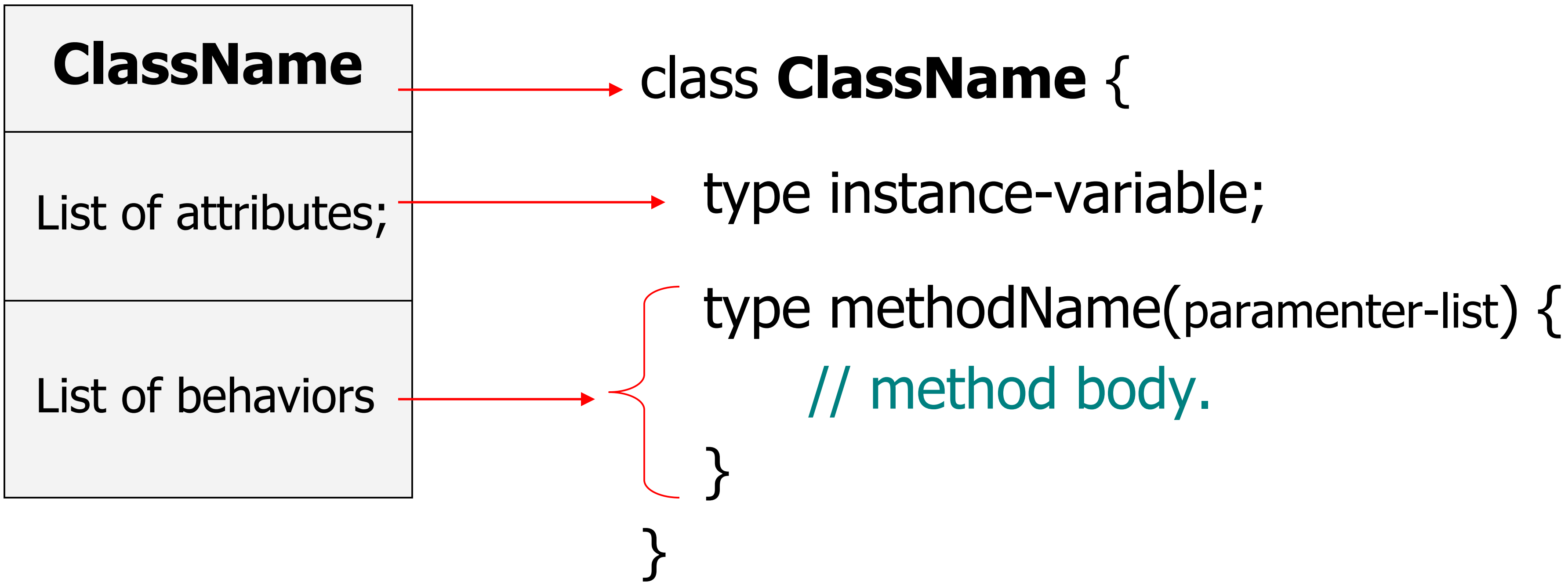
Class fundamentals
Methods
Constructors
Access Modifiers
Inner Classes

Class Fundamentals

- What is a Class?
  - A Class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.

The General Form of a Class

UML Class Diagram **Java Representation**



ClassName

List of attributes;

List of behaviors

```
class ClassName {
    type instance-variable;
    type methodName(paramenter-list) {
        // method body.
    }
}
```

Instance Variable

- Any item of data that is associated with a particular object. Each object has its own copy of the instance variables defined in the class. Also called a field.

Instance Method

- Any method that is invoked with respect to an instance of a class. Also called simply a method.

class variable

- A data item associated with a particular class as a whole--not with particular instances of the class. Class variables are defined in class definitions. Also called a *static field*.

class method

- A method that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class. Also called a *static method*.

A Simple Class

```
class Box {
    double width;
    double height;
    double depth;
}
```

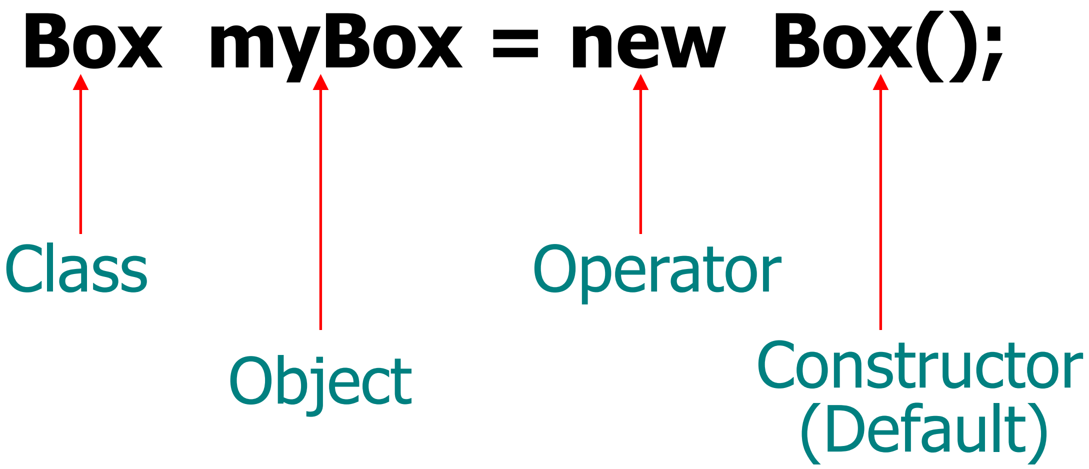
Object Creation - instantiation

```
class Box {
        double width;
        double height;
        double depth;
}
```

```
public class BoxDemo {
    public static void main(String args[]) {
        Box myBox = new Box();
        double vol;
        myBox.width = 10;
        myBox.height = 20;
        myBox.depth = 15;
        vol = myBox.width * myBox.depth
                * myBox.height;
        System.out.println("Volume is:" +vol);
    }
}
```

This way we can create any number of objects of Box say myBox1, myBox2, myBox3, …

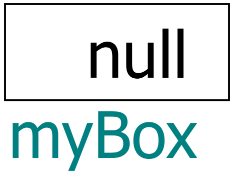
Class Instantiation Statement:



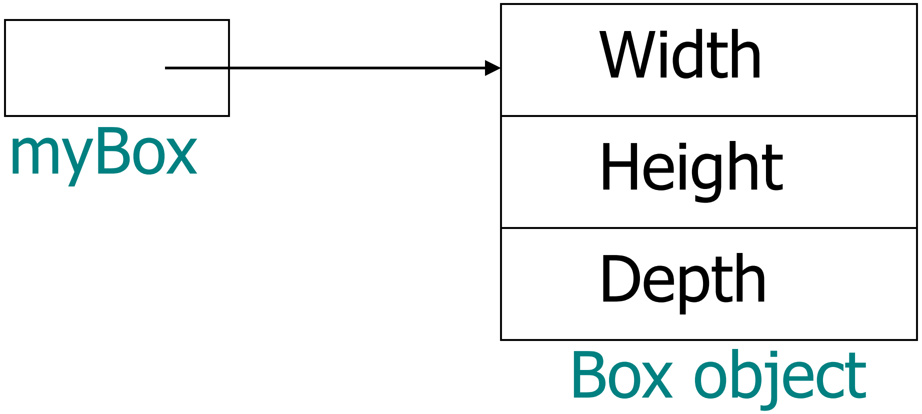
Box myBox = new Box();

Instantiation is a 2 step process:

```
Box  myBox;
```

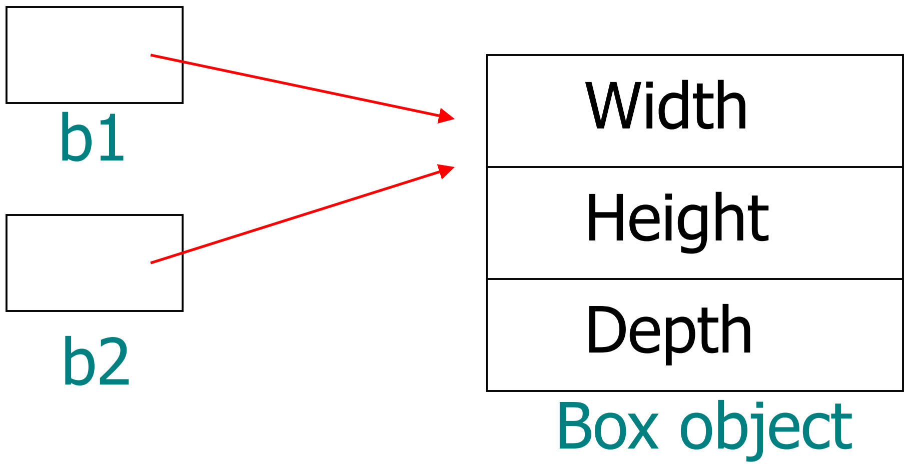


```
myBox = new  Box();
```



Object Reference Assignment

```
Box b1 = new  Box();

Box b2 = b1;
```



- Both b1 and b2 refer to the same object and not two distinct objects, but they are not linked in any other way.

Methods

- Method signature:

```
return_type methodName(parameter_list) {
    // method body
}
```

- ***return_type*** specifies the type of data returned by the method. If the method does not return a value, its return type must be ***void***.
- ***methodName*** – any legal identifier other than the keywords.

- ***parameter_list*** – sequence of type and identifier pairs separated by commas.
- Methods that have a return type other than void return a value to the calling routine using a return statement as given below:

  return value;

  Here value is the value returned.

Box Class - Adding a method

```
class Box  {
        double width;
        double height;
        double depth;
        //  Instance method.
        void getVolume()  {
                System.out.print("Volume is: ");
                System.out.println(width*height*depth);
        }
}
```

Instance variables

```
public class BoxDemo  {
    public static void main(String args[])  {
        Box myBox1 = new Box();
        Box myBox2 = new Box();

        myBox1.width = 10;
        myBox1.height = 20;
        myBox1.depth = 15;

        myBox2.width = 3;
        myBox2.height = 6;
        myBox2.depth = 9;

        myBox1.getVolume();

        myBox2.getVolume();
    }
}
```

Method returning a value

```
class Box  {
        double width;
        double height;
        double depth;

        //  Instance method.
        double getVolume()  {
                double volume = width*height*depth;
                return volume;
        }
}
```

```
public class BoxDemo  {
    public static void main(String args[])  {
          Box myBox1 = new Box();
          double volume;

          myBox1.width = 10;
          myBox1.height = 20;
          myBox1.depth = 15;

          volume = myBox1.getVolume();
          System.out.println(“Volume is: “ + volume);

    }
}
```

Method that takes a parameter

```
class Box  {
        double width;
        double height;
        double depth;

        //  Instance method.
        double getVolume()  {
                return width*height*depth;
        }

        void setDim( double w, double h, double d)  {
                width = w;
                height = h;
                depth = d;
        }
}
```

Formal parameters

```
public class BoxDemo  {

    public static void main(String args[])  {

        Box myBox1 = new Box();
        double volume;

        myBox1.setDim(10, 20, 15);

        volume = myBox1.getVolume();
        System.out.println(“Volume is: “ + volume);

    }
}
```

Actual parameters

Constructors

- Instead of using a separate method for initializing an object during its creation, it is more convenient and concise to initialize them automatically when they are created.

- This automatic initialization is done by a special method called a ***constructor***.

- A ***constructor*** is special because it does not have a return type, not even void.

```
class Box  {
        double width;
        double height;
        double depth;

        //  Constructor
        Box()  {
                width = 10;
                height = 10;
                depth = 10;
        }
}
```

Instantiation:

```
        Box myBox = new Box();
```

```
class Box  {
        double width;
        double height;
        double depth;
        //  Constructor
        Box(double w, double h, double d)  {
                width = w;
                height = h;
                depth = d;
        }
        //  Instance method.
        double getVolume()  {
                return width*height*depth;
        }
}
```

```
public class BoxDemo  {
    public static void main(String args[])  {
          Box myBox1 = new Box(10, 20, 15);
          Box myBox2 = new Box(3, 6, 9);
          double volume;

          volume = myBox1.getVolume();
          System.out.println(“Volume is: “ + volume);

          volume = myBox2.getVolume();
          System.out.println(“Volume is: “ + volume);

    }
}
```

Constructors

- Instead of using a separate method for initializing an object during its creation, it is more convenient and concise to initialize them automatically when they are created.
- This automatic initialization is done by a special method called a ***constructor***.
- A ***constructor*** is special because it does not have a return type, not even void.

```
class Box  {
        double width;
        double height;
        double depth;

        //  Constructor
        Box()  {
                width = 10;
                height = 10;
                depth = 10;
        }
}
```

Instantiation:

```
        Box myBox = new Box();
```

```
class Box  {
        double width;
        double height;
        double depth;
        //  Constructor
        Box(double w, double h, double d)  {
                width = w;
                height = h;
                depth = d;
        }
        //  Instance method.
        double getVolume()  {
                return width*height*depth;
        }
}
```

```
public class BoxDemo  {
   public static void main(String args[])  {
         Box myBox1 = new Box(10, 20, 15);
         Box myBox2 = new Box(3, 6, 9);
         double volume;

         volume = myBox1.getVolume();
         System.out.println(“Volume is: “ + volume);

         volume = myBox2.getVolume();
         System.out.println(“Volume is: “ + volume);

   }
}
```

Method Overloading

- Defining two or more methods within the same class that share the same name, as long as their parameter declarations are different is called method overloading.
- This is the way Java implements polymorphism.

```
class OverloadDemo  {

    void test() {
            System.out.println(“No parameters”);
    }

    void test(int a) {
            System.out.println(“a: ” +a);
    }
                                            // double test(int a) {. . . } - Wrong
    void test(int a, int b) {
            System.out.println(“a and b: ” +a + “ “ +b);
    }

    double test(double a) {
            System.out.println(“double a: ” +a);
            return a*a;
    }
}
```

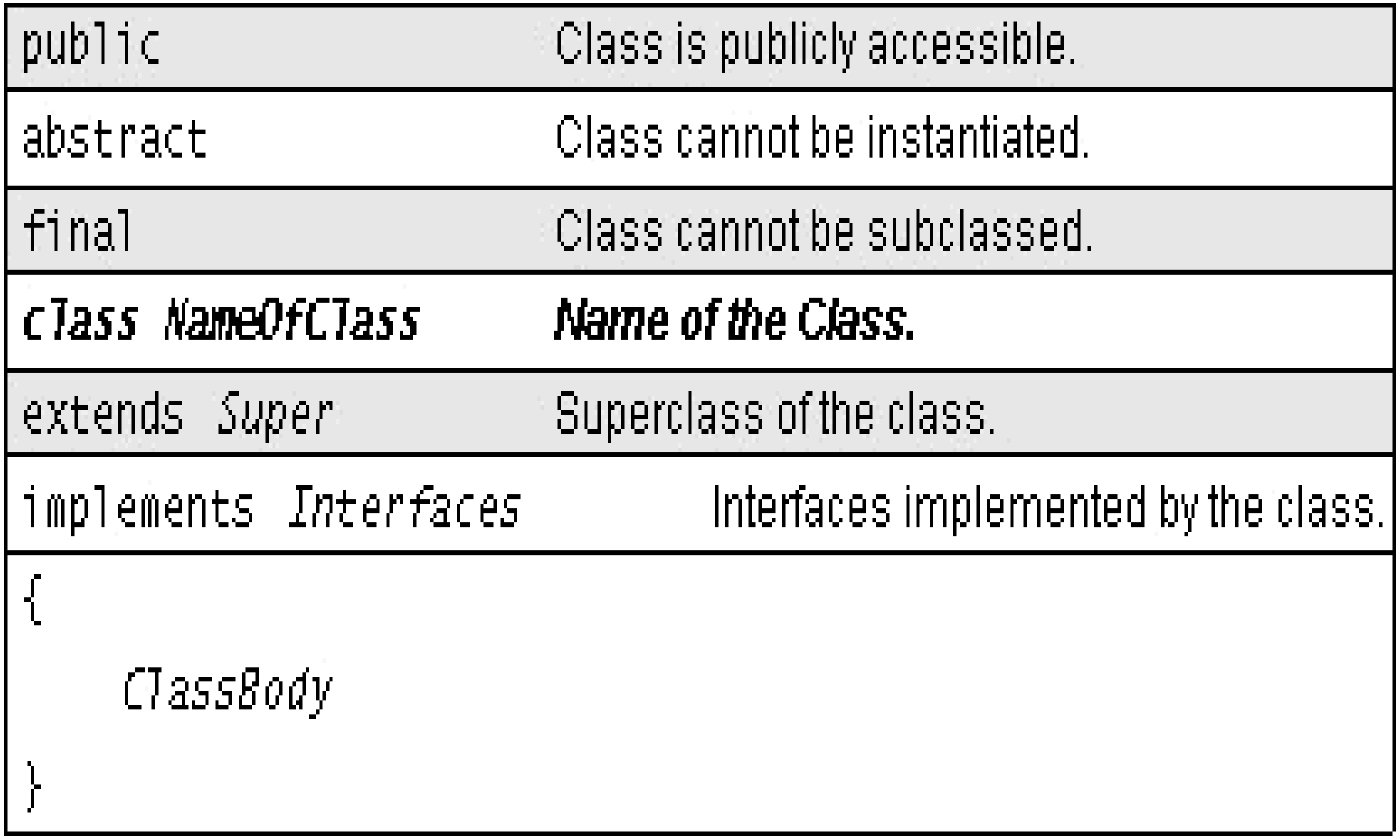
```
class Overload  {
   public static void main(String args[])  {
        OverloadDemo ob = new OverloadDemo();
        double result;

        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): "
                                    +result);
   }
}
```

Constructor Overloading

- Constructors can also be overloaded.

```
Box(double w, double h, double d)  {
        width = w;   height = h;   depth = d;
}

Box()  {
        width = -1;   height = -1;   depth = -1;
 }

Box(double len)  {
        width = height = depth = len;
}
```

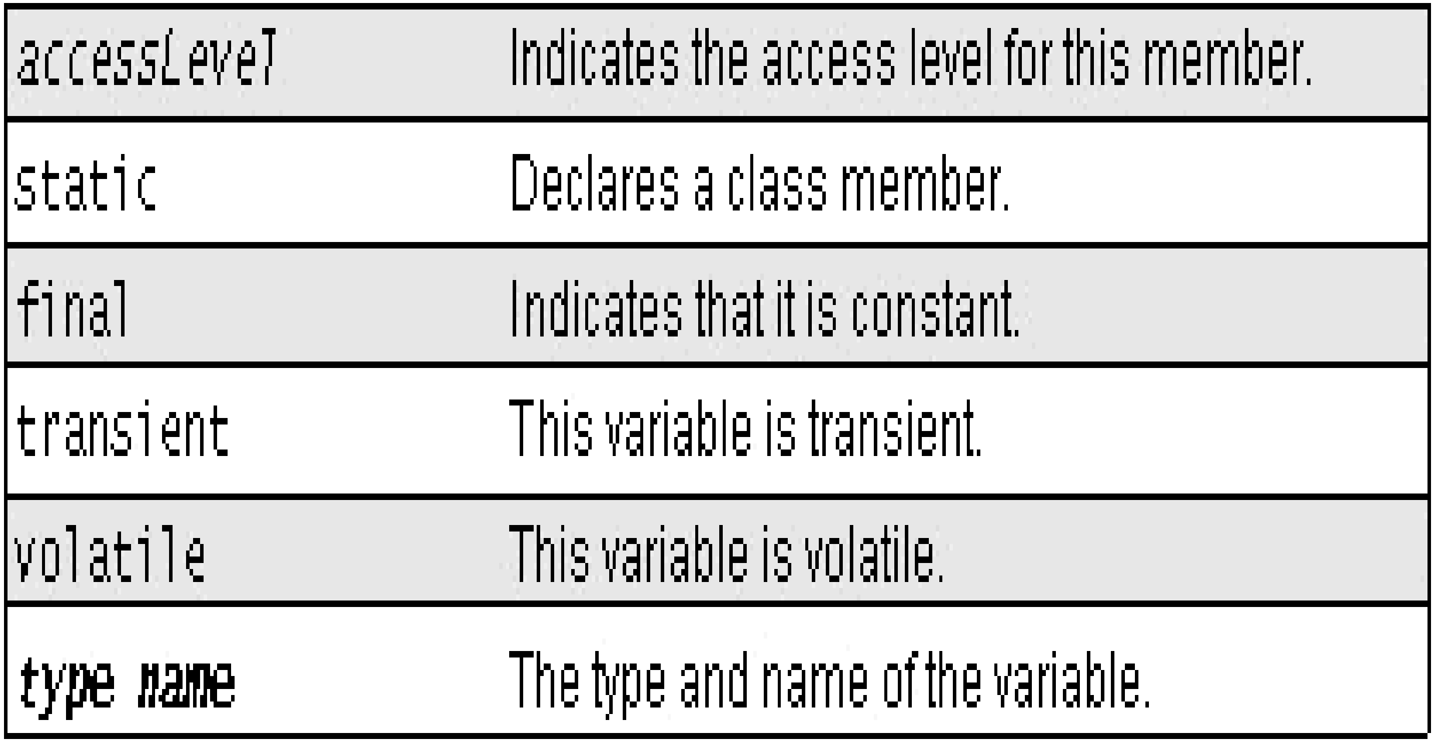
The Class Declaration



public	Class is publicly accessible.
abstract	Class cannot be instantiated.
final	Class cannot be subclassed.
class *NameOfClass*	***Name of the Class.***
extends *Super*	Superclass of the class.
implements *Interfaces*	Interfaces implemented by the class.

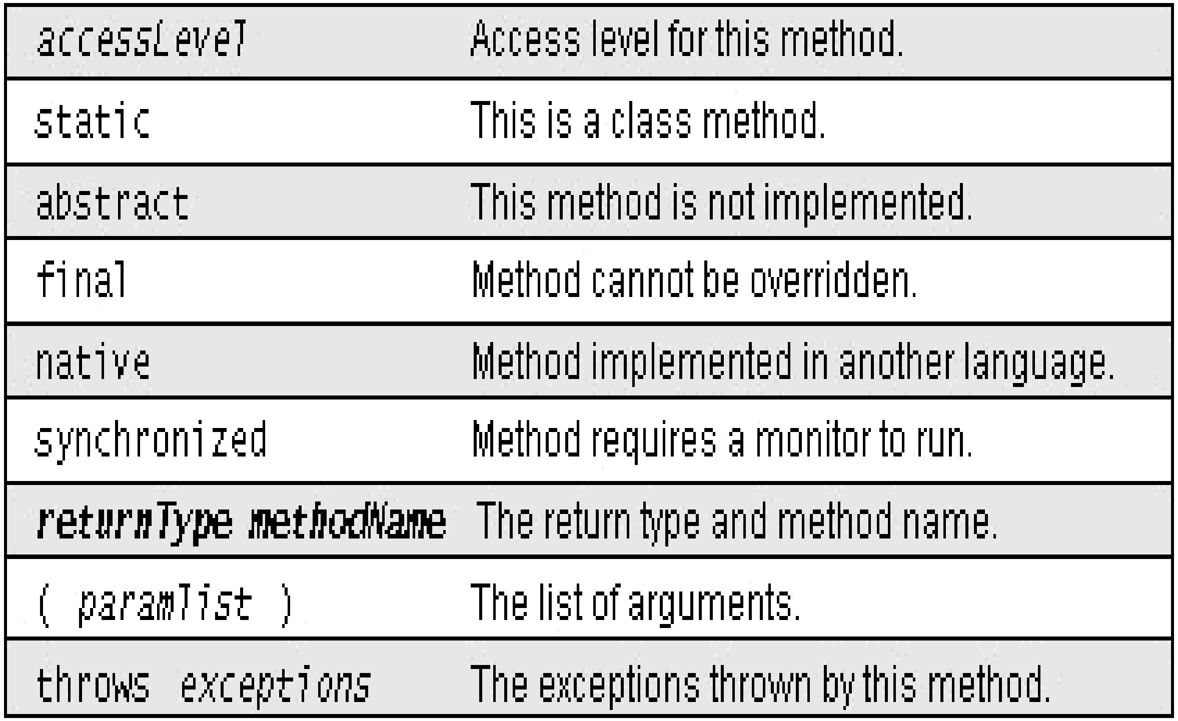
{
ClassBody
}	

Declaring Member Variables



accessLevel	Indicates the access level for this member.
static	Declares a class member.
final	Indicates that it is constant.
transient	This variable is transient.
volatile	This variable is volatile.
type name	The type and name of the variable.

Details of a Method Declaration



accessLevel	Access level for this method.
static	This is a class method.
abstract	This method is not implemented.
final	Method cannot be overridden.
native	Method implemented in another language.
synchronized	Method requires a monitor to run.
returnType methodName	The return type and method name.
(*paramlist*)	The list of arguments.
throws *exceptions*	The exceptions thrown by this method.

'this' keyword

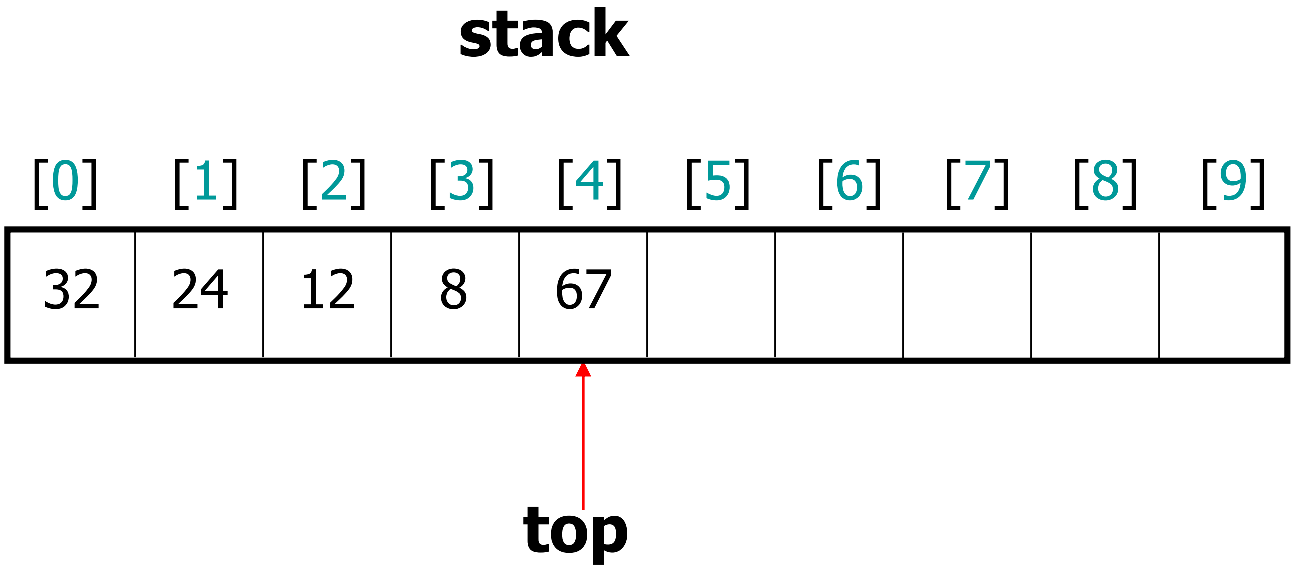
Local variables

```
Box(double width, double height, double depth)  {
        this.width = width;
        this.height = height;
        this.depth = depth;
}
```

Instance variables

Example: Stack implementation

- Implement a class **Stack** with 2 basic operations – ***push*** and ***pop***.



stack

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32	24	12	8	67					

top

Static Members

- A static class member can be accessed directly by the class name and doesn’t need any object. A single copy of a static member is maintained throughout the program regardless of the number of objects created.

- Static variables are initialized only once and at the start of the execution during the lifetime of a class. These variables will be initialized first before the initialization of any instance variables.

Static Members

Methods declared as static (class methods) have several restrictions:

- They can only call other static methods.
- They must only access static data.
- They cannot refer to *this* or *super* in anyway.
- These methods can be accessed using the class name rather than a object reference.
- *main()* method should be always static because it must be accessible for an application to run, before any instantiation takes place.
- When *main()* begins, no objects are created, so if you have a member data, you must create an object to access it.

Static methods/Data members

```
public class Print  {
    public static String name = “default”;
    public static void printName()      {
            System.out.println(name);
    }
    public static void main(String arg[])  {
            System.out.println(Print.name);
            Print.printName();
    }
}
```

```
class TrackObj
{
        //class variable
        private static int counter = 0;

        //instance variable
        private int x = 0;

        TrackObj()
        {
                counter++;
                x ++;
        }

        //member method
        public int getX()
        {
                return x;
        }

        //class method
        public static int getCounter()
        {
                return counter;
        }
}
```

03. Class concepts – (Java)

Class fundamentals
Methods
Constructors
Inner Classes

Static Members

- A static class member can be accessed directly by the class name and doesn’t need any object. A single copy of a static member is maintained throughout the program regardless of the number of objects created.
- Static variables are initialized only once and at the start of the execution during the lifetime of a class. These variables will be initialized first before the initialization of any instance variables.

Static Members

Methods declared as static (class methods) have several restrictions:

- They can only call other static methods.
- They must only access static data.
- They cannot refer to *this* or *super* in anyway.
- These methods can be accessed using the class name rather than a object reference.
- *main()* method should be always static because it must be accessible for an application to run, before any instantiation takes place.
- When *main()* begins, no objects are created, so if you have a member data, you must create an object to access it.

Static methods/Data members

```
public class Print  {

     public static String name = “default”;

     public static void printName()       {
               System.out.println(name);
     }

     public static void main(String arg[])  {
               System.out.println(Print.name);
               Print.printName();
     }
}
```

```
class TrackObj
{
        //class variable
        private static int counter = 0;

        //instance variable
        private int x = 0;

        TrackObj()
        {
                counter++;
                x ++;
        }

        //member method
        public int getX()
        {
                return x;
        }

        //class method
        public static int getCounter()
        {
                return counter;
        }
}
```

Access Modifiers

- Java provides a number of access modifiers to set the level of access for classes, fields, methods and constructors.
- A member has package or default accessibility when no accessibility modifier is specified.
- **Access Modifiers:**

  1. private 2. protected 3. default 4. public

private access modifier

- The *private* (most restrictive) access modifier is used for fields or methods and cannot be used for classes and Interfaces.
- It also cannot be used for fields and methods within an interface.
- Field, method declared private are strictly controlled, and that member can be accessed only by other members of that class.
- A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier

- Discussed later under the topic - Inheritance

default access modifier

- Java provides a default access modifier which is used when no access modifier is specified.
- Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package.
- The default modifier is not used for fields and methods within an interface.

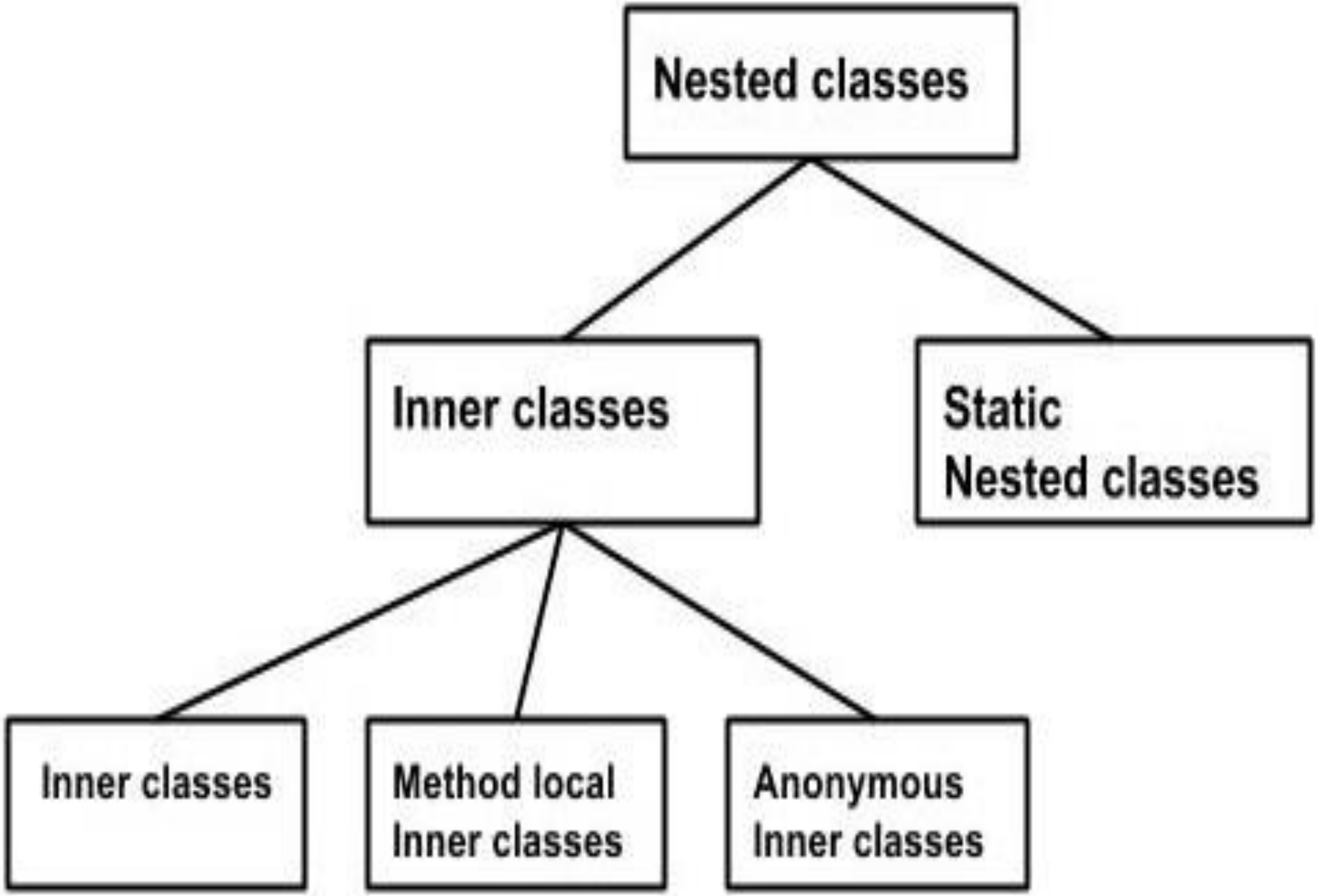
public access modifier

- Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

Access Levels

The following table shows the access to members permitted by each modifier.

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



Nested classes

Inner classes

Static
Nested classes

Inner classes

Method local
Inner classes

Anonymous
Inner classes

Nested Classes

- The Java programming language allows you to define a class within another class. Such a class is called a *nested class*.

```
class OuterClass {
    ...
    class NestedClass {
    ...
    }
}
```

Nested Classes

- Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*

Nested Classes

```
  class OuterClass {
       ...
       static class StaticNestedClass {
       ...
        }
       class InnerClass {
        ...
        }
}
```

Static Nested Classes

- As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class — it can use them only through an object reference.

Inner Classes

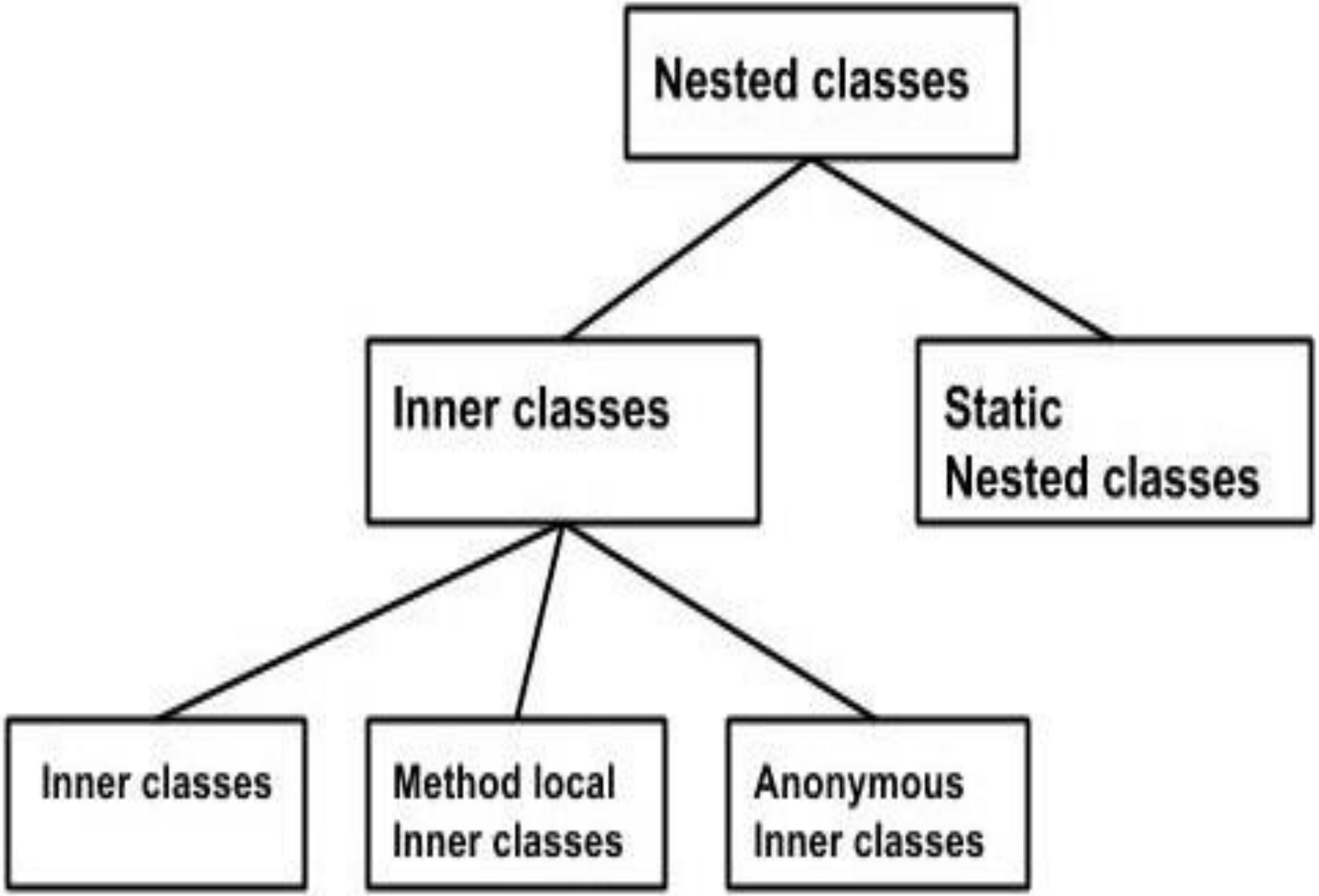
- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Local and Anonymous Inner Classes

- There are two additional types of inner classes. You can declare an inner class within the body of a method. Such a class is known as a *local inner class*. You can also declare an inner class within the body of a method without naming it. These classes are known as *anonymous inner classes*.

03. Class concepts – (Java)

Class fundamentals
Methods
Constructors
Inner Classes



Nested classes

Inner classes

Static
Nested classes

Inner classes

Method local
Inner classes

Anonymous
Inner classes

Nested Classes

- The Java programming language allows you to define a class within another class. Such a class is called a *nested class*.

```
class OuterClass {
        ...
        class NestedClass {
        ...
        }
}
```

Nested Classes

- Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called *static nested classes*. Non-static nested classes are called *inner classes*

Nested Classes

```
  class OuterClass {
        ...
        static class StaticNestedClass {
        ...
         }
        class InnerClass {
         ...
         }
}
```

Static Nested Classes

- As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class — it can use them only through an object reference.

Inner Classes

- As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Local and Anonymous Inner Classes

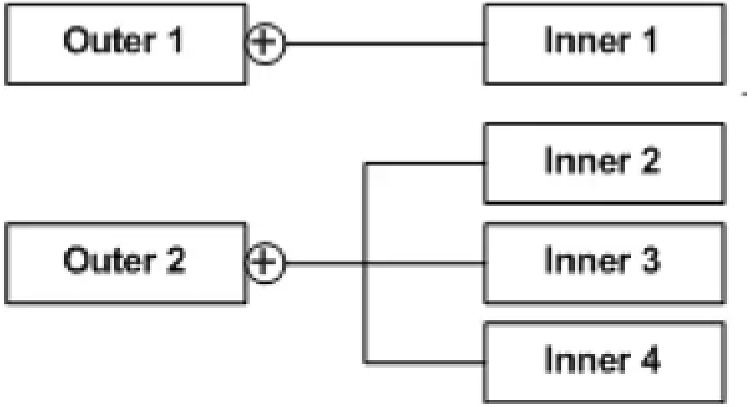
- There are two additional types of inner classes. You can declare an inner class within the body of a method. Such a class is known as a *local inner class*. You can also declare an inner class within the body of a method without naming it. These classes are known as *anonymous inner classes*.

Inner Classes

- Advantage of Inner Classes
  - Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class**, including private.
  - Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
  - **Code Optimization**: It requires less code to write.

Inner Classes

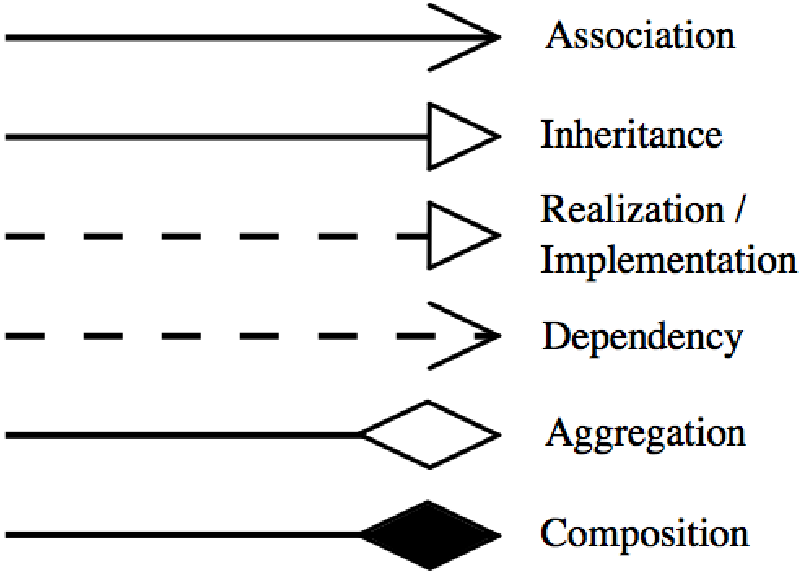
Nested class in UML (for any language) can be represented as:



Here

1. Class Inner1 is nested inside the outer class Outer 1
2. Classes Inner2, Inner3, Inner4 classes are nested inside Outer2

04. OO Relationships

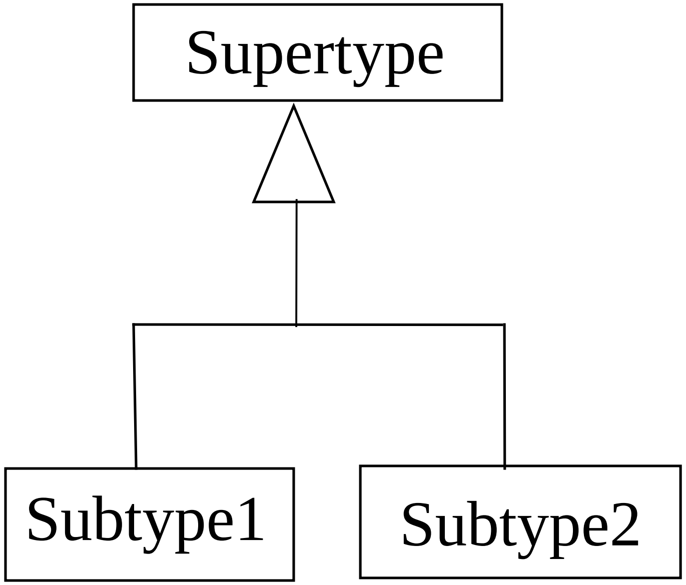


OO Relationships

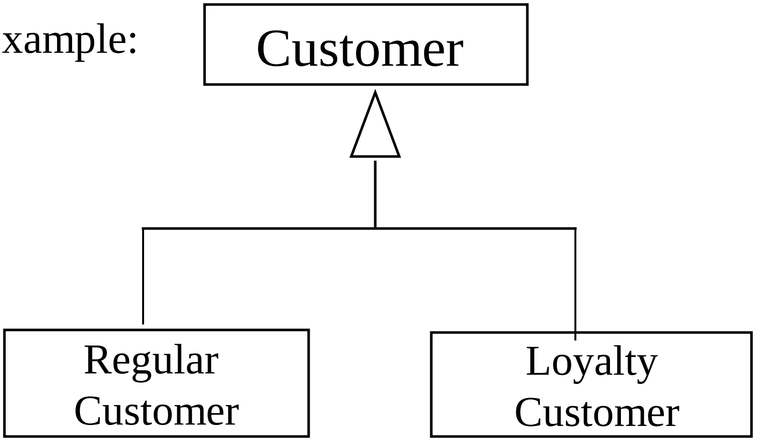
- There are three kinds of Relationships
  - Generalizations (parent-child relationship)
  - Associations (student enrolls in course)
  - Dependencies

- Associations can be further classified as
  - Aggregation
  - Composition

OO Relationships: **Generalization**



Example:



-Inheritance is a required feature of object orientation

-Generalization expresses a parent/child relationship among related classes.

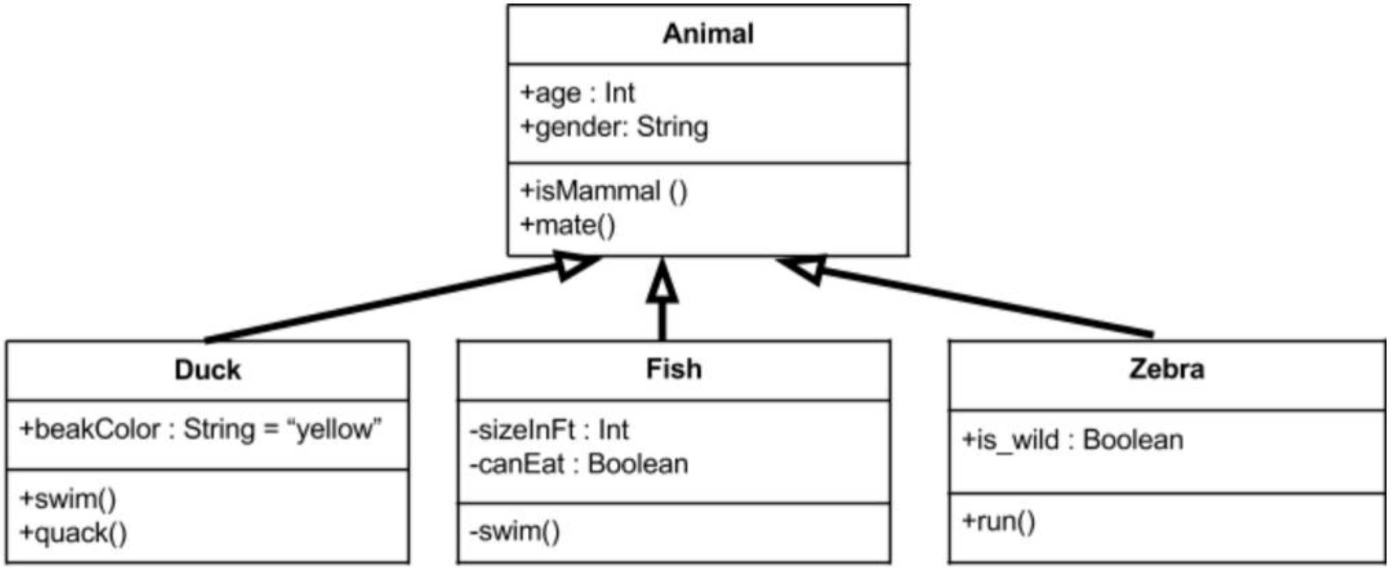
-Used for abstracting details in several layers

Inheritance

- Allows the creation of hierarchical classification.
- Using inheritance, we can create a general class that defines traits common to a set of related items.
- A class that is inherited is called a '*superclass*'.
- The class that does the inheriting is called a '*subclass*'.

Inheritance

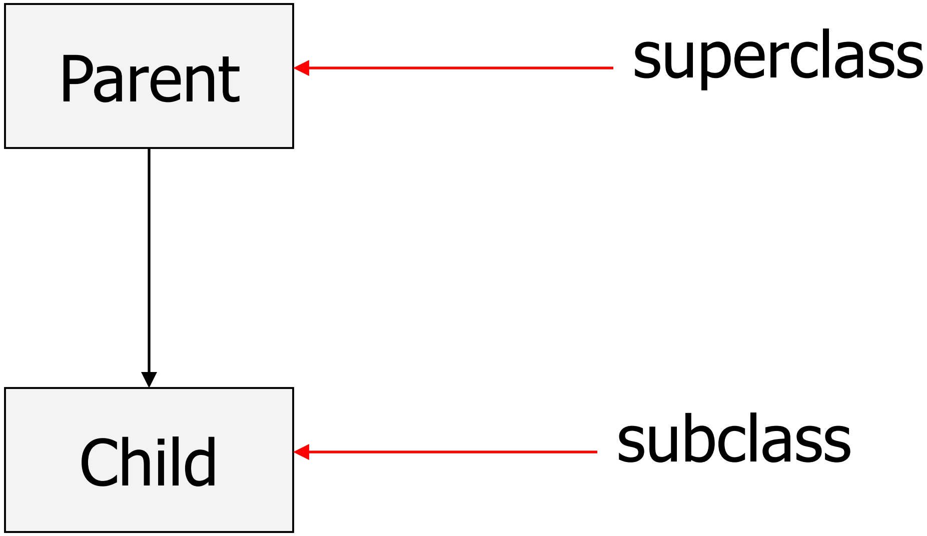
- Indicates that child (subclass) is considered to be a specialized form of the parent (super class).
- For example consider the following:



Inheritance

- Derive new classes from old classes
- Improve code re-use
- Easier to manage and understand complexity

Inheritance



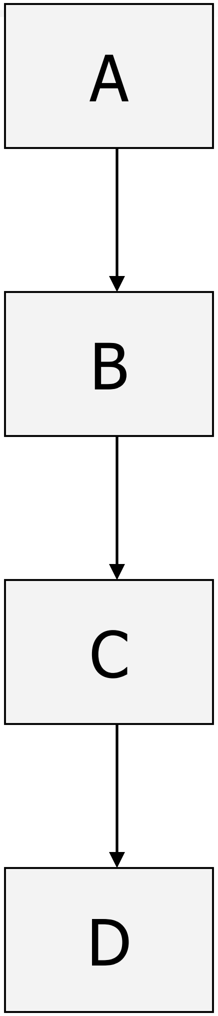
- Java supports multilevel inheritance but not multiple inheritance.

Java Implementation

```
public class Parent {
    . . .
}

public class Child extends Parent  {
    . . .
}
```

Multilevel Inheritance

```
class A {. . . }

class B extends A { . . . }
class C extends B { . . . }
class D extends C { . . . }
```



What is inherited?

- All *public* data members and methods (except constructors) in the superclass are *inherited* by the subclass. It is as if their definitions are copied into the subclass's class definition.
- No members of the subclass are visible to the superclass.

Inheritance Example

```
public class Person  {

    public String name;
    public int age;
    public Date birthDay;

    public Person(String name, int age, Date birthDay)  {
        this.name = name;
        this.age = age;
        this.birthDay = birthDay;
    }

    public void setName(String name)  {
        this.name = name;
    }
}
```

Example - con’t

```
public class Student extends Person  {
    public int studentID;
    public String dept;
    private float GPA;

    public Student(String name, int age, Date birthDay, int
                            studentID, String dept, float GPA)  {

        super(name, age, birthDay);
        …
        this.studentID = studentID;
        …
    }
}
```

super Keyword

- *super* is used to refer to the members of the current object’s superclass.
- Used for calling the superclass version of a method which the subclass has over-ridden.

```
public Roof getRoof()  {
        if ( convertibleRoofIsBroken )
                return super.getRoof();
        return new ConvertibleRoof();
}
```

super - con’t

- Can be used in a constructor to access a super-class constructor.
- Must be first line in constructor if present
- Syntax: super(<constructor args>);

```
public class Student extends Person  {
    …
    public Student(String name, int age, Date birthday, int studentID)  {
            super(name, age, birthday);
            this.studentID = studentID;
    }
}
```

Method Overriding

- Redefine methods inherited from superclass to add or change functionality.
- That is, When a method in the subclass has the same name and type signature as a method in its superclass.
- Method overriding allows Java to support run-time polymorphism.

Method Overriding: Example

```
class A  {
    int i, j;

    A(int a, int b)  {
          i = a;
          j = b;
    }

    void display()  {
          System.out.println(“i and
                    j: “ +i+ “ “ +j);
    }
}
```

```
class B extends A  {
    int k;

    B(int a, int b, int c)  {
          super(a, b);
          k = c;
    }

    void display()  {
          // super.display();
          System.out.println(“k: “+k);
    }
}
```

```
class Override  {
    public static void main(String
                              args[])  {
          B subOb = new B(1, 2, 3);
          subOb.display();
    }
}
```

Run-time polymorphism - Example

```
class Figure  {
    double dim1;
    double dim2;

    Figure(double a, double b)  {
        dim1 = a;
        dim2 = b;
    }

    double area()  {
        System.out.println(“Area
                          undefined”);
        return 0;
    }
}
```

```
class Rectangle extends Figure  {

    Rectangle(double a, double b)  {
        super(a, b);
    }

    double area()  {
        System.out.println(“Inside
                          rectangle”);
        return dim1*dim2;
    }
}
```

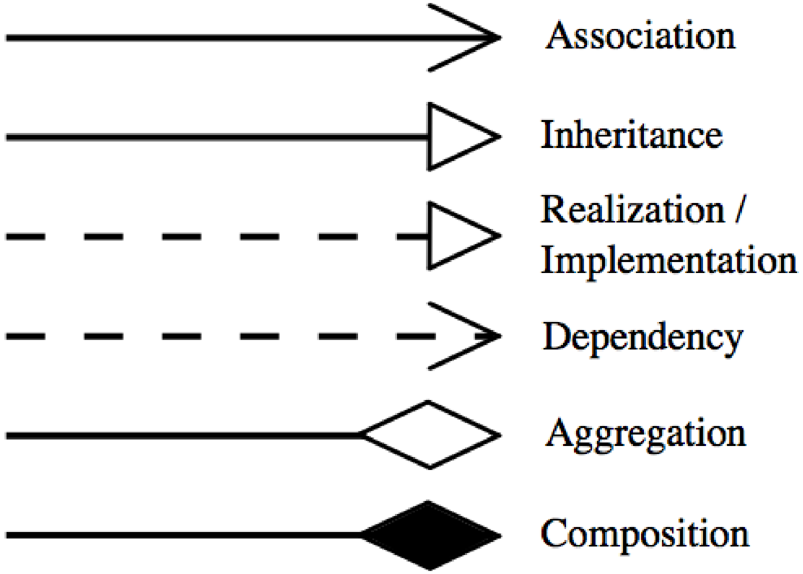
```
class Triangle extends Figure  {

    Triangle(double a, double b)  {
        super(a, b);
    }

    double area()  {
        System.out.println(“Inside triangle”);
        return dim1*dim2/2;
    }
}
```

```
class AreaFinder  {

    public static void main(String args[])  {

          Figure f = new Figure(10, 10);
          Rectangle r = new Rectangle(9, 5);
          Triangle t = new Triangle(10, 8);

          Figure figRef;

          figRef = r;
          System.out.println(“Area is “ + figRef.area());

          figRef = t;
          System.out.println(“Area is “ + figRef.area());

          figRef = f;
          System.out.println(“Area is “ + figRef.area());
    }
}
```

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how java implements run-time polymorphism.

04. OO Relationships



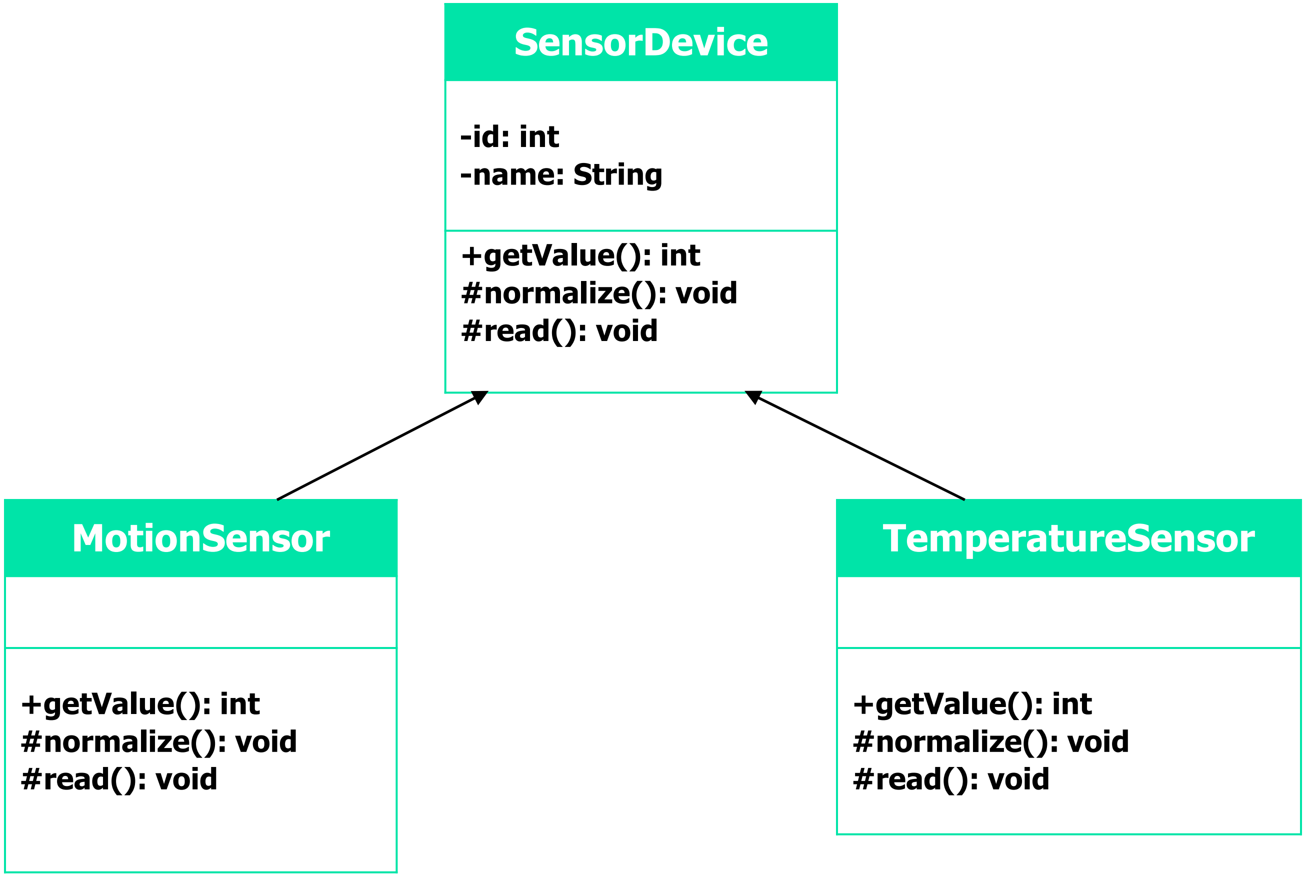
OO Relationships

- Associations
  - Indicate that instances of one model element are connected to instances of another model element
- Generalizations
  - Indicate that one model element is a specialization of another model element

OO Relationships

- Realizations
  - Indicate that one model element provides a specification that another model element implements
- Dependencies
  - Indicate that a change to one model element can affect another model element

Inheritance



Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how java implements run-time polymorphism.

protected access modifier

- The *protected* access modifier is used for fields or methods and cannot be used for classes and Interfaces.
- It also cannot be used for fields and methods within an interface.
- Fields, methods and constructors declared protected in a superclass can be accessed only by its subclasses.
- Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

abstract modifier

- We declare a class *abstract* when we want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

- That is, when a superclass is unable to create a meaningful implementation for a method.

- The *abstract* modifier can be applied to classes and methods.

Abstract Class

- An abstract class cannot be instantiated.
- Abstract classes provide a way to defer implementation to subclasses.
- Declaration:

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

Abstract Method

- No implementation for a method. Only the signature of the method is declared.
- Used to put some kind of compulsion on the person who inherits from this class. i.e., the person who inherits **MUST** provide the implementation of the method to create an object.
- A method can be made abstract to defer the implementation. i.e., when you design the class, you know that there should be a method, but you don't know the algorithm of that method.

Abstract Method

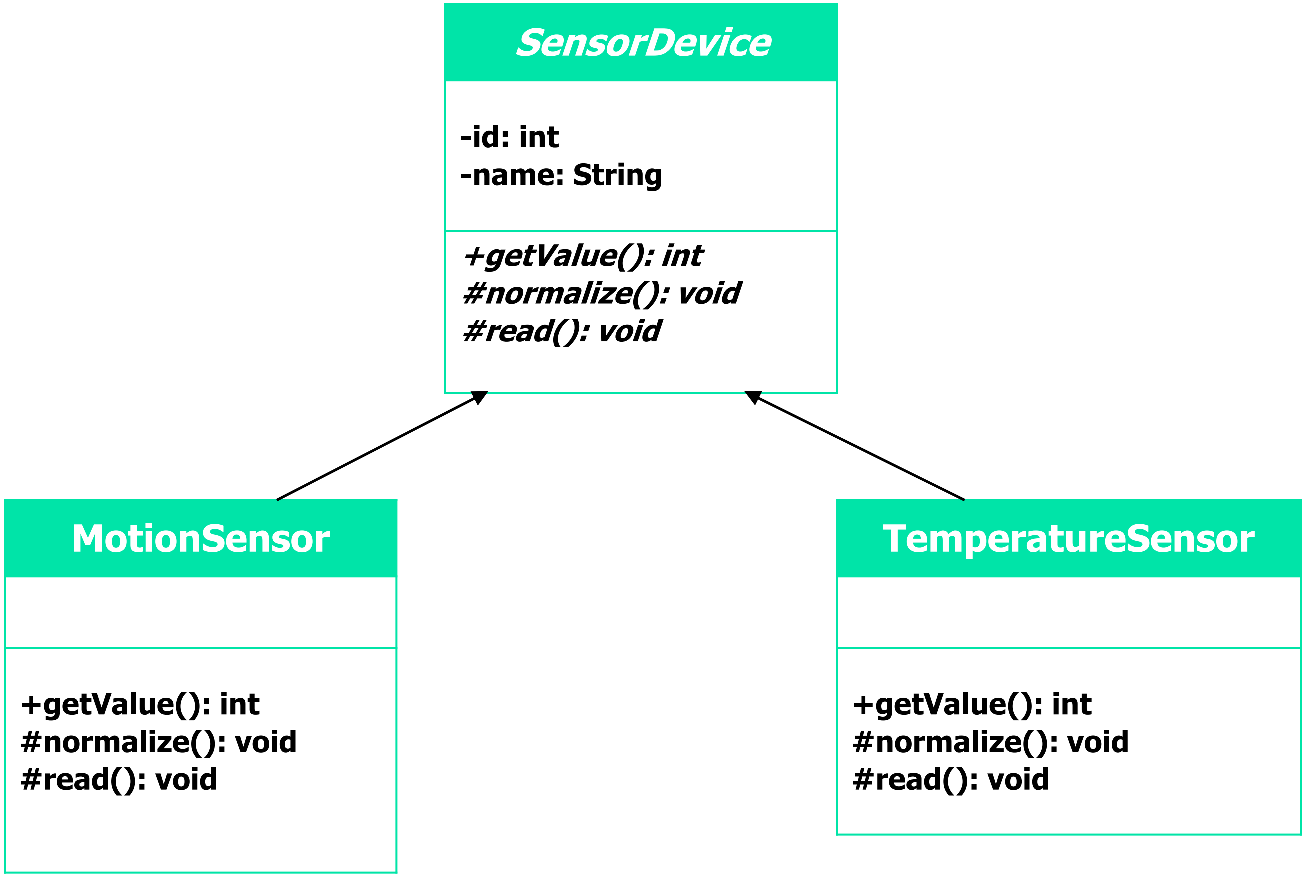
- Declaration:

```
abstract void myMethod();
```

abstract modifier

- A class **must** be declared *abstract* if any of the following conditions is true:
  - The class has one or more abstract methods.
  - The class inherits one or more abstract methods (from an abstract parent) for which it does not provide implementations.

Inheritance



SensorDevice

-id: int
-name: String

+getValue(): int
#normalize(): void
#read(): void

MotionSensor

+getValue(): int
#normalize(): void
#read(): void

TemperatureSensor

+getValue(): int
#normalize(): void
#read(): void

Example – Abstract class

```
Abstract class Shape  {
    double dim1;
    double dim2;

    Shape(double a, double b)  {
        dim1 = a;
        dim2 = b;
    }

    abstract double area();
}
```

```
class Rectangle extends Shape  {

    Rectangle(double a, double b)  {
        super(a, b);
    }

    double area()  {
        System.out.println("Inside
                         rectangle");
        return dim1*dim2;
    }
}
```

```
class Triangle extends Shape  {

    Triangle(double a, double b)  {
            super(a, b);
    }

    double area()  {
            System.out.println(“Inside triangle”);
            return dim1*dim2/2;
    }
}
```

```
class AreaFinder  {

    public static void main(String args[])  {

        // Shape f = new Shape(10, 10);  // illegal now.
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);

        Shape ref;

        ref = r;
        System.out.println(“Area is “ + ref.area());

        ref = t;
        System.out.println(“Area is “ + ref.area());
    }
}
```

final modifier

- The *final* modifier can be applied to variables, methods, and classes.

final variables

- A variable can be declared as *final*.
- Doing so prevents its contents from being modified.
- We must initialize a *final* variable when it is declared. (***final*** ≈ ***const*** in C / C++ / C#)

final variables

Example:

```
final int FILE_NEW = 1;
final double PI = 3.142857;
```

- It is common coding convention to use all uppercase letters for final variables.
- Variables declared as final do not occupy memory on a per-instance basis.

final methods

- Methods declared as final cannot be overridden.

```
class A  {
    final void myMethod() {
          System.out.println(“This is a final method”);
    }
}

class B extends A  {
    void myMethod()  {  // ERROR! Cannot Override.
          System.out.println(“Illegal”);
    }
}
```

final classes

- Used to prevent a class from being inherited.
- Declaring a class final implicitly declares all of its methods as final too.
- It is illegal to declare a class as both abstract and final.

final classes

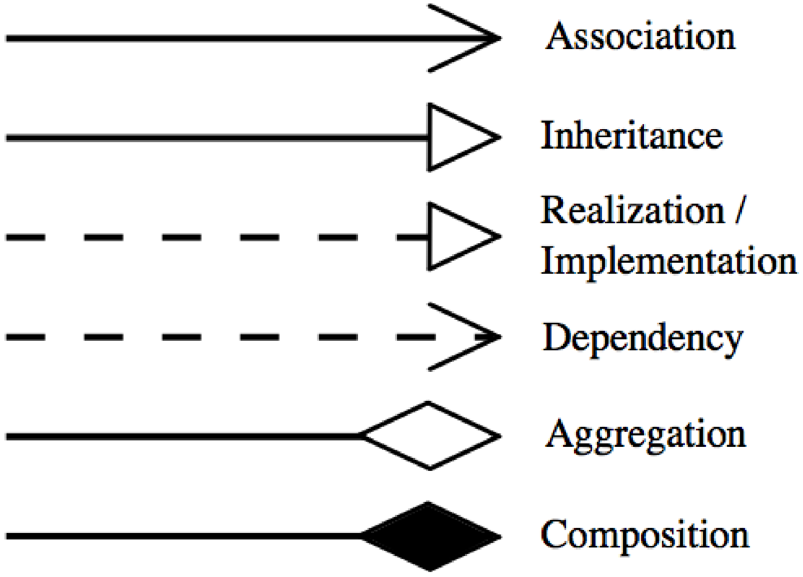
- Example:

```
final class A  {
      . . .
}

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

Summary:
Classes and Abstract Classes

- Classes
  - All declared methods must be defined.
  - No restriction on member variables.
- Abstract Classes
  - Some methods may be defined.
  - No restriction on member variables.

04. OO Relationships



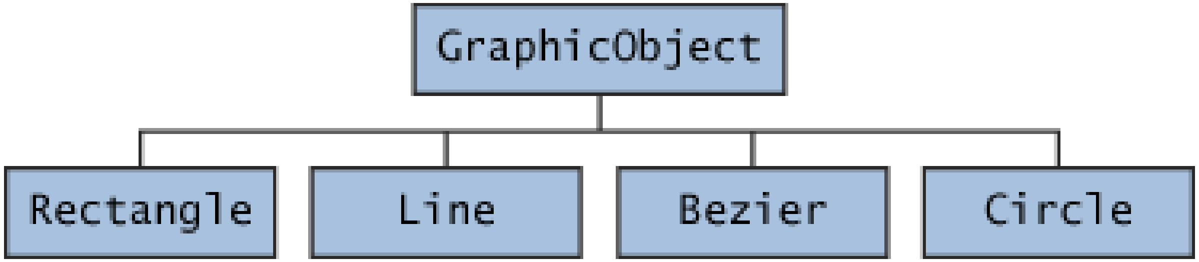
Summary: Inheritance

- Except for the Object class, a class has exactly one direct superclass. A class inherits fields and methods from all its superclasses, whether direct or indirect.

- An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods—methods that are declared but not implemented. Subclasses then provide the implementations for the abstract methods.

Summary: Inheritance

- You can prevent a class from being subclassed by using the final keyword in the class's declaration. Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.



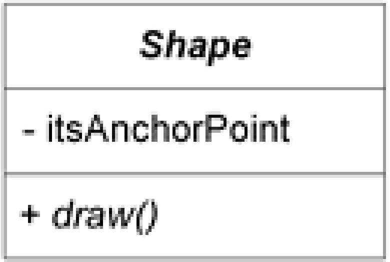
```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int
      newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

```
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

abstract class

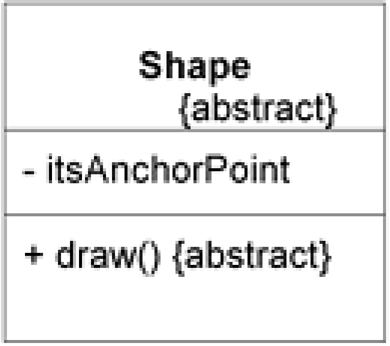
In UML there are two ways to denote that a class or a method is abstract. You can write the name in italics, or you can use the {abstract} property.



Shape

- itsAnchorPoint

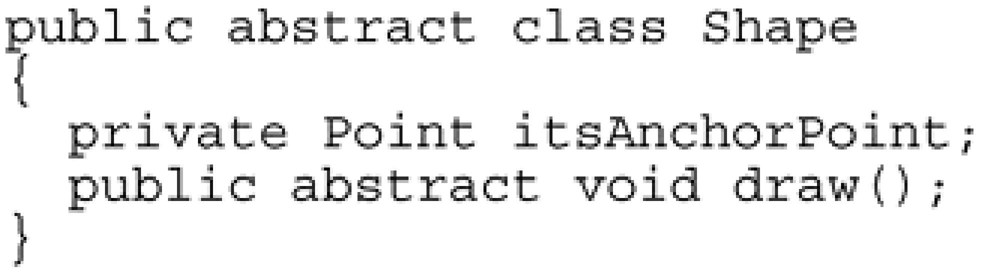
+ *draw()*



Shape
{abstract}

- itsAnchorPoint

+ draw() {abstract}



```
public abstract class Shape
{
  private Point itsAnchorPoint;
  public abstract void draw();
}
```

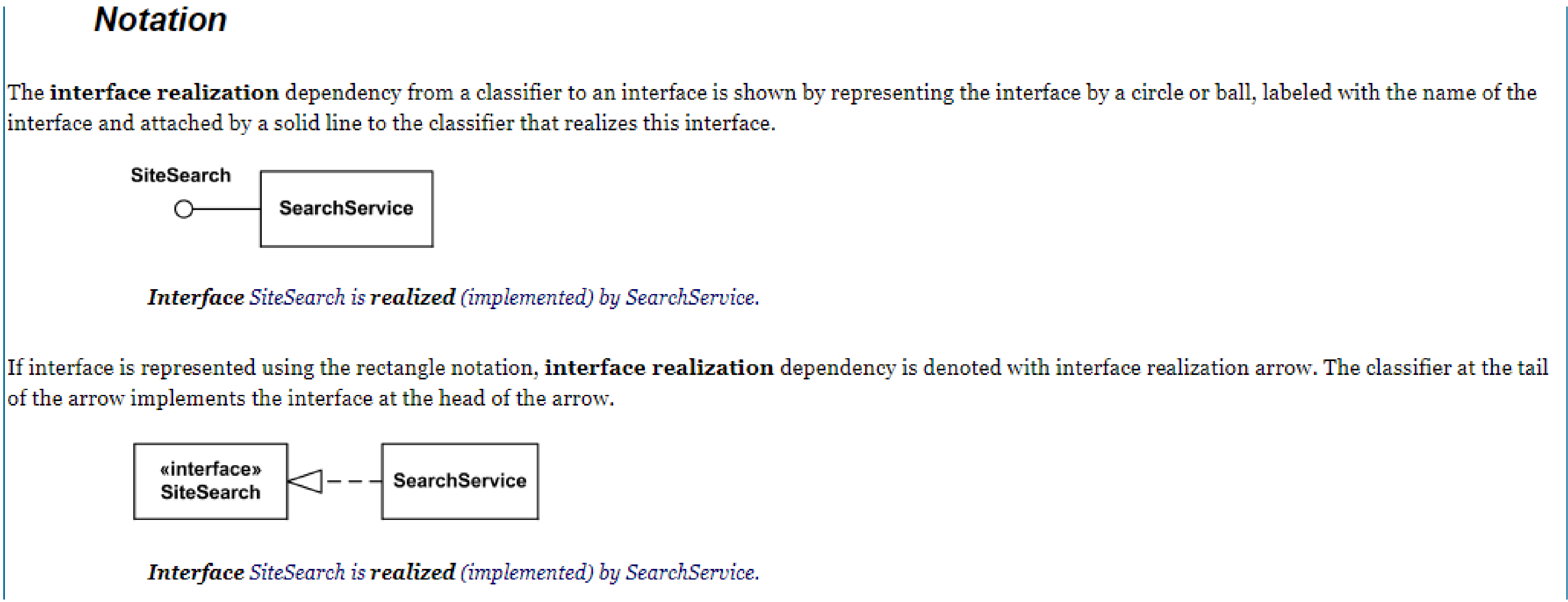
Realization/Implementation

- Realization is a relationship between two elements in a UML diagram where one element specifies behavior and the other element implements or executes, in other words, realizes, that behavior.
- There is a source element, called the realization element, and a target element, called the specification element, and the relationship is also often referred to as being between a supplier and client.

Realization/Implementation

- In many cases, the specification element will be an interface, or a collection of operations, with the realization element as the implementation of those behaviors or operations.

Realization/Implementation



Notation

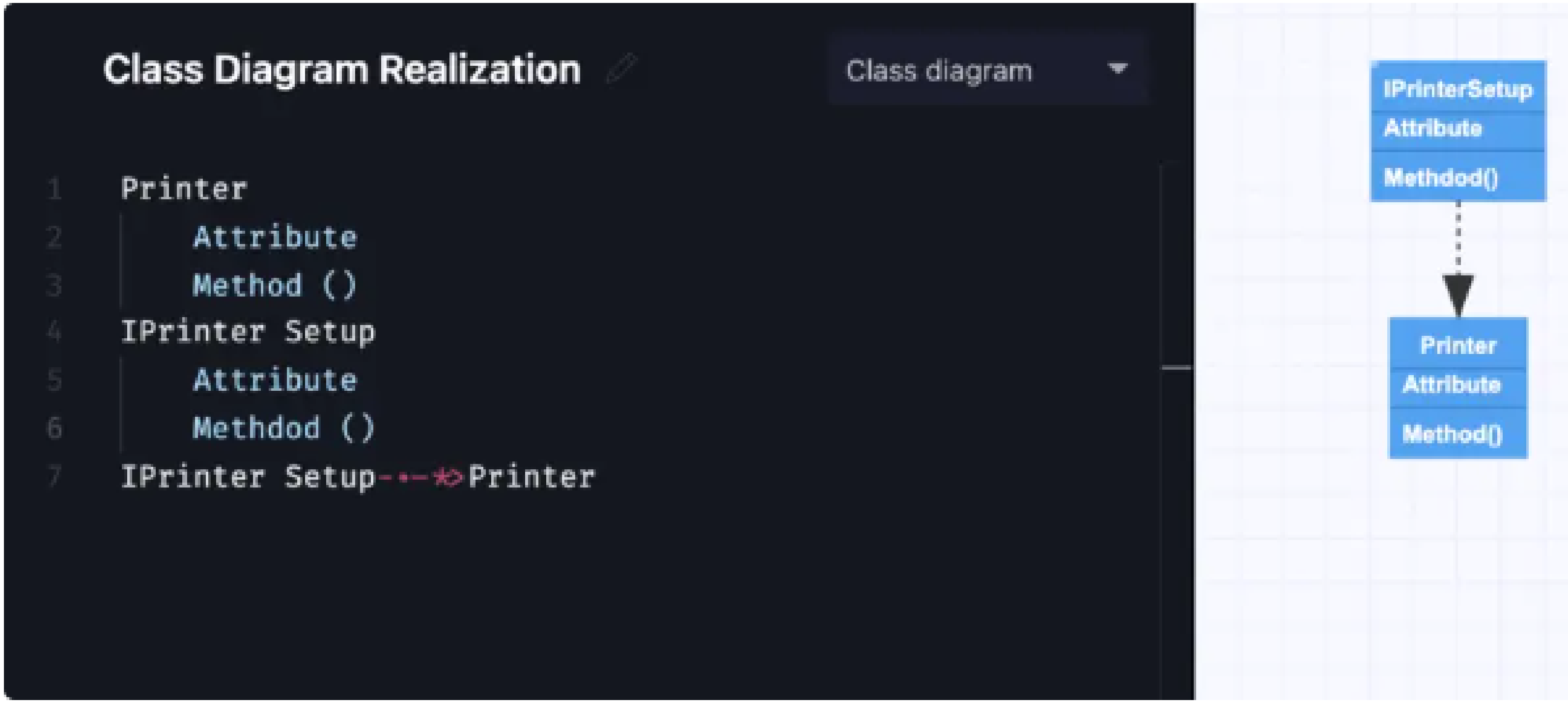
The **interface realization** dependency from a classifier to an interface is shown by representing the interface by a circle or ball, labeled with the name of the interface and attached by a solid line to the classifier that realizes this interface.

***Interface** SiteSearch is **realized** (implemented) by SearchService.*

If interface is represented using the rectangle notation, **interface realization** dependency is denoted with interface realization arrow. The classifier at the tail of the arrow implements the interface at the head of the arrow.

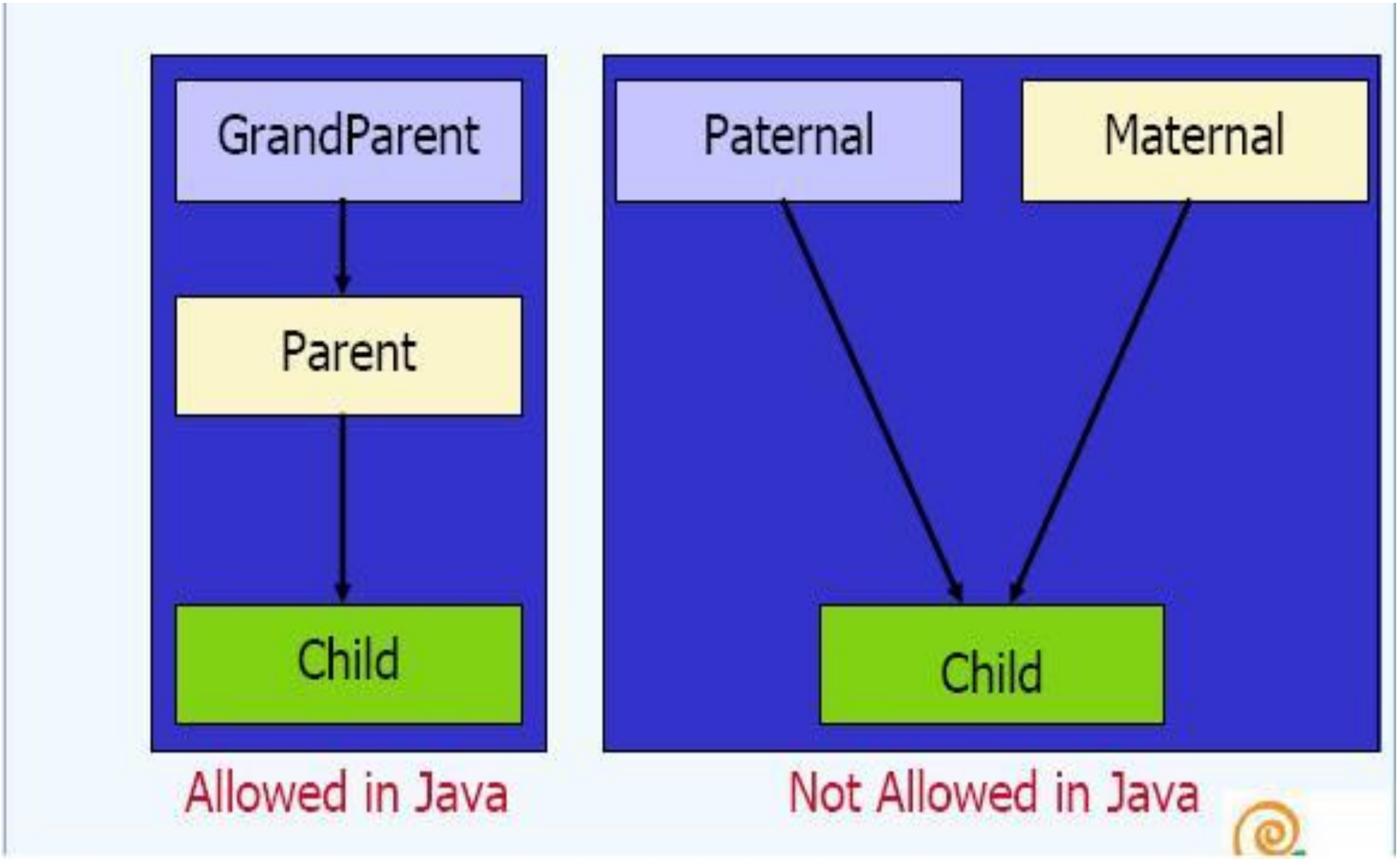
***Interface** SiteSearch is **realized** (implemented) by SearchService.*

Realization/Implementation



```
Printer
    Attribute
    Method ()
IPrinter Setup
    Attribute
    Methdod ()
IPrinter Setup-.-*>Printer
```

Interfaces in Java



Interfaces in Java

- An ***interface*** is a set of predefined methods to be implemented by one or more classes in future.
- An Interface will just give what the method should do, but it will not give the implementation for it.
- This helps the programmer to write his own logic in his class, which implements the particular interface.
- The methods in an interface will have no body and it just mention the method signature.

Interface

- A completely abstract class
- Only constants and abstract methods are allowed
- The *'class'* keyword is replaced by *'interface'*

```
public interface Drawable  {
            void draw();
            double getArea();
}
```

Implementing an Interface

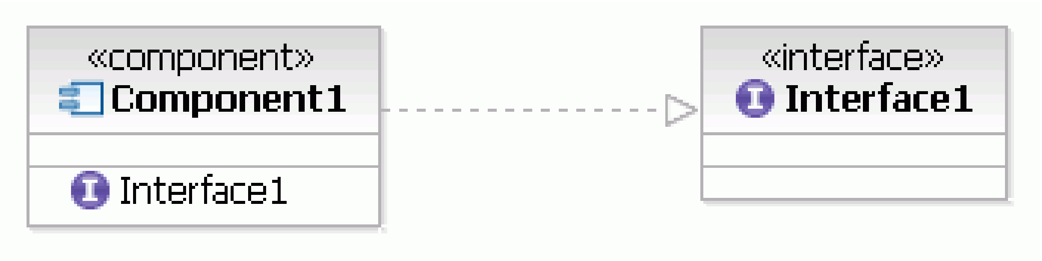
- The general form of implementing an interface is:

```
class classname [extends superclass] implements
                               interface1 [, interface2, . . .]  {
          // body……
}
```

Interface - cont’d

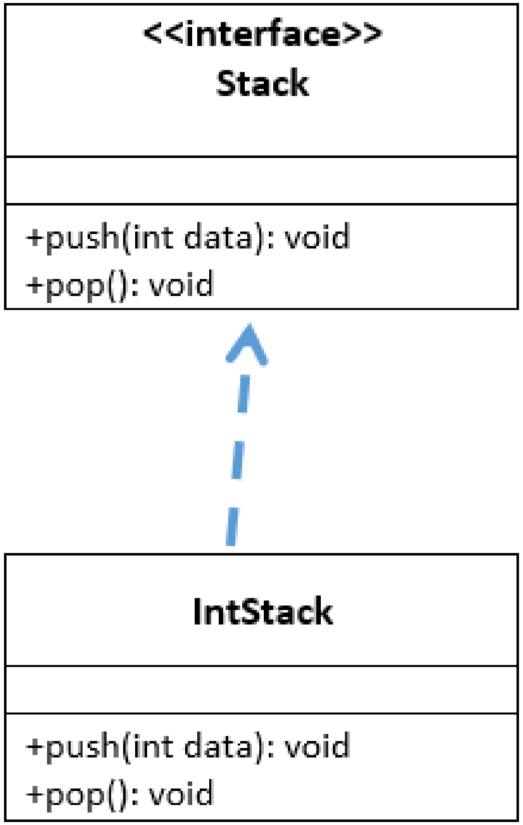
- All members are implicitly public; no need to supply access modifiers (and supplying any other than public is an error).
- Cannot be instantiated.
- Other classes may *‘implement’* an interface.

Interface



- An interface realization relationship is displayed in the diagram editor as a dashed line with a hollow arrowhead. The interface realization points from the classifier to the provided interface.

Exercise



<<interface>>
Stack

+push(int data): void
+pop(): void

IntStack

+push(int data): void
+pop(): void

Example:

```
public class Circle extends Shape implements Drawable  {
     public void draw()  { … }
     public double getArea()  {
               return radius*radius*Math.PI;
     }
     public display()  { . . . }
}
```

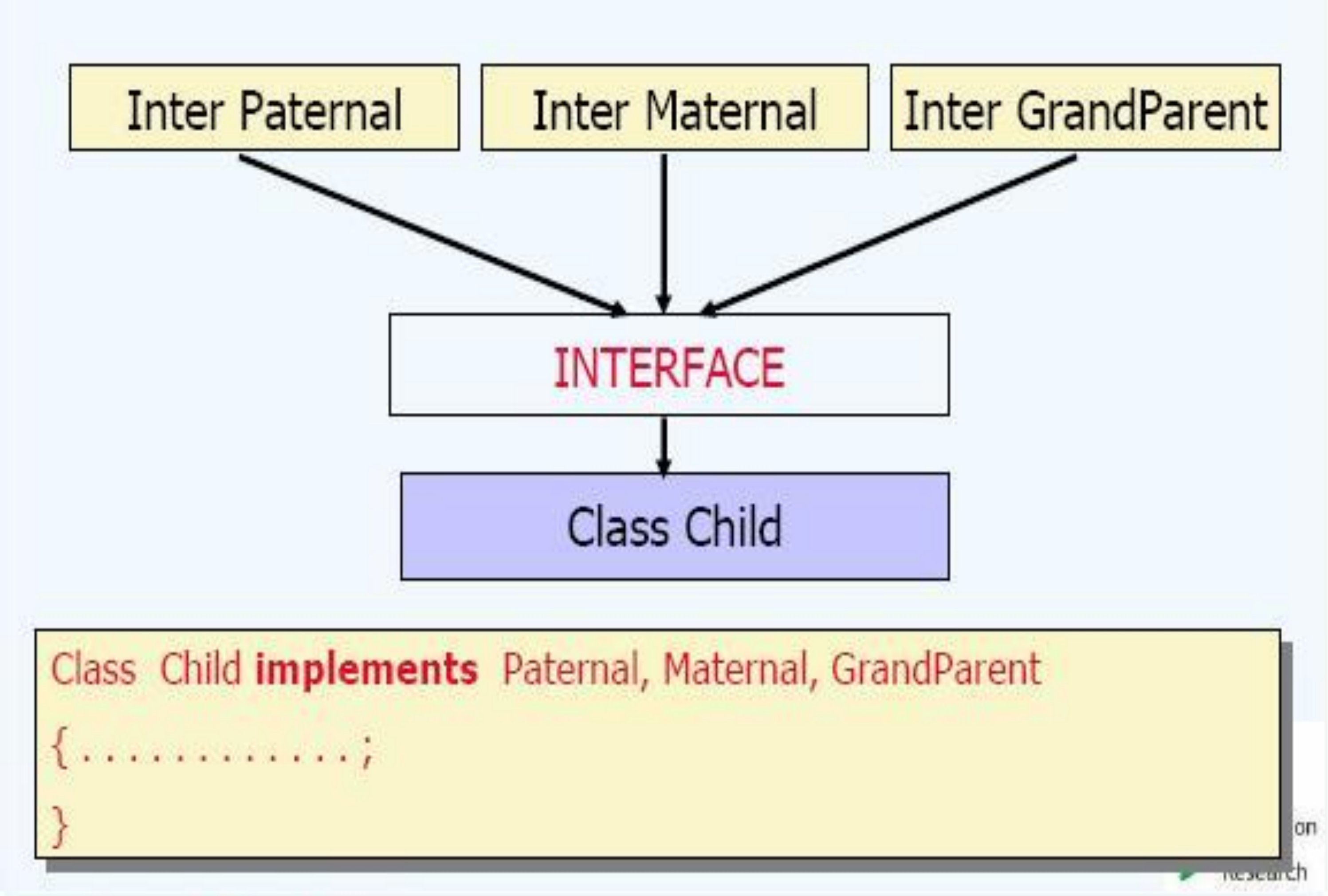
```
abstract class GraphicObject {
   int x, y;
   ...
   void moveTo(int newX, int
     newY) {
      ...
   }
   abstract void draw();
   abstract void resize();
}
```

```
interface GraphicObject {

     void draw();
     void resize();
}
```

```
class Circle extends GraphicObject {
   void draw() {
      ...
   }
   void resize() {
      ...
   }
}
```

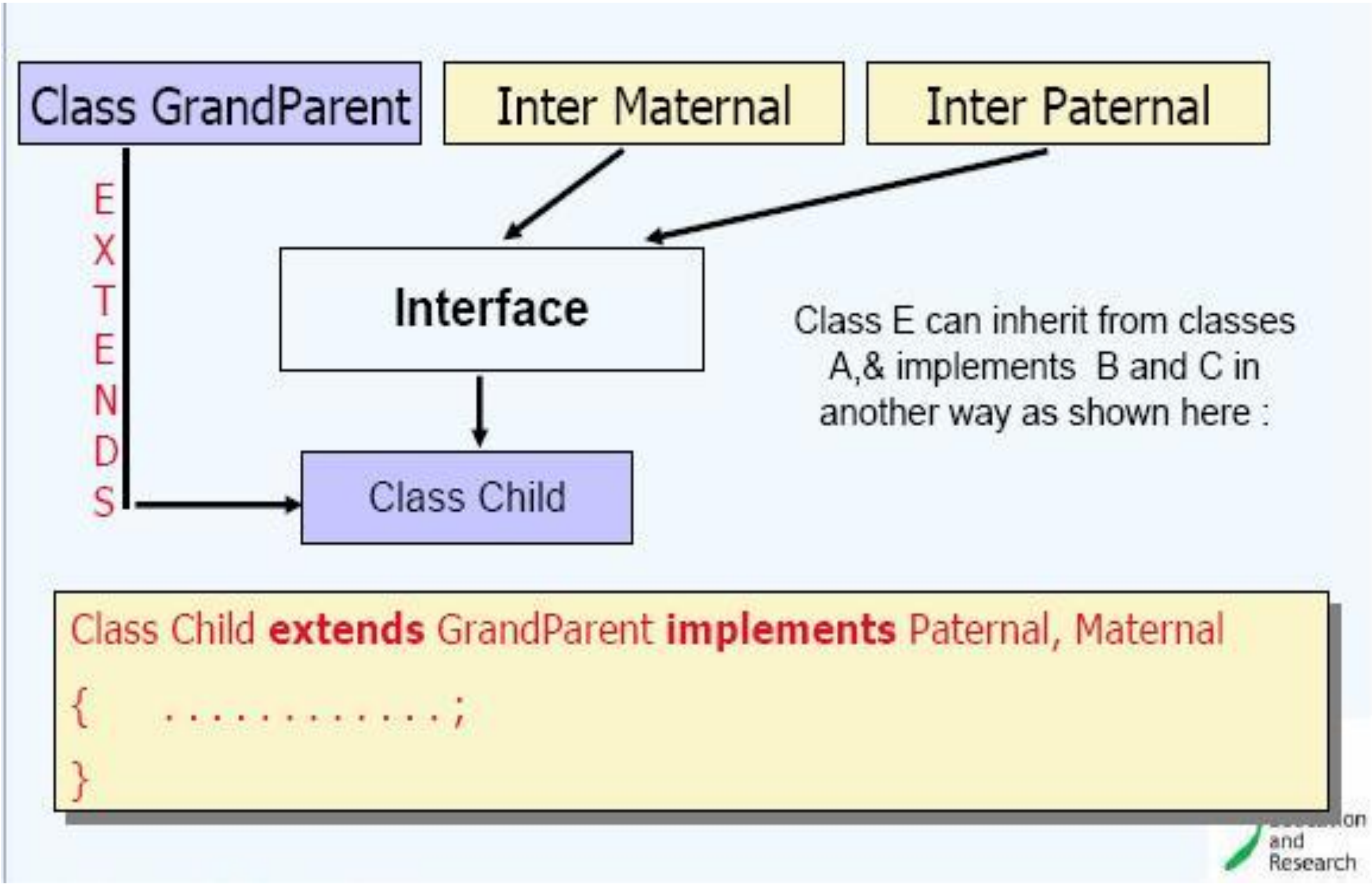
```
class Circle implements GraphicObject {
   void draw() {
      ...
   }
   void resize() {
      ...
   }
}
```

Implementing Multiple Inheritance in Java



```
Class  Child implements  Paternal, Maternal, GrandParent
{.............;
}
```

Implementing Multiple Inheritance in Java



Class E can inherit from classes A,& implements B and C in another way as shown here :

```
Class Child extends GrandParent implements Paternal, Maternal
{    .............;
}
```

Abstract Class v/s Interfaces

- Specifies the full set of methods for an object.
- Implements none, some or all of its methods.
- Useless without being subclassed.

- Specifies a subset of methods for an object.
- Implements none of its methods.
- Useless without being implemented.

Both Abstract classes and Interfaces cannot be instantiated.

Summary:
Classes, Abstract Classes, and Interfaces

- Classes
  - All declared methods must be defined.
  - No restriction on member variables.
- Abstract Classes
  - Some methods may be defined.
  - No restriction on member variables.
- Interfaces
  - Methods are only declared.
  - Member variables must be static and final.

Polymorphism Example

```
public class Animal  {
      public void sound() {  }
}

public class Dog extends Animal  {
      public void sound()  {
            System.out.println(“Woof !!”);
      }
}
 public class Duck extends Animal  {
       public void sound()  {
             System.out.println(“Quack!!”);
       }
 }
```

Polymorphism Example cont’d

```
public class TryPolymorphism  {
    public static void main(String args[])  {
        // theAnimals is an array of superclass references
        Animal[] theAnimals= { new Dog(), new Duck() };
        Animal petChoice = theAnimals[0];
        petChoice.sound();                      // calls Dog’s method
        petChoice = theAnimals[1];
        petChoice.sound();                      // calls Duck’s method
    }
}
```

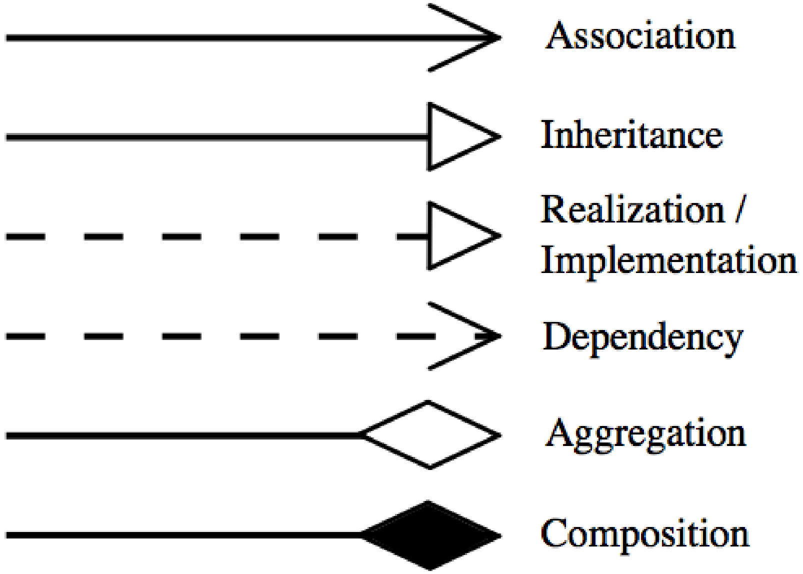
Polymorphism via Interfaces

- An object of a class implementing an interface may be treated as an object of type corresponding to the interface.
- Different classes support the same set of operations by implementing the same interface.
- At runtime, any of the implementing classes can be used

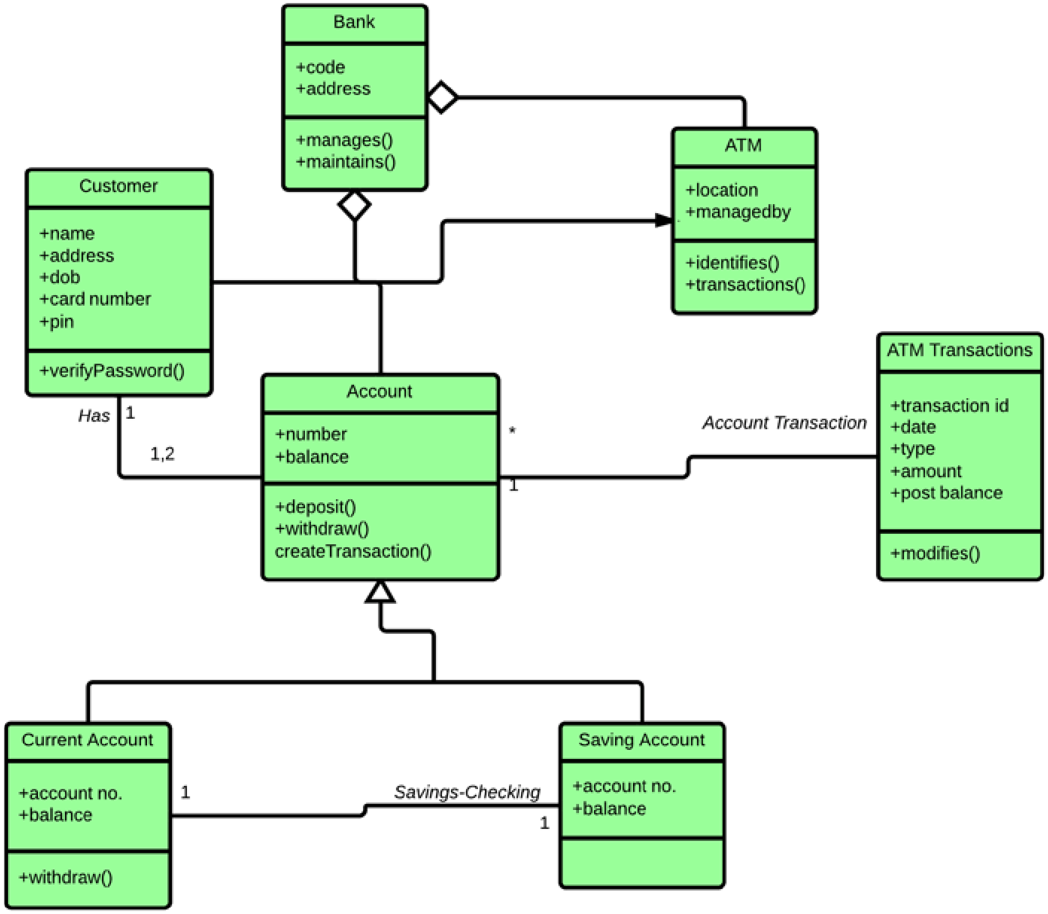
Example:

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

public class Circle implements Shape  {
       public double getArea()  { return PI * r * r ; } }

public class Rectangle implements Shape {
       public double getArea()  { return height * width ; } }

public class Triangle implements Shape {
       public double getArea()  { return  0.5 * base * height ; } }
public class ShapeAreaFinder   {

    public static void main(String[] args)  {
        Shape [] theShapes = { new Circle(1, 2, 3),  new
                          Rectangle(10, 23),  new Triangle(12, 20) };

        System.out.println(theShapes[0].getArea());
        System.out.println(theShapes[1].getArea());
        System.out.println(theShapes[2].getArea());
    }
}
```

04. OO Relationships



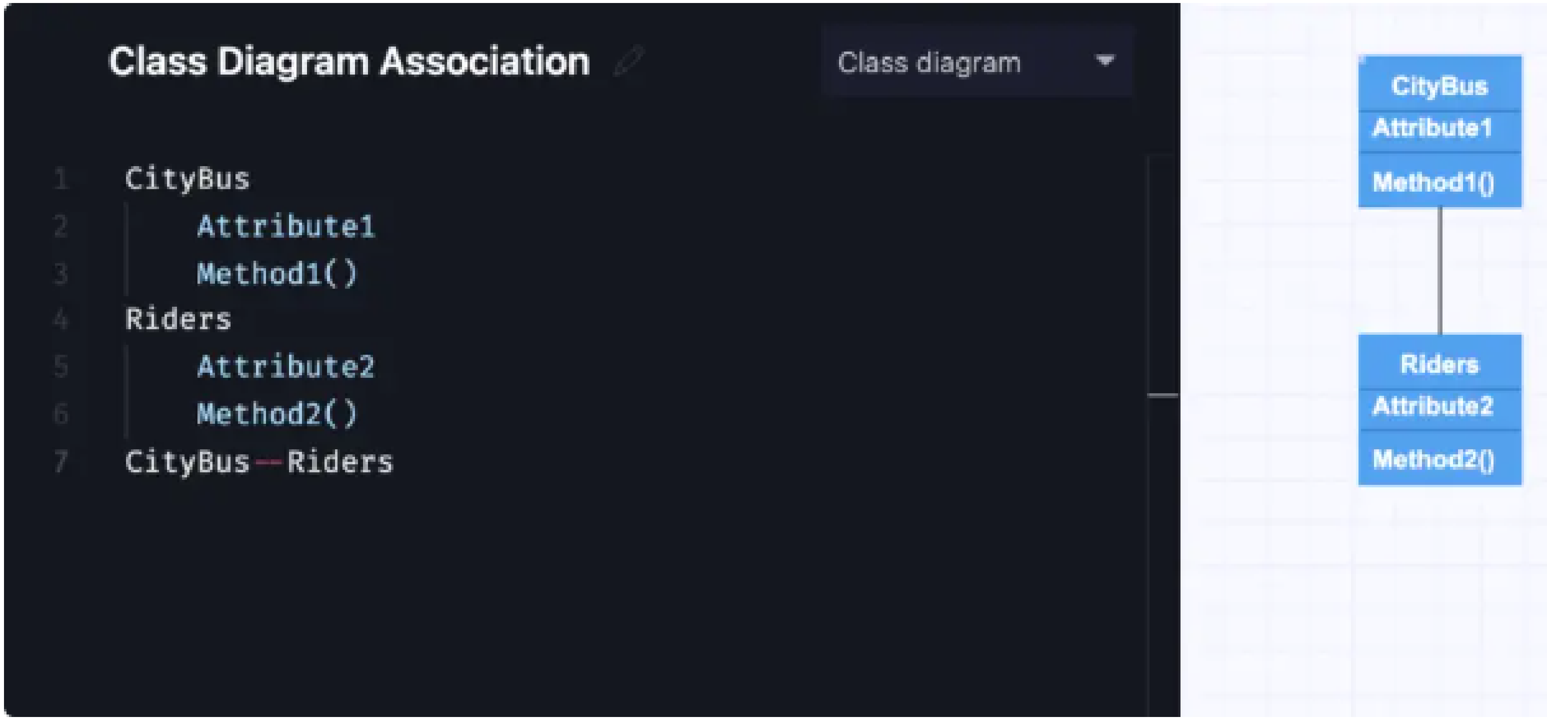
OO Relationship



Association

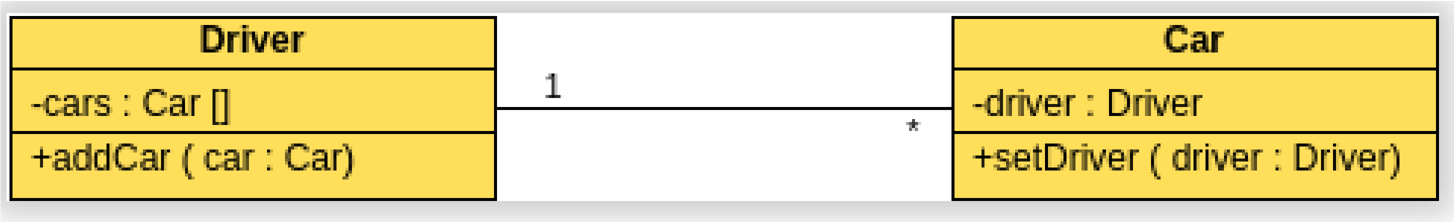
- This simply means that one model element is linked in some way to another model element. The association indicates the nature and rules that govern the relationship. The basic way to represent association is with a line between the elements.

Association



```
CityBus
    Attribute1
    Method1()
Riders
    Attribute2
    Method2()
CityBus--Riders
```

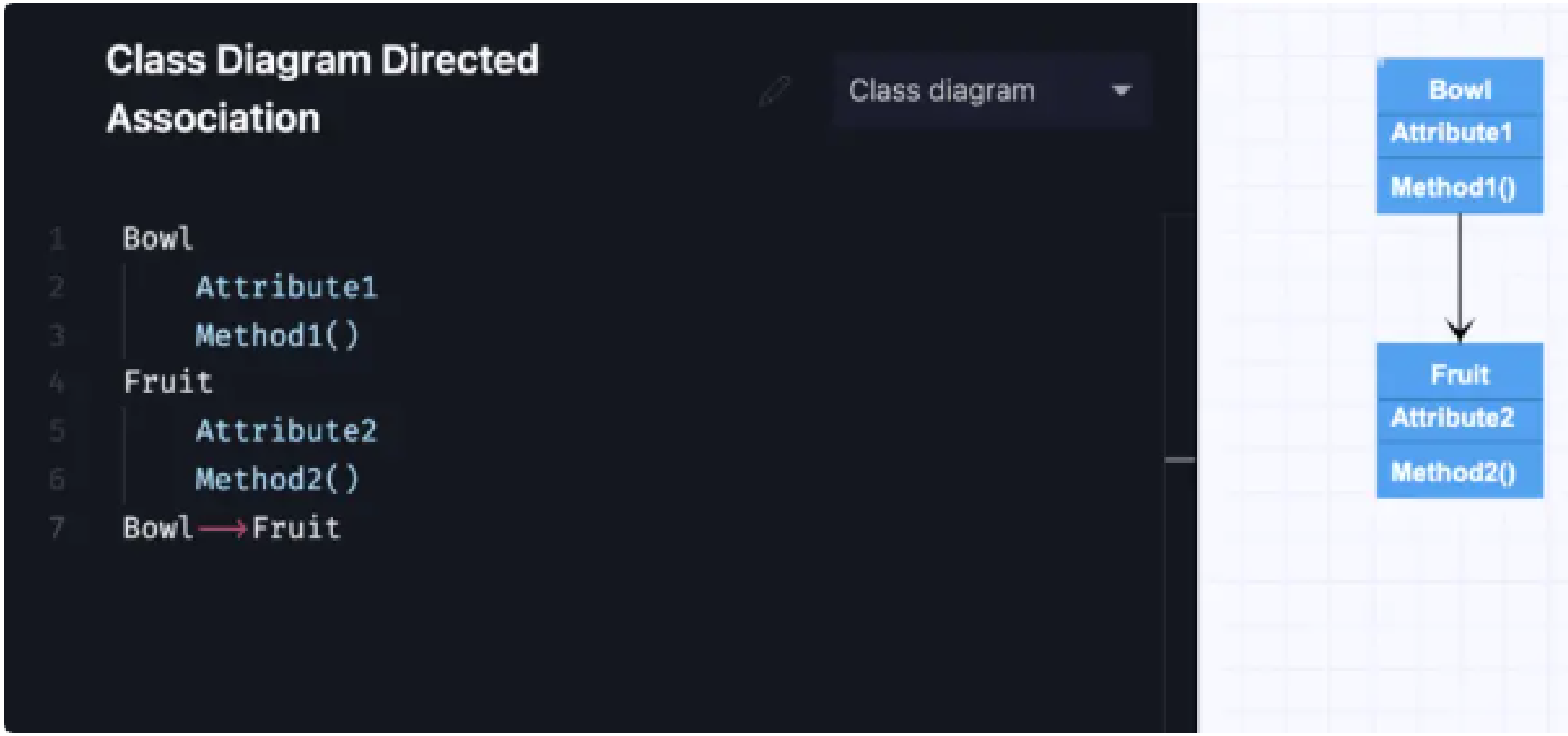
Association



Association

- Association can be more complex, in that it can be directed, which is represented by an arrow showing the flow of control, or even reflexive, in cases where the element has a relationship to itself. In this case, the arrow loops back to the element.

Association



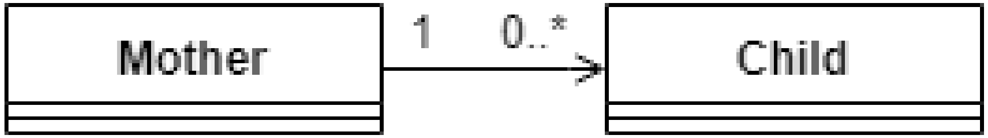
Class Diagram Directed Association

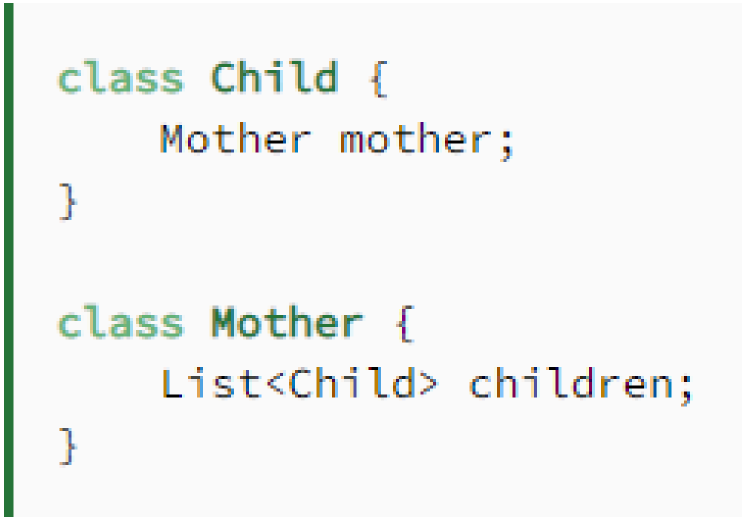
```
Bowl
    Attribute1
    Method1()
Fruit
    Attribute2
    Method2()
Bowl⟶Fruit
```

Association (Multiplicity)

- An association relationship between elements can also have cardinality, for instance, one-to-one, one-to-many, many-to-one, or many-to-many, zero-to-many, and so on. This can also be shown in a label on the line..

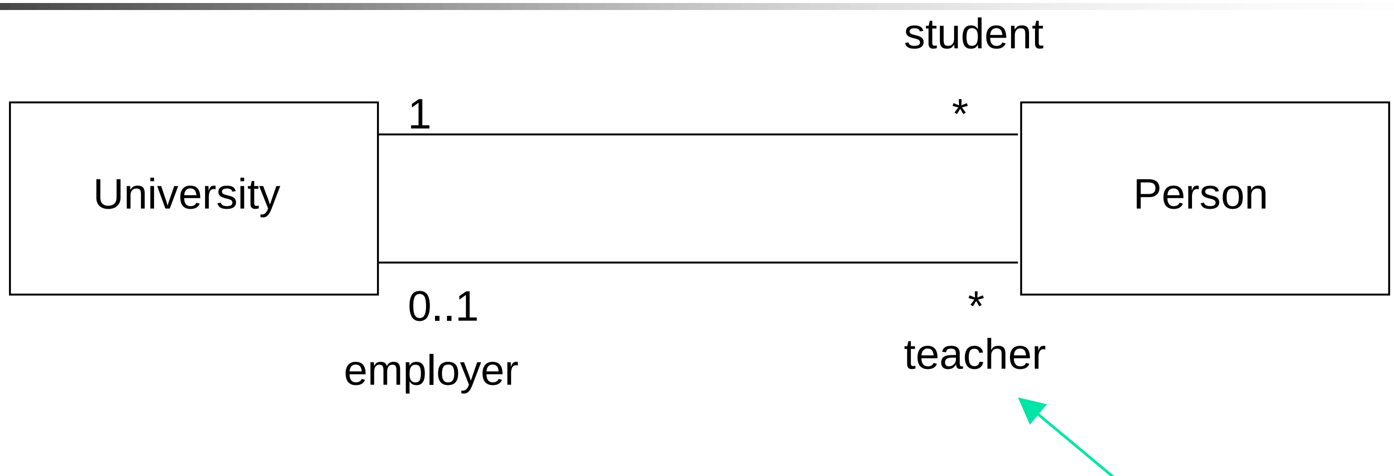
Association





```
class Child {
    Mother mother;
}

class Mother {
    List<Child> children;
}
```

Association: Multiplicity and Roles



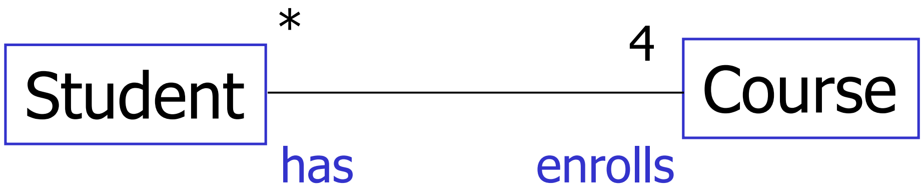
Role

Multiplicity	
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

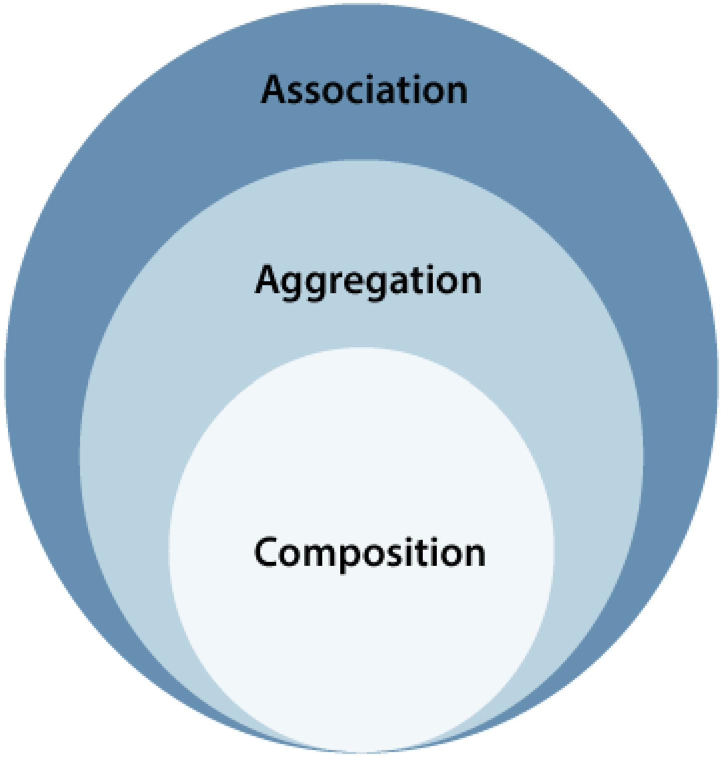
"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

Association: Model to Implementation



```
Class Student {
    Course enrolls[4];
}

Class Course {
    Student have[];
}
```

Association

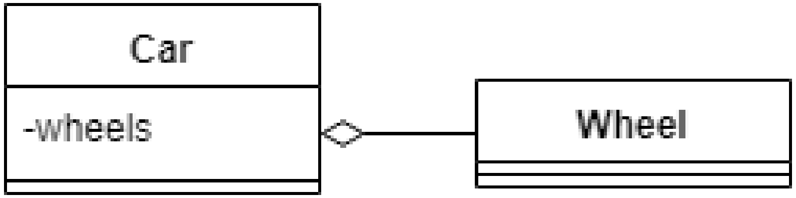


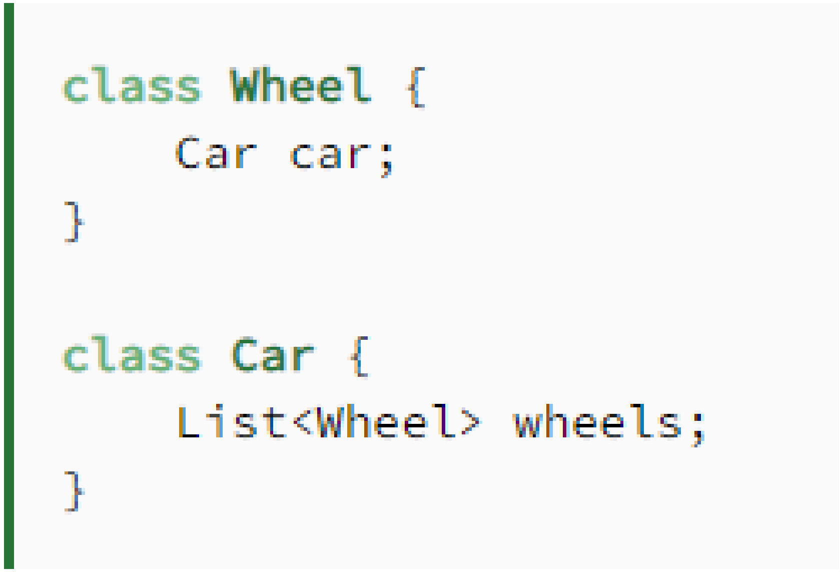
Aggregation

- This type of association relationship indicates an element is formed by a collection of other elements. For instance, a company has departments or a library has books.

- The aggregate element relies on other elements as parts, but those other elements can also exist independently of it.

Aggregation



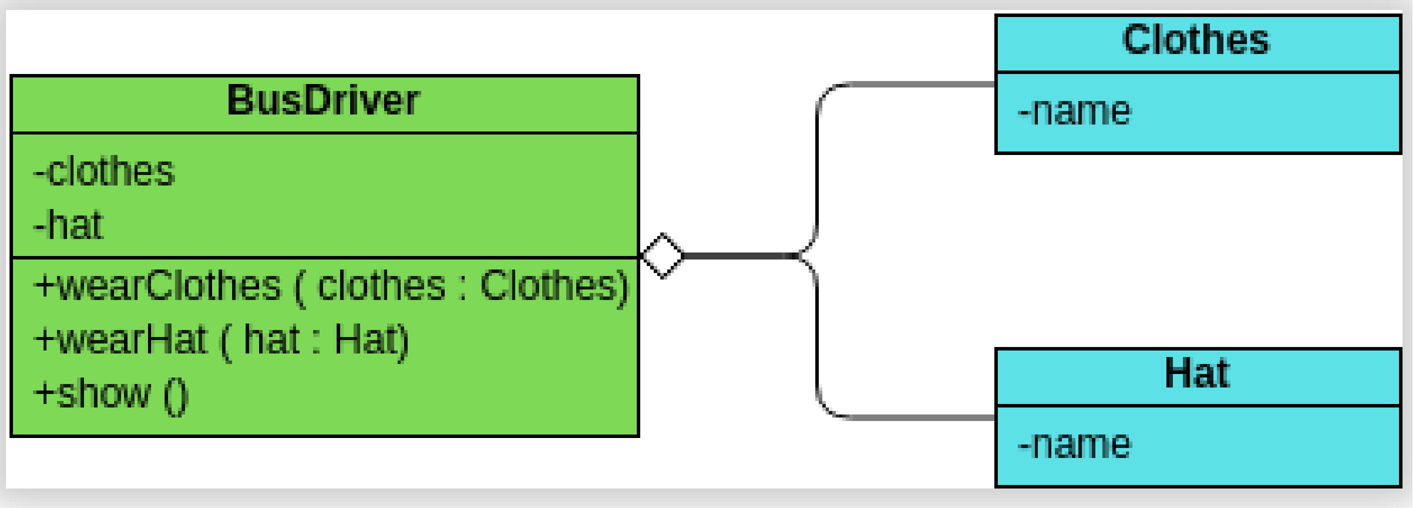


```
class Wheel {
    Car car;
}

class Car {
    List<Wheel> wheels;
}
```

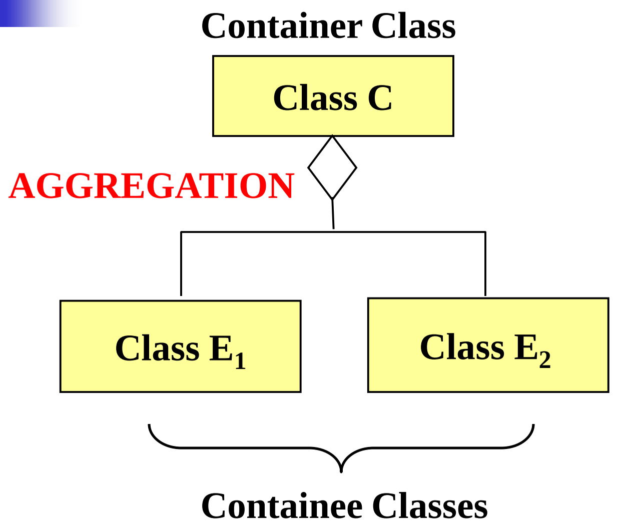
Aggregation

- An aggregation is represented by a line from one class to another, with an unfilled diamond shape near the aggregate, or the element that represents the class that is assembled by combining the part elements.

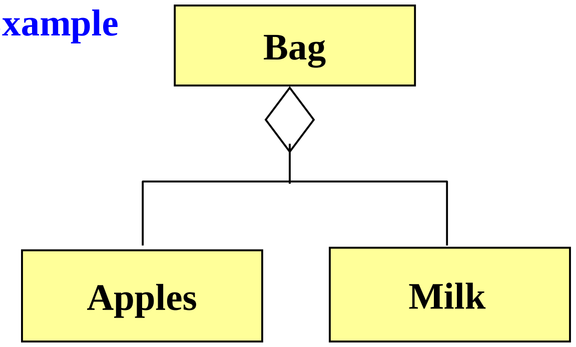
Aggregation



Aggregation



Example



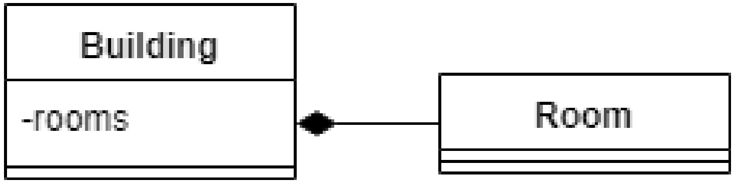
Aggregation:
expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

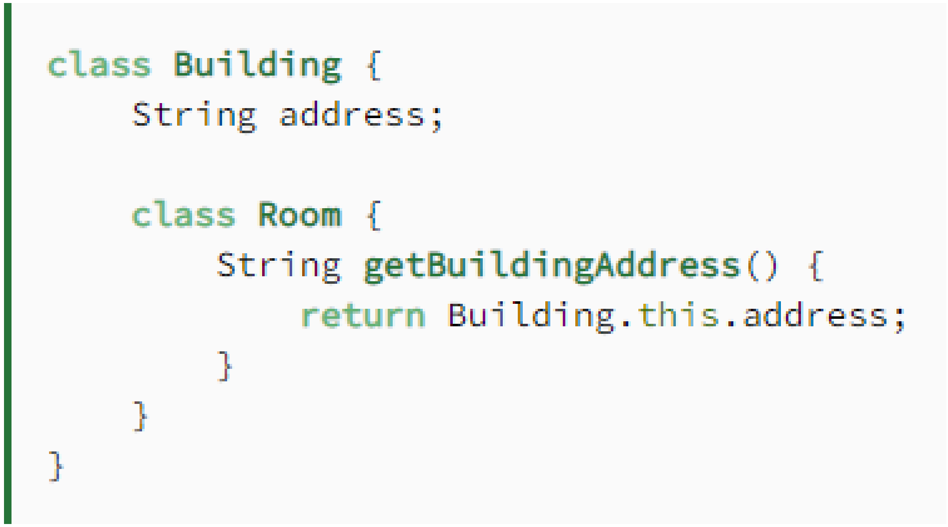
express a more informal relationship than composition expresses.

Compostion

- Another type of aggregation relationship, composition, is one in which the part elements cannot exist without the aggregate. For instance, the rooms in a house cannot continue to exist if the house is destroyed.
- For a composition relationship, a filled diamond is shown on the line near the aggregate.

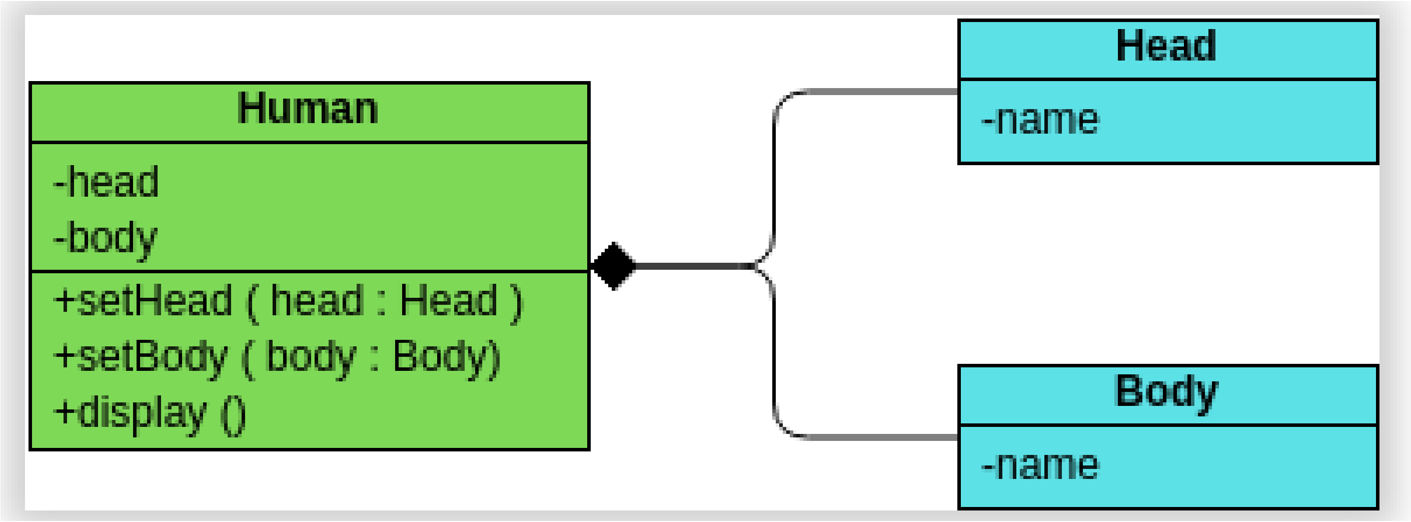
Composition



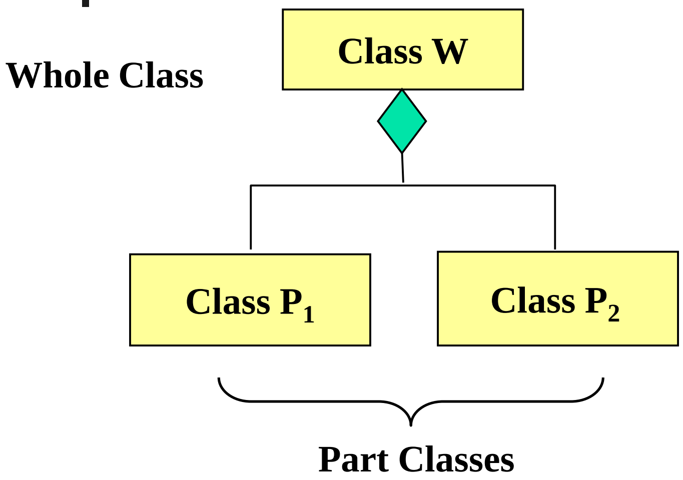


```
class Building {
    String address;

    class Room {
        String getBuildingAddress() {
            return Building.this.address;
        }
    }
}
```

Composition

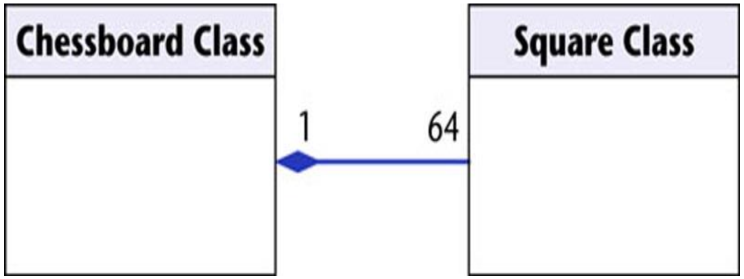


Composition



[From Dr.David A. Workman]

Example



Association

Models the part–whole relationship

Composition

Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

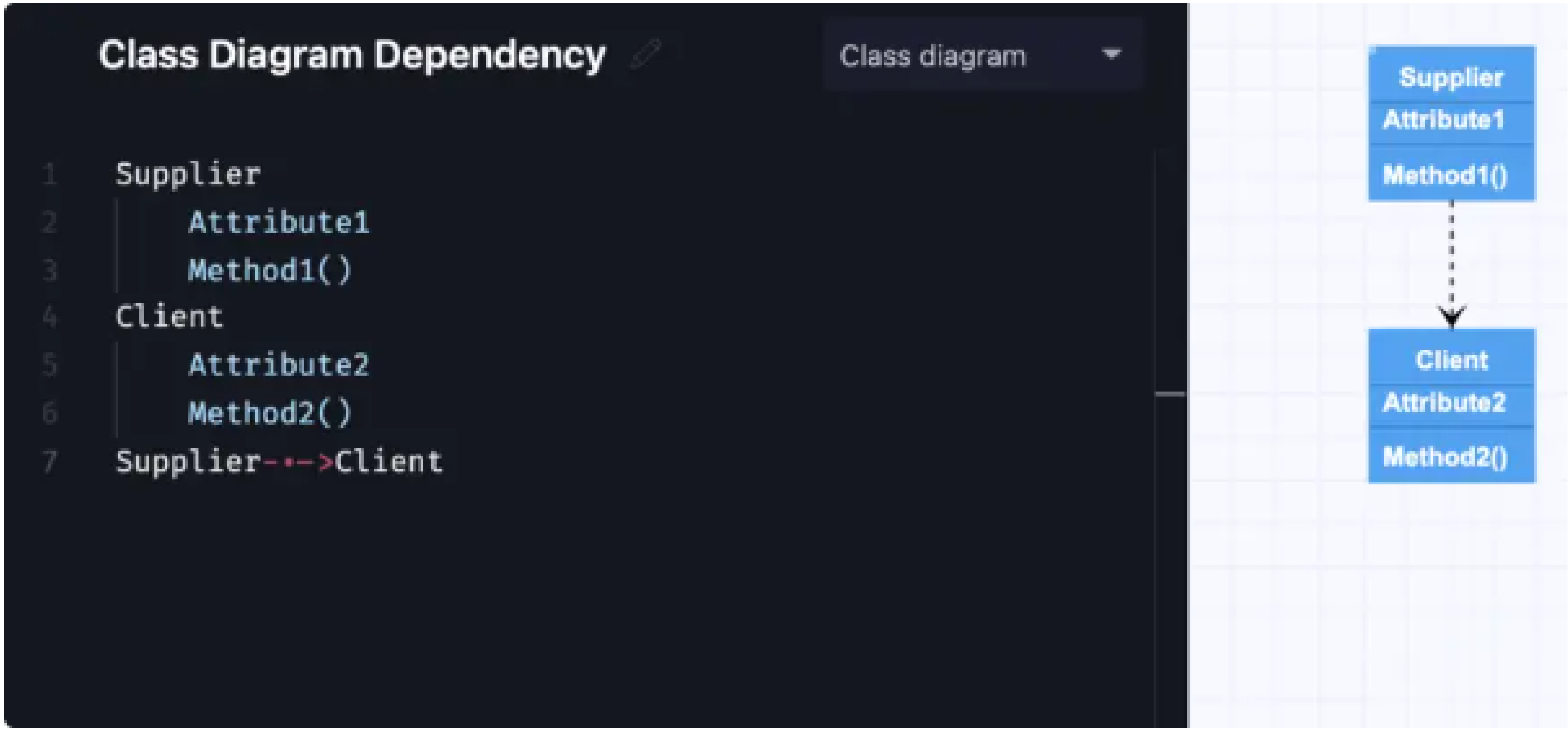
Example:

A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

Dependency

- Dependencies in UML indicate that a source element, also called the client, and target element, also called the supplier, are related so that the source element makes use of, or depends upon, the target element.
- Changes in the behavior or structure of the target may mean changes in the source.

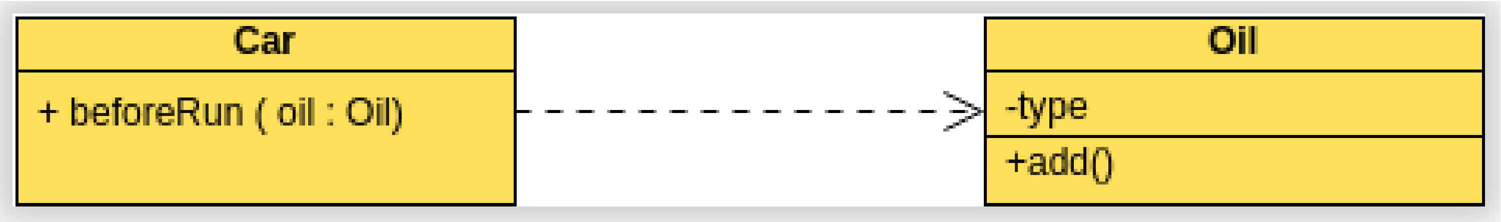
Dependency



Class Diagram Dependency

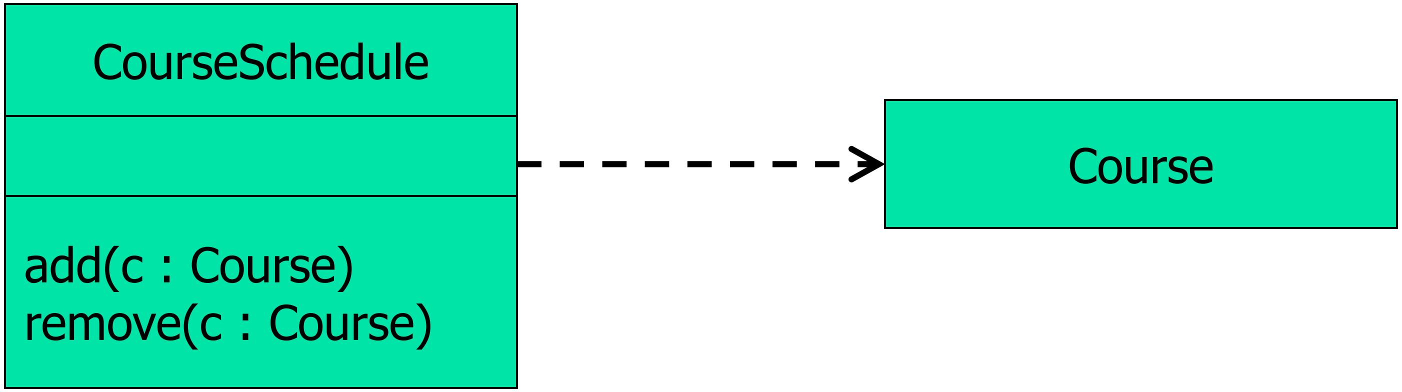
```
Supplier
    Attribute1
    Method1()
Client
    Attribute2
    Method2()
Supplier-.->Client
```

Dependency



Dependency

A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



Example

- Let us model a university, which has its departments. Professors work in each department, who also has friends among each other.

Example

- Will the departments exist after we close the university?

- Of course not, therefore it's a **composition**.

Example

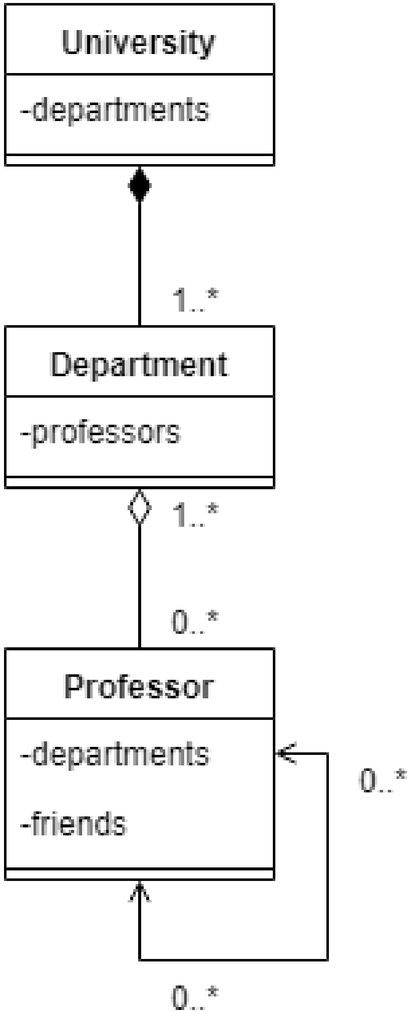
- But the professors will still exist (hopefully). We have to decide which is more logical: if we consider professors as parts of the departments or not. Alternatively: are they members of the departments or not?

- Yes, they are. Hence it's an aggregation. On top of that, a professor can work in multiple departments.

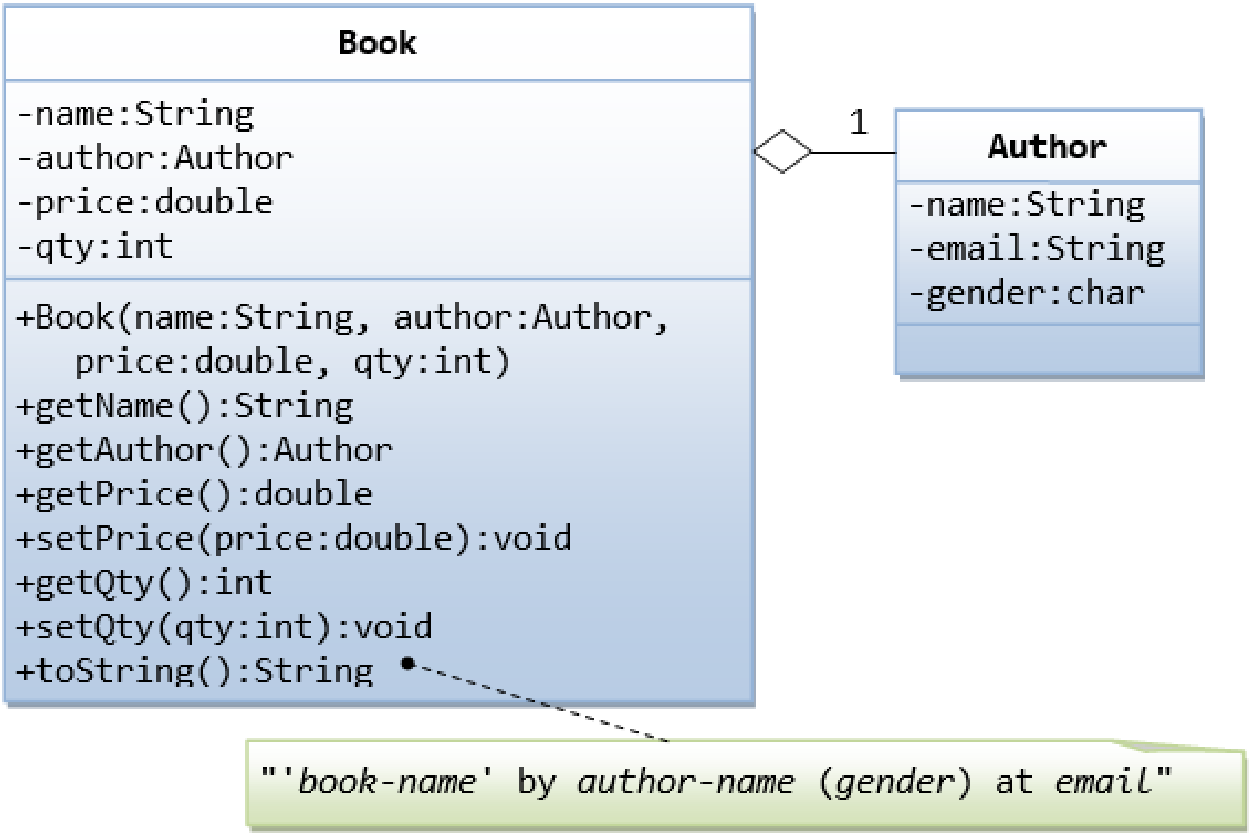
Example

- Finally, the relationship between professors is association because it doesn't make any sense to say that a professor is part of another one.

Example


University
-departments
1..*
Department
-professors
1..*
0..*
Professor
-departments
-friends
0..*
0..*

Exercise



Book

```
-name:String
-author:Author
-price:double
-qty:int
```

```
+Book(name:String, author:Author,
   price:double, qty:int)
+getName():String
+getAuthor():Author
+getPrice():double
+setPrice(price:double):void
+getQty():int
+setQty(qty:int):void
+toString():String
```

1

Author

```
-name:String
-email:String
-gender:char
```

"'*book-name*' by *author-name* (*gender*) at *email*"