Week 5 & 6:
Objects & Classes & Relationships

Overview of Lecture

- Objects
- Classes
- Constructors
- Accessors and Mutators
- Class Implementation Examples
- Static Variables and Methods
- Arrays of Objects
- Relationships

Objects

- Objects are instances of classes.
- They have state (attributes) and exhibit behavior (methods)

- Objects = state + behaviour + identity

- We would like objects to be:
  - highly cohesive - have a single purpose; make use of all features
  - loosely coupled - be dependent on only a few other classes

State

- The state of an object represents the condition that an object is in at a given moment i.e. the values of all its properties (attributes).
- e.g. The state of an employee object could be:

Property	Value
name	“Brad Pitt”
socialsecno	2425232237673
department	“Acting”
salary	10,000,000

- The properties of an object are known as attributes.

Behaviour and Identity

- Behaviour
  - An object performs some actions or is acted upon by other objects. This defines the behaviour of an object.
  - An object's behavior corresponds to the set of **methods** in its class because operations are implemented by methods in the object's class.
- Identity
  - Identity is that property of an object that distinguishes it from all other objects. [Booch]
  - Identity defines the uniqueness of each object.

Messaging

- Objects interact by sending messages
- Object A sends a message to Object B to ask it to perform a task
  - When done, B may pass a value back to A
  - Sometimes A == B
  - i.e., an object can send a message to itself

Messaging

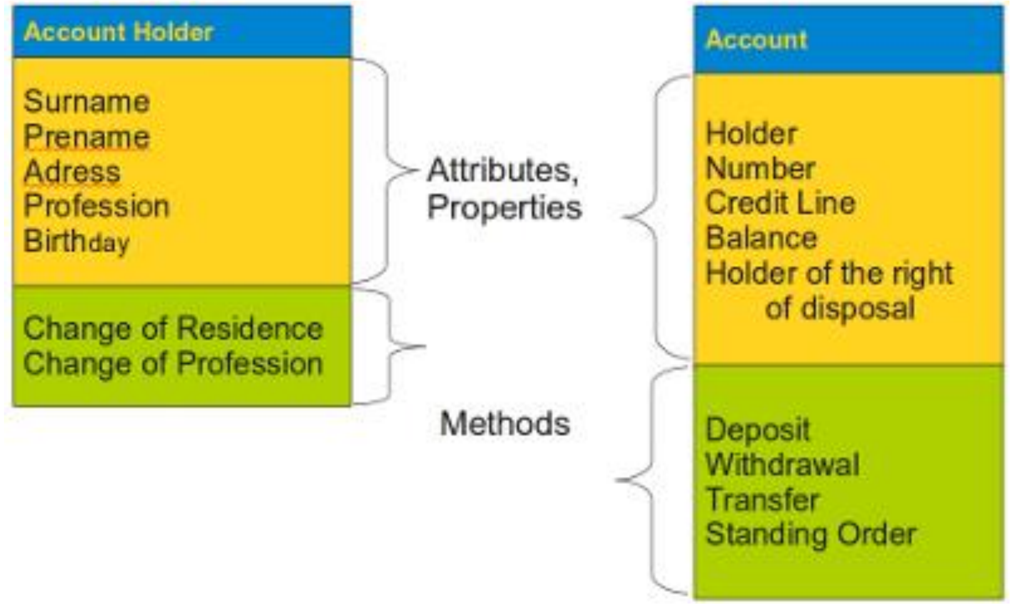
- In response to a message, an object may
  - update its internal state
  - return a value from its internal state
  - perform a calculation based on its state and return the calculated value
  - create a new object (or set of objects)
  - delegate part or all of the task to some other object
- i.e. they can do pretty much anything in response to a message

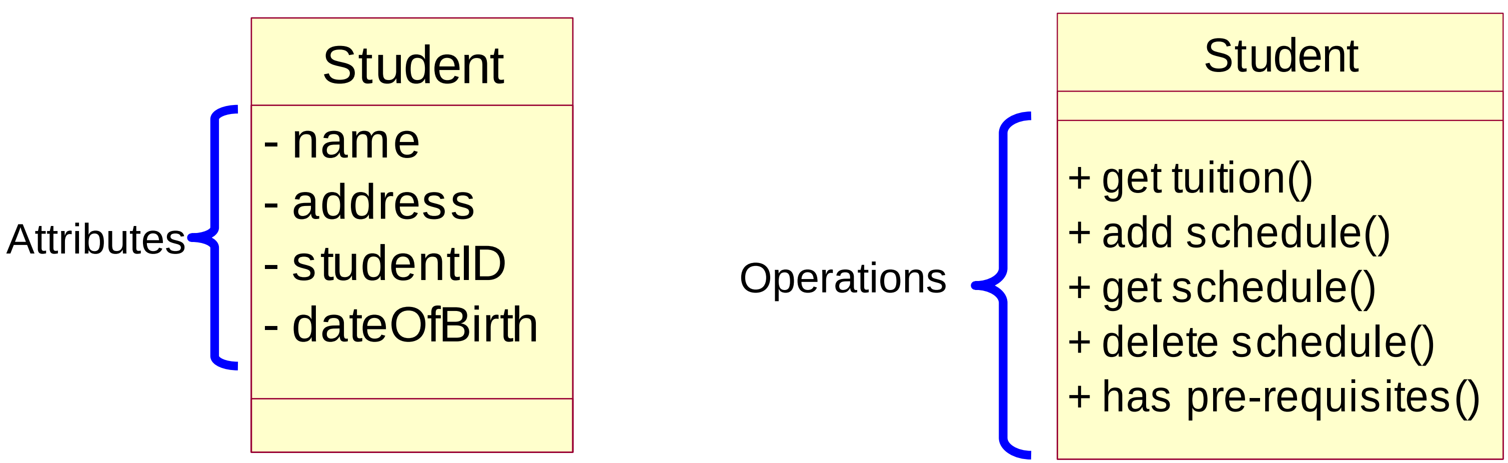
Class

- A **class** describes a group of objects with similar properties (attributes),common behaviour (operations/methods), and common relationships to other objects.

- **Objects** that are sufficiently similar to each other are said to belong to the **same class** i.e. they will share the same attributes, methods and relationships.

- Examples of classes: Person, Animal, Company
- Examples of objects: John Doe, Dog, Adgate Ltd.
- **John Doe, Mary Sharp, Adam, Smith are instances of the class Person…**


Account Holder
Surname
Prename
Adress
Profession
Birthday
Change of Residence
Change of Profession
Attributes,
Properties
Methods
Account
Holder
Number
Credit Line
Balance
Holder of the right
of disposal
Deposit
Withdrawal
Transfer
Standing Order


Student
- name
- address
- studentID
- dateOfBirth
Attributes
Student
+ get tuition()
+ add schedule()
+ get schedule()
+ delete schedule()
+ has pre-requisites()
Operations

Class with attribute and method

```
class Student { //className = 'Student'
        private String name = "Maunick"; // propertyName = 'name'
        private String id; // propertyName = 'id'

        public void setId(String id){ //methodName='setId'
        this.id =id; // setting value instance variable 'id' to value of parameter 'id'
        }
}
```

- Instance variables: name, id.
- Instance method: setId()
- Note: If no modifier is used for the class, then public is assumed.

Main Class

- The main class allows the execution of a java program.
- A java program can have many classes, but it should have at least 1 main class to be executable.
- The main class should contain the main method.
- E.g of a main class

```
class StudentMain{ //main class for student program
        public static void main(String[] args){
        //main method of the class
        Student uomStudent= new Student();
        // instantiates an object uomStudent from class Student
        }
}
```

Constructor (1) – Default Constructor

- The purpose of a constructor is to create an object based on a class's blueprint.

- A **constructor** has the same name as the name of the class to which it belongs. Constructor's signature does not include a return type, since constructors never return a value.

- Each class must have at least one constructor; if you do not write a constructor, java will create a default constructor for you.

- **The default constructor does not take any parameters.**
- E.g. For a class Student, the default constructor will look like `public Student() {//codes }`
- Writing `Student myStudent = new Student();` in the main class will invoke the default constructor from class Student.

Constructor (2) – Defined Constructor

- **Constructors can also take parameters.**
- E.g. public Student(String name, String id) { //codes}
- To invoke this constructor, the user has to supply the name and id values while creating the myStudent object in the main class
- i.e. Student myStudent = new Student (“Maunick”, “1324534”);

Constructor Example (1)

```
public class Student{
   private String name;
   private String id;

   public Student(){ //default constructor
      name="New Student";
      id="0000000";
   }

   public Student(String name, String id){ //defined constructor
      this.name=name;  // this keyword used to differentiate
      this.id=id;      between classwide variable 'name'
                       and constructor parameter 'name'
   }
}
```

Constructor Example (2)

```
public class StudentMain{
        public static void main(String[] args){
        Student student1 =  new Student();
        Student student2 =  new Student("Amanda", "1305342");
                }
}
```

Accessors

Accessor methods read property values and are conventionally named get<FieldName>()

E.g.

```
public int getMark() // accessor method for field mark
{ return mark;
}
```

- Return type of the method is ‘int’ as it return ‘mark’ which is of type ‘int’.

Mutators

- Mutator methods set property values and are often named set<FieldName>() etc.
- Mutators can be used to ensure that the property's value is valid in both range and type.
- E.g.

```
public void setMark(int myMark) //mutator method
{       if (myMark >= 0)
                this.mark = myMark
        else
                this.mark = 0;
}
```

Accessor and Mutator Methods

```java
public class Airplane {

    private int speed;

    public Airplane(int speed) {
        this.speed = speed;
    }

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

}
```

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

    Airplane a = new Airplane(5);
    a.setSpeed(10);

    System.out.println("The speed of the airplane is "+ a.getSpeed());
            }
}
```

Person Example (1)

- Create a class Person that has classwide attributes name and age.
- Include a default constructor that initialises name to ‘Person’ and age to18
- Include a defined constructor that takes in values for the two fields (name and age) and initialises the Person object.
- Include Accessor and mutator methods for every field.
- Include a method display to display the name and age

Person Example (2)

```
public class Person {
   private String name;
   private int age;

   public Person(){
      name = "Person";
      age = 18; }

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

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

public String getName(){
      return name;}

public void setAge(int age){
            this.age = age;
}

public int getAge(){
      return age;
   }

   public void display(){
      System.out.println("Name is: "+name);
      System.out.println("Age is: "+age);
   }
}
```

•

Person Example (3)

- Create a class PersonMain.

- Include the main method that allow the creation of two persons object, i.e, Person1 and Person2 respectively.
- Create an object Person1 using the default constructor
- Use the mutator methods to set Person1's name to "Maunick" and age to 21.
- Use the accessor methods to retrieve the attribute values of Person1 and display them in PersonMain.

- Create an object Person2 using a defined constructor with the following parameters Name: Amanda and Age: 20
- Display the attribute values of Person2

Person Example (4)

```
public class PersonMain {
    public static void main(String[] args){
        Person person1 =  new Person();
        person1.setName(“Maunick");
        person1.setAge(21);
        System.out.println("Name: "+person1.getName());
        System.out.println("Age: "+person1.getAge());
        Person person2 = new Person(“Amanda",20);
        person2.display();
    }
}
```

Employee Example (1)

```
public class Employee{
  private String name;
  private int age;
  private String designation;
  private double salary;

  // This is the constructor of the class Employee
  public Employee(String name){
    this.name = name;
  }
  // Assign the age of the Employee  to the
variable age.
  public void empAge(int empAge){
    age =  empAge;
  }
```

```
/* Assign the designation to the variable
designation.*/
  public void empDesignation(String empDesig){
    designation = empDesig; }

  /* Assign the salary to the variable salary.*/
  public void empSalary(double empSalary){
    salary = empSalary; }

  /* Print the Employee details */
  public void printEmployee(){
    System.out.println("Name:"+ name );
    System.out.println("Age:" + age );
    System.out.println("Designation:" +
designation );
    System.out.println("Salary:" + salary);
  }
}
```

Employee Example (2)

```
public class EmployeeTest{

   public static void main(String args[]){
      /* Create two objects using constructor */
      Employee empOne = new Employee("James Smith");
      Employee empTwo = new Employee("Mary Anne");

      // Invoking methods for each object created
      empOne.empAge(26);
      empOne.empDesignation("Senior Software Engineer");
      empOne.empSalary(1000);
      empOne.printEmployee();

      empTwo.empAge(21);
      empTwo.empDesignation("Software Engineer");
      empTwo.empSalary(500);
      empTwo.printEmployee();
   }
}
```

Class Variable

- **Static Variable (Class Variable)**
- Sometimes it is desirable to have a variable that is shared among all instances of a class. You can achieve this effect by marking the variable with the keyword static. Such a variable is sometimes called a class variable to distinguish it from a member or instance variable which is not shared.
- Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier. Fields that have the static modifier in their declaration are called *static fields* or *class variables*. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory.
- Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

Static variable example

```
class Count {
private int serialNum;
private static int counter =0;
Count() {
        counter ++;
        serialNum = counter;
}
void display() {
        System.out.println(“Serial
        number: “ +serialNum); }
}
```

```
public class CountObject {
public static void main (String args[]) {
        Count product1 = new Count();
        product1.display();
        Count product2 = new Count();
        product2.display();
        Count product3 = new Count();
        product3.display();
        }
}
```

Static Methods

- Sometimes you need to access program code when you do not have an instance of a particular object available.

- A method that is marked using the keyword static can be used in this way and is sometimes called a class method.

- A static method can be invoked without any instance of the class to which it belongs, i.e. methods that are static can be accessed using the class name rather than a reference.

- Static variables and methods are closely related together.
  - Non-static methods can operate with static variables.
  - Static methods can only deal with static variables and static methods.

Static methods example

```
class GeneralFunction {
static int addUp(int x, int y) {
return x+y;
}
}
class UseGeneral {
public void useAdd() {
int a = 9, b = 10;
int c = GeneralFunction.addUp(a,b);
System.out.println("addUp() gives "
+c);
}
}
```

```
class StaticMethods {
public static void display () {
System.out.println("This is a static
method.");
}
public static void main(String args[]) {
UseGeneral us = new UseGeneral();
//call the useAdd() to call static addUp()
us.useAdd();
display();
}
}
```

The Bicycle Class

```
public class Bicycle {

   private int cadence;
   private int gear;
   private int speed;

   private int id;

   private static int numberOfBicycles = 0;


   public Bicycle(int startCadence,
              int startSpeed,
              int startGear){
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;

      id = ++numberOfBicycles;
   }

   public int getID() {
      return id;
   }

   public static int getNumberOfBicycles() {
      return numberOfBicycles;
   }
```

```
 public int getCadence(){
      return cadence;
   }

   public void setCadence(int newValue){
      cadence = newValue;
   }

   public int getGear(){
   return gear;
   }

   public void setGear(int newValue){
      gear = newValue;
   }

   public int getSpeed(){
      return speed;
   }

   public void applyBrake(int decrement){
      speed -= decrement;
   }

   public void speedUp(int increment){
      speed += increment;
   }
}
```

Array of Objects Example(1)

```java
public class Employee {
	private String name;
	private double salary;

	Employee (String name, double salary) {
		this.name = name;
		this.salary = salary;
	}

	void display() {
		System.out.println("Name = "+name);
		System.out.println("Salary = "+salary);
	}
}
```

Array of Objects Example(2)

```
import java.util.*;
public class EmployeeRecords {
        public static void main (String args[]) {
        Employee emp[]=new Employee[3];
        String name;
        double salary;
        Scanner sn=new Scanner(System.in);

        for (int i=0;i<3;i++){
                System.out.println("Enter name: ");
                name=sn.nextLine();
                System.out.println("Enter salary: ");
                salary=sn.nextDouble();
                sn.nextLine();
                emp[i]=new Employee(name,salary); }

        for (int j=0;j<3;j++){
                emp[j].display();       }
        sn.close();
        }
}
```

Passing classes to methods by reference

/* This program demonstrates the concept of passing classes to functions. If a class member is modified within a function, it will also modified for the calling code. */

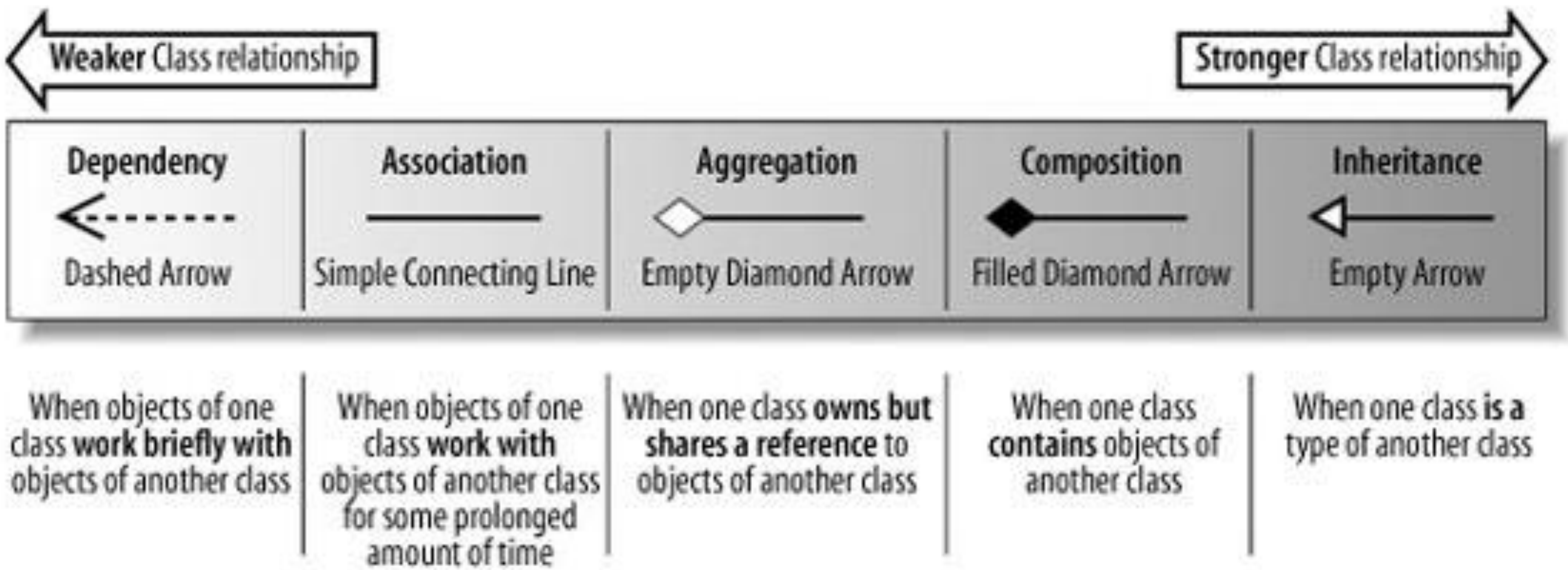
```
import java.awt.Point;
public class Params2 {
        public static void main(String[] args) {
        Point p = new Point(10,20);
        System.out.println("before calling: " + p);
        func(p);
        System.out.println("after calling: " + p);
        }

        public static void func(Point q) {
                System.out.println(" before modification: " + q);
                q.x++;
                q.y++;
                System.out.println(" after modification: " + q);
        }
}
```

Relationships:
Aggregation, Association, Generalisation

Relationships

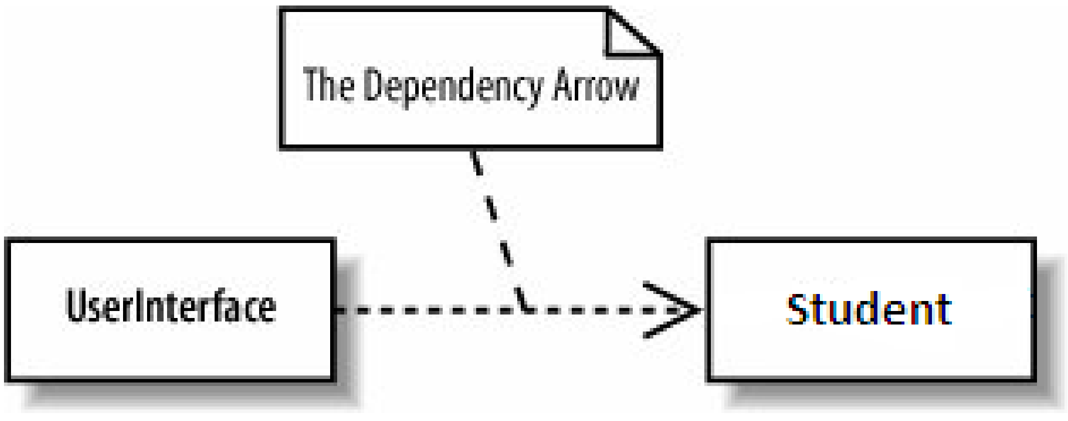
- Classes do not live in a vacuum; they work together using different types of relationships.
- Relationships between classes come in different strengths as shown in the diagram (See next slide).
- The strength of a class relationship is based on how dependent the classes involved in the relationship are on each other.
- Two classes that are strongly dependent on one another are said to be tightly coupled ; changes to one class will most likely affect the other class.
- Tight coupling is usually, but not always, a bad thing; therefore, the stronger the relationship, the more careful you need to be.

Types of relationships



Dependency

- A dependency between two classes declares that a class needs to know about another class to use objects of that class.
- E.g. If the UserInterface class of the OO-SIS needed to work with a Student class's object, then this dependency would be drawn using the dependency arrow, as shown in below



Dependency (2)

- The **UserInterface** and **Student** classes simply work together at the times when the user interface wants to display the student details.

- In class diagram terms, the two classes of object are dependent on each other to ensure they work together at runtime.

- A dependency implies only that objects of a class **can** work together; therefore, it is considered to be the weakest direct relationship that can exist between two classes.

- **Note:** The dependency relationship is often used when you have a class that is providing a set of general-purpose utility functions, such as in Java's regular expression (java.util.regex) and mathematics (java.math) packages. Classes depend on the java.util.regex and java.math classes to use the utilities that those classes offer.

Association

- Associations are meaningful relationships between classes.
- Just as a class describes a set of similar instances, an association describes a set of similar links.
- Links relate *objects.*
- Associations relate *classes.*
- You give the association a name to help others understand the nature of the relationship between two classes.

Association example

- A simple **rents** association between the Client class and the Car class.
- Clients do not purchase or make cars; clients rent cars.

- **Note:** Because a link between two objects carries the same name as the association between the objects' classes, the link name is often omitted.

Associations naming

- When you name an association, use a verb phrase that best describes what these two classes do with (or to) each other.

- If you consider the classes at either end of an association along with the association name, then the whole thing can be read as a sentence, such as, "**A client rents a car**."

- Although associations have meaning in both directions, the name you choose should be readable from left to right or from top to bottom when someone is looking at your diagram.

- Q:What happens if you need to read from right to left?
- A: Use arrow head <. And ^ for bottom to top.

- **Note:** Try to find an active verb phrase that relates the two classes. This enables others to understand your diagrams more easily.

Associations

- An association provides an abstract and general way of specifying links between two classes.

Multiplicity (Relating many Objects) -1

- As in the real world, you can link one object to many instances of another class.

- Surely, if you want to have a successful business renting Movies to clients, your clients should be able to rent more than one Movie at a time—and a Movie should be rentable to more than one client over time.

- Specifying how many instances can be linked together is called multiplicity.

Multiplicity (Relating many Objects) - 2

- When showing multiplicity on your association, remember to do the following:

- Position the multiplicity numbers above or below the association line, close to the class.

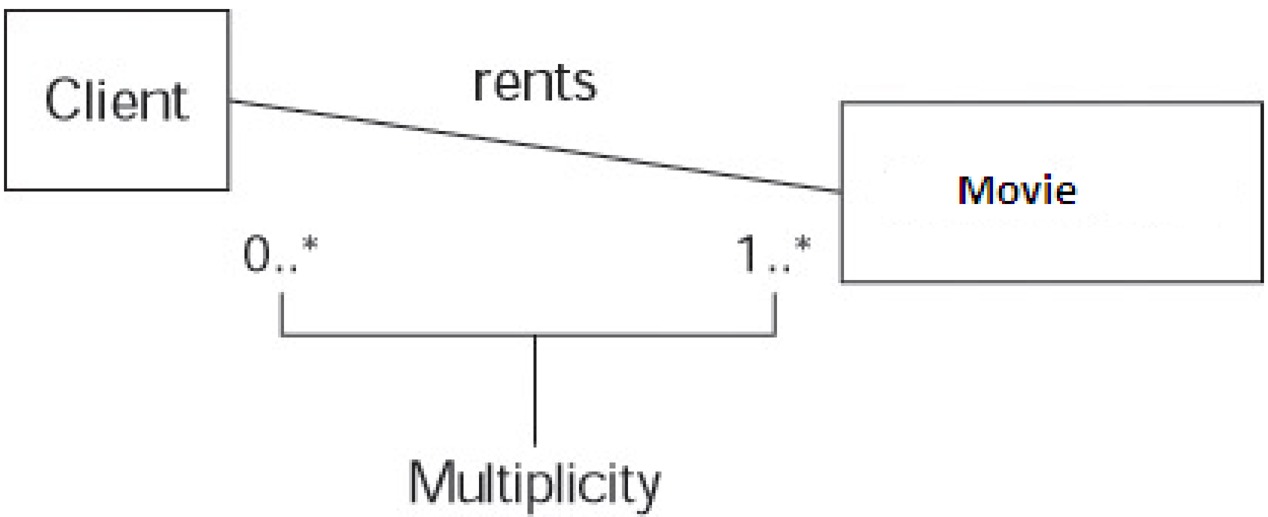
- Place multiplicity numbers at both ends of an association.

- Use multiplicity to show how many things at either end of an association are potentially linked together.

Multiplicity example

- A client rents at least one or more movies. In other words the appearance of 1..* represents the idea of having one or more instances of Movies that a Client rents. The 1 in the 1..* means that a client *must* rent at least one movie. The * in 1..* indicates that a client can rent more than one movie, and does not place an upper limit in the number that can be rented.

- Because associations have meaning in both directions, you also place a multiplicity symbol on the association line next to the Client class. Below, you see that a Movie can be rented by zero or more instances of Client (0..*).

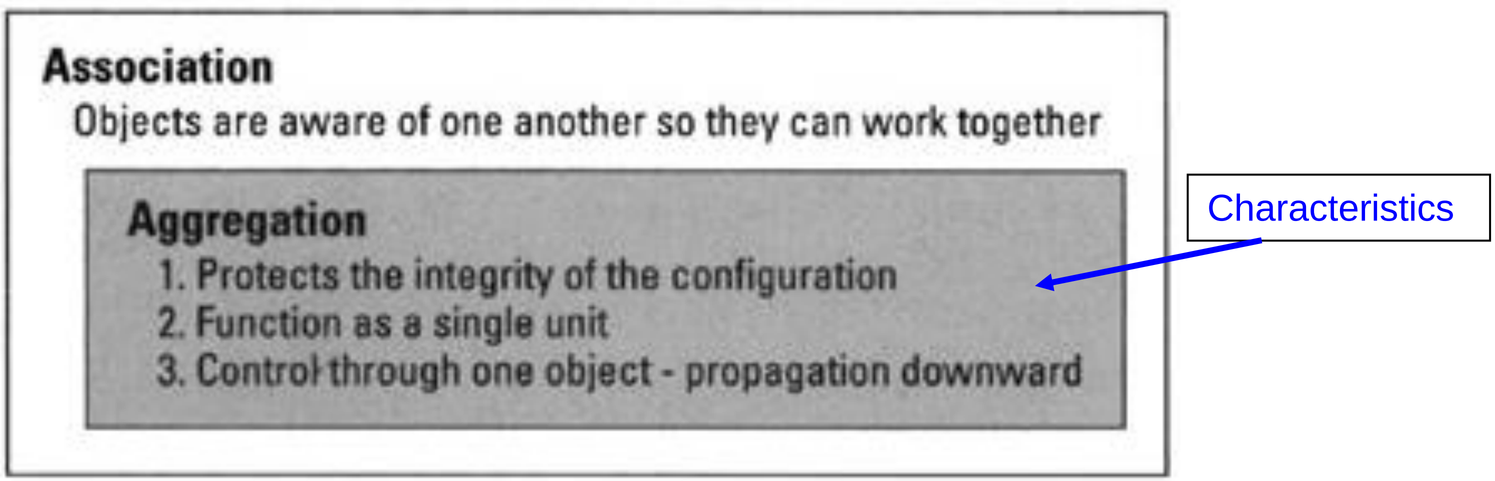


Aggregation

- Aggregation is a strong form of association in which an aggregate is *made of* components. Components are *part of* the aggregate.
- In a typical association the participating classes are peers. Each class remains independent of the other and neither class is superior to the other. They simply communicate.
- Aggregation defines a definite hierarchical relationship.
- Saying "The whole is greater than the sum of the parts" is true with aggregation
- In an aggregation of objects there has to be a point of control, a boss, one object that represents the interface to the assembly and assumes responsibility for coordinating the behavior of the aggregation.
- **Aggregations are usually read as "...owns a...".**

Relationship between association and aggregation

- The diagram shows that aggregations are a subset of all associations.
- The aggregation subset is a specialization that introduces a new set of characteristics that regular associations do not have:



Aggregation Example (1)

- If you have a class such as SalesRegion and you want to model the SalesRegion and its parts (such as office, RetailOutlet, etc), you use aggregation.

- Aggregation is the relationship between the whole and its parts.

Aggregation Example (2)

- SalesRegion is the whole.
- Office, RetailOutlet and WholesaleWarehouse are the parts.
- Hollow diamonds placed at whole always!

A weak form of aggregation-some parts survive if the whole goes away

Advantages of Aggregation

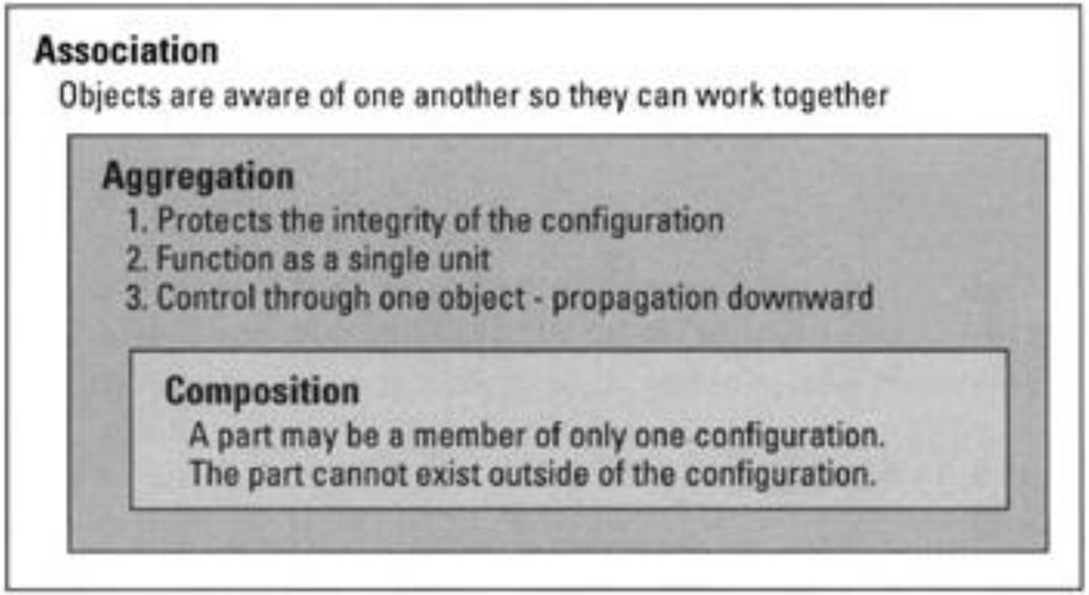
- Aggregation derives its advantages mostly from the use of encapsulation
  - Simplicity
  - Safety
  - Specialisation
  - Structure
  - Substitution

Composition

- Composition is used for aggregations where the life span of the member object depends on the life span of the aggregate.
- The aggregate not only has control over the behavior of the member, but also has control over the creation and destruction of the member.
- In other words, the member object cannot exist apart from the aggregate.
- This greater responsibility is why composition is sometimes referred to as strong aggregation
- A composition relationship is usually read as "...is part of..." (the whole)

Relationship between association, aggregation and composition

- The diagram shows that composition is a subset of aggregation, just as aggregation is a subset of association.



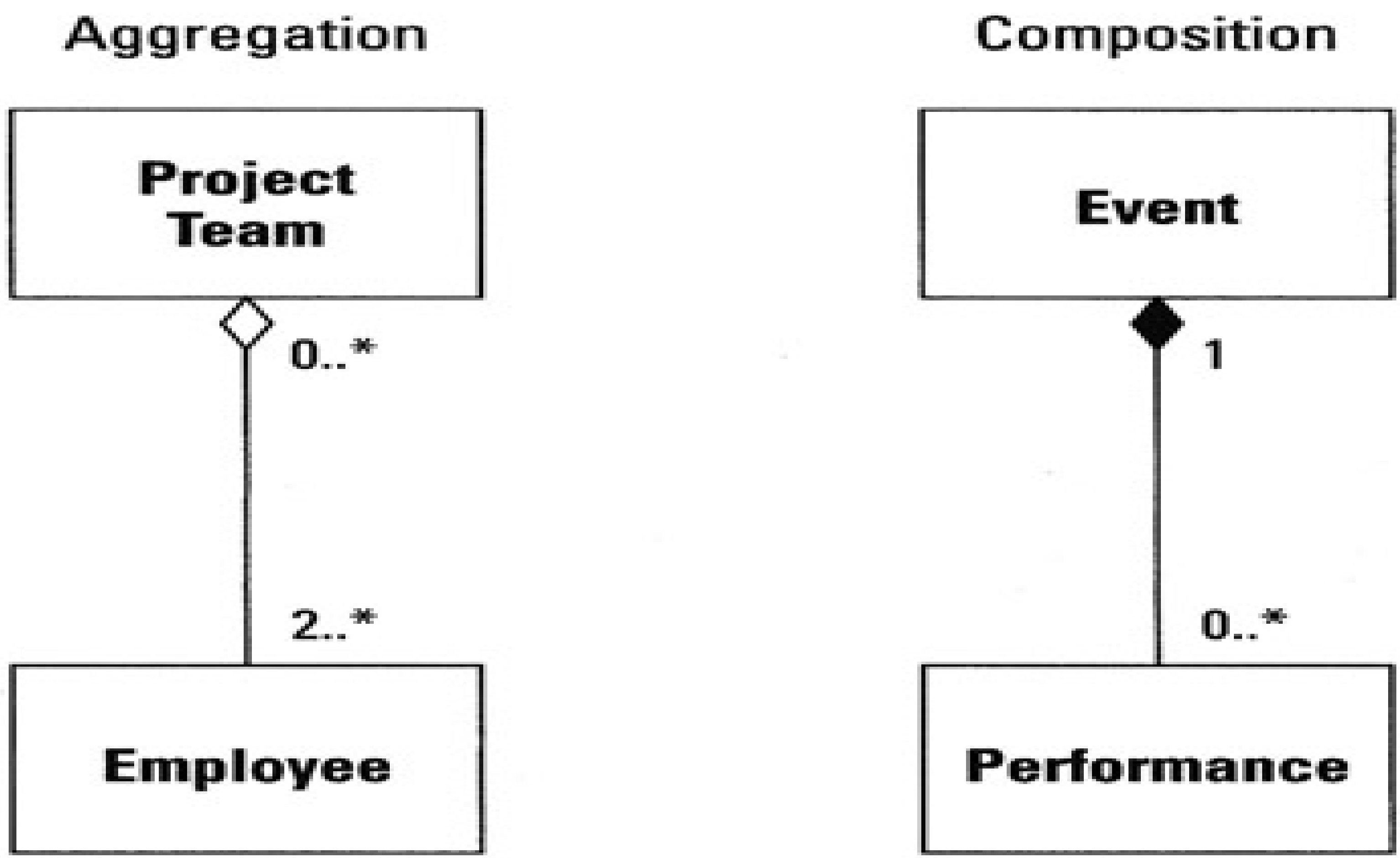
Draw this stronger form of aggregation simply by filling in the aggregation diamond

Composition Example (1)

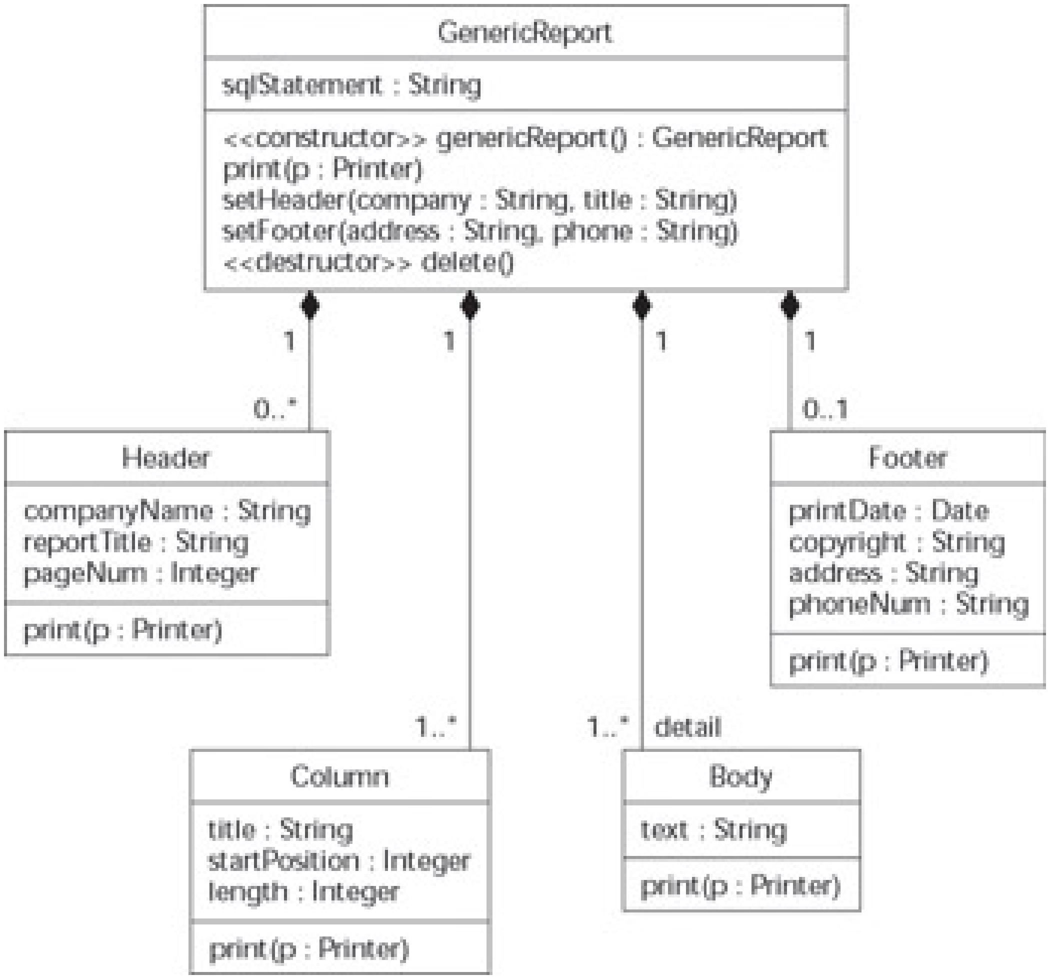
- The project team example (next slide) uses aggregation, the hollow diamond; employees are assembled into a project team.
- But if the team is disbanded, the employees live on

- The theater event example uses composition, the solid diamond; an event is composed of one or more performances.
- The performances would not continue to exist elsewhere on their own. If the event were deleted, the performances would cease to exist along with the event

Composition Example (2)



Composition Example (3)



Example of composition, a strong form of aggregation.

Composition v/s Aggregation (1)

- The multiplicity provides some clues about the distinction between aggregation and composition. On the project team example, each employee may or may not be a member of a project team (0..*).
- Employees may exist independent from project teams. In fact, an employee may simultaneously participate in many project teams.
- Aggregation is a weak association in that it allows the members to participate or not participate or even participate in other aggregations at the same time.

- Composition is exemplified by the event example that says that a performance must be associated with one and only one event (1..1). This means that a performance cannot exist independent of the event.
- Composition does allow a member object to be moved to another composite before the original composite is destroyed.

Composition vs Aggregation (2)

- It might be difficult to decide between modeling a relationship as an association, an aggregation, or a composition.
- Clues to look for when you’re modeling relationships:
- If you hear words like “part of,” “contains,” or “owns,” then you probably have an *aggregation* relationship.
- If the life-cycle of the parts are bound up within the life-cycle of the whole, then you have a *composite*.
- If the parts are shared, then it’s an *aggregation*.
- If the parts are not shared, then you may have *composition*.

Composition vs Aggregation (3)

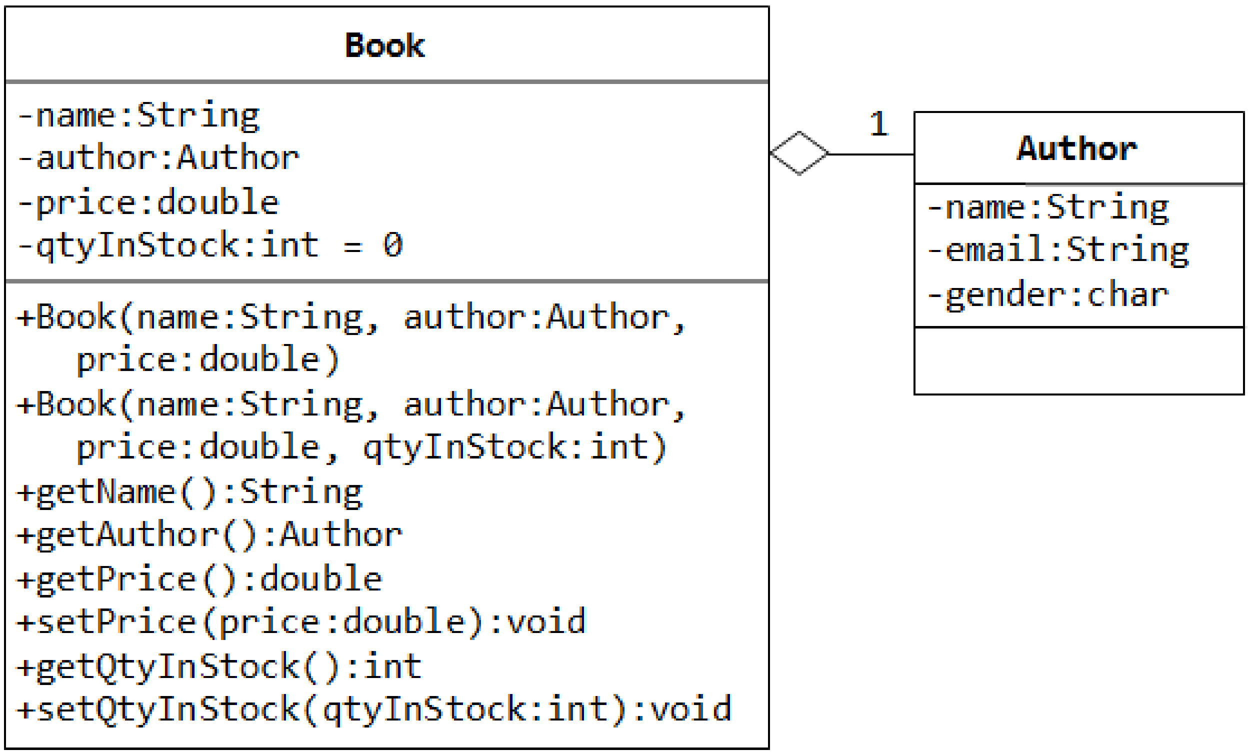
Decision Result	Criteria
Aggregation or Composition	Part-of, contains, owns words are used to describe relationship between two classes
Aggregation or Composition	No symmetry
Aggregation or Composition	Transitivity among parts
Composition	Parts are not shared
Composition	Multiplicity of the whole is *1* or *0..1*
Aggregation	Parts may be shared
Aggregation	Multiplicity of the whole may be larger than 1
Association	Relationship does not fit the other criteria

Example: Aggregation

Author
-name:String -email:String -gender:char
+Author(name:String, email:String, gender:char) +getName():String +getEmail():String +setEmail(email:String):void +getGender():char +toString():String

- A class called Author is designed as shown in the class diagram. It contains:
  - Three private instance variables: name (String), email (String), and gender (char of either 'm' or 'f');
  - One constructor to initialize the name, email and gender with the given values;
  - public Author (String name, String email, char gender) {......}
  - (There is no default constructor for Author, as there are no defaults for name, email and gender.)
  - public getters/setters: getName(), getEmail(), setEmail(), and getGender();
    (There are no setters for name and gender, as these attributes cannot be changed.)
  - A toString() method that returns "*author-name (gender) at email*", e.g., "Tan Ah Teck (m) at ahTeck@somewhere.com".

Example: Aggregation



Example: Aggregation

- A class called Book is designed as shown in the class diagram. It contains:
  - Four private instance variables: name (String), author (of the class Author you have just created, assume that each book has one and only one author), price (double), and qtyInStock (int);
  - Two constructors:
  - public Book (String name, Author author, double price) {...}
  - public Book (String name, Author author, double price,
  - int qtyInStock) {...}
  - public methods getName(), getAuthor(), getPrice(), setPrice(), getQtyInStock(), setQtyInStock().
  - toString() that returns "*'book-name' by author-name (gender) at email*". (Take note that the Author's toString() method returns "*author-name (gender) at email*".)

Author anAuthor = new Author(......);

Book aBook = new Book("Java for dummy", anAuthor, 19.95, 1000);

Composition and Java

- Composition does have a coding equivalent in Java

- It is the private *inner class* construct.

- Private inner classes in Java are inaccessible to any classes other than the outer class they are created within.
- As a result, they only serve that outer class and would generally be garbage collected when the object of the outer class is garbage collected

Generalisation (1)

- A generalization relationship conveys that the target of the relationship is a general, or less specific, version of the source class or interface. Generalization relationships are often used to pull out commonality between different things.

- if you had a class named Cat and a class named Dog, you can create a generalization of both of those classes called Animal

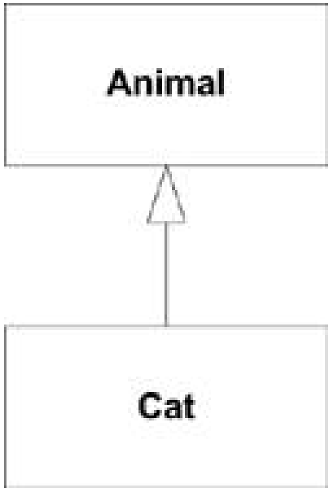
- Phrases like “**is-a** kind of" and “**is-a** type of" are often used to describe a generalization relationship between classes

- E.g. a cat is a type of animal, a play is a type of event and an agent is a kind of vendor.

Generalisation Example

- The generalization relationship is shown with a solid line with a closed arrow, pointing from the specific class to the general class.
- E.g. of the Cat to Animal relationship

Cat specializes the Animal base class



Unlike associations, generalization relationships are typically not named and don't have any kind of multiplicity.

Superclass

- A superclass has general characteristics that are inherited by all classes under it.

- The term superclass borrows from the superset concept.

- A superclass contains only the features that are common to every object in the set.

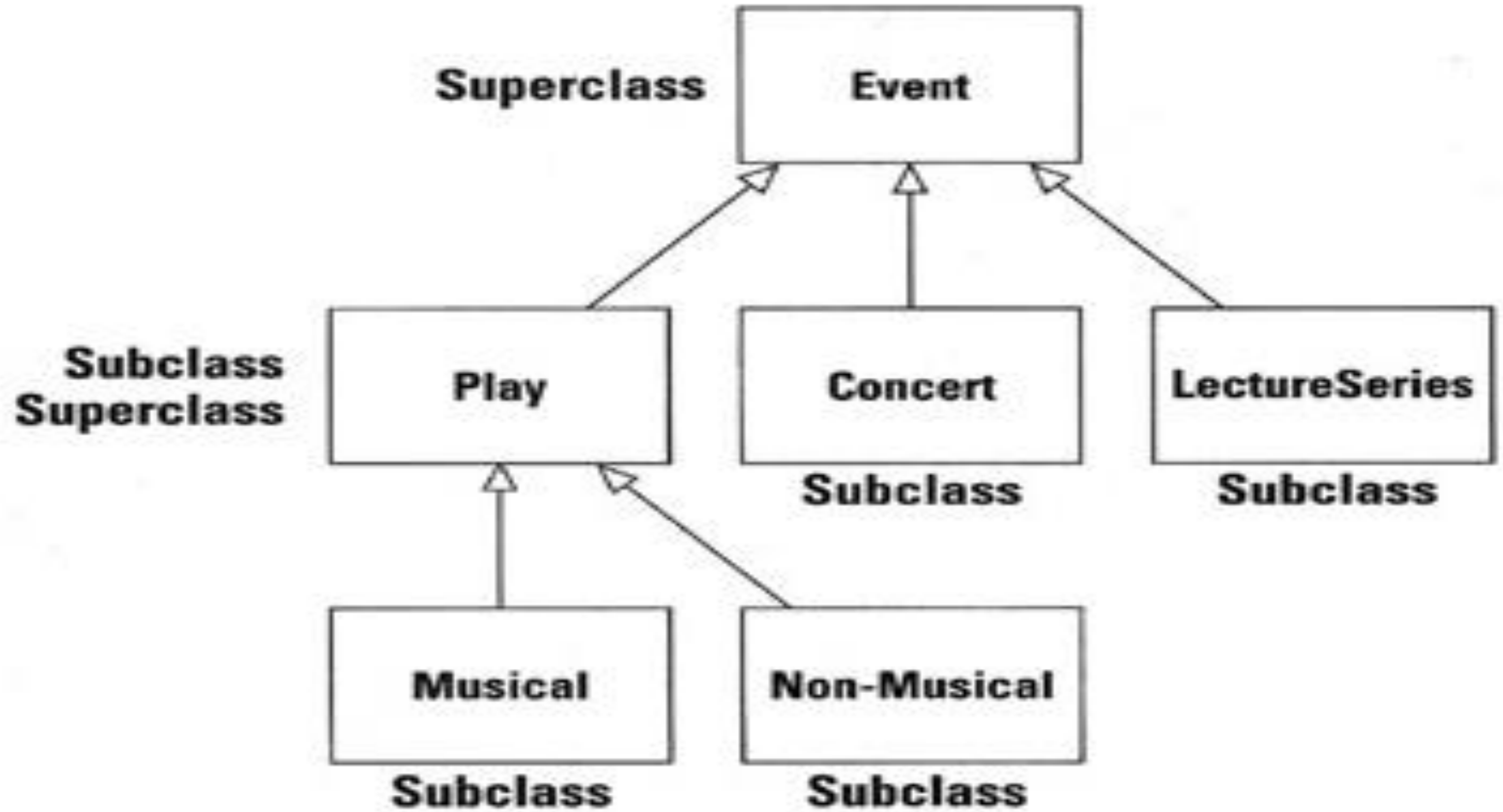
Subclass

- A subclass is a class that contains some combination of the features that are unique to a subset of the objects defined by a superclass.

- The term subclass reflects the subset concept.

- A subclass contains a set of features that is unique among the objects that make up the set of events

Superclass & Subclass Example



Superclasses and subclasses in a generalization hierarchy.

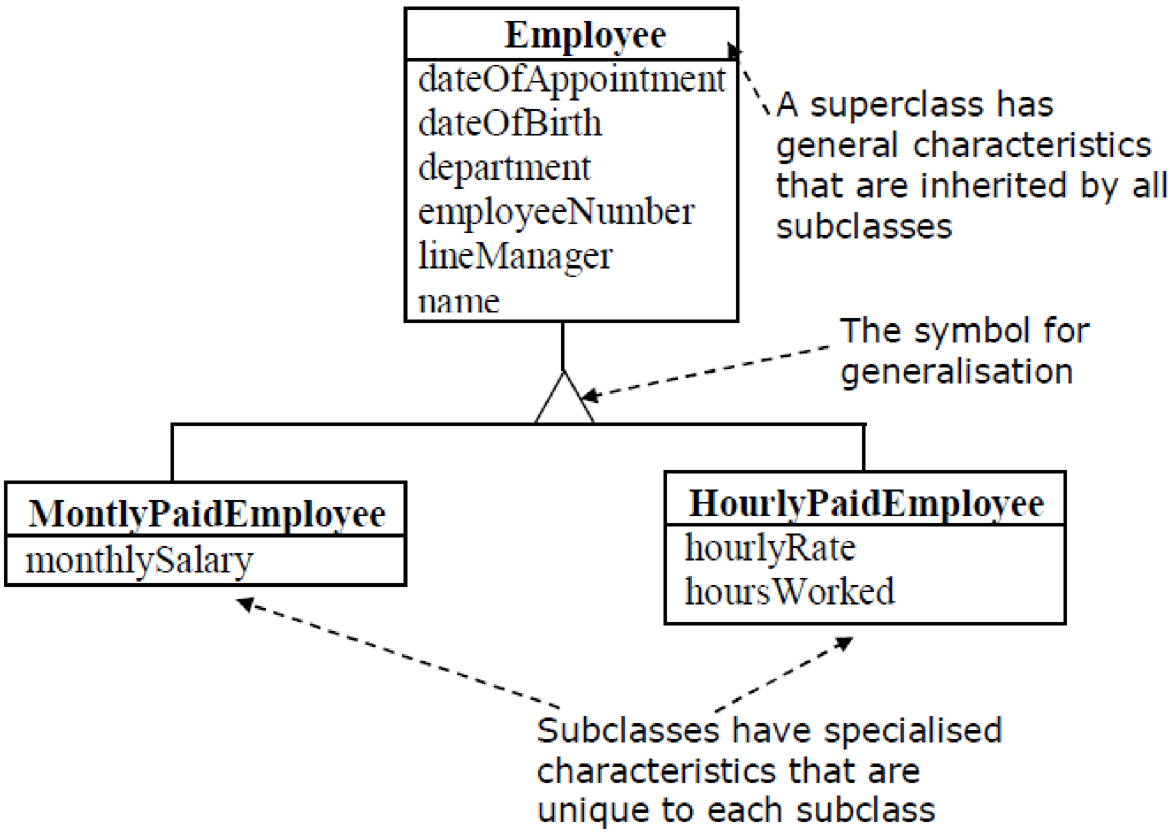
Benefits of Generalisation

- Using generalisation, we can build logical structures that make explicit the degree of similarity or difference between classes.

- A hierarchy can be easily extended to fit a changing picture. For example: If the company were to decide that a new, weekly paidtype of employee is required, it would be easy to add a new subclass to cater for this.

Adding a new subclass (1)

- Consider the following hierarchy.



Adding a new subclass (2)

- After adding the new subclass WeeklyPaidEmployee