NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

CS3005 –Software Design & Architecture Lab

Sobia Iftikhar "Sobia.iftikhar@nu.edu.pk"

Lab 04

Objective: To Understand Class Diagram

Class diagrams:

In software engineering, a class diagram in the **Unified Modeling Language (UML)** is **a type of static structure diagram** that gives an overview of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects -- they display what interacts but not what happens when they do interact.

Purpose of Class Diagrams:

1. Shows static structure of classifiers in a system
2. Diagram provides a basic notation for other structure diagrams prescribed by UML
3. Helpful for developers and other team members too
4. Shows static structure of classifiers in a system

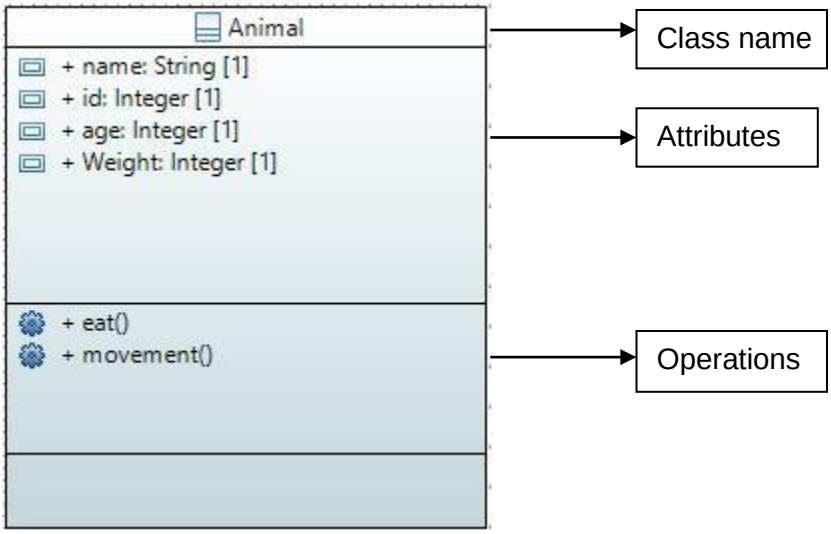
Class Notation:

UML class notation is a rectangle divided into three parts: class name, attributes, and operations.

1. **Class Name**
    - The name of the class appears in the first partition.
2. **Class Attributes**
    - Attributes are shown in the second partition.
    - The attribute type is shown after the colon.
    - Attributes map onto member variables (data members) in code.
3. **Class Operations** (Methods)
    - Operations are shown in the third partition. They are services the class provides.
    - The return type of a method is shown after the colon at the end of the method signature.
    - The return type of method parameters is shown after the colon following the parameter name.
    - Operations map onto class methods in code

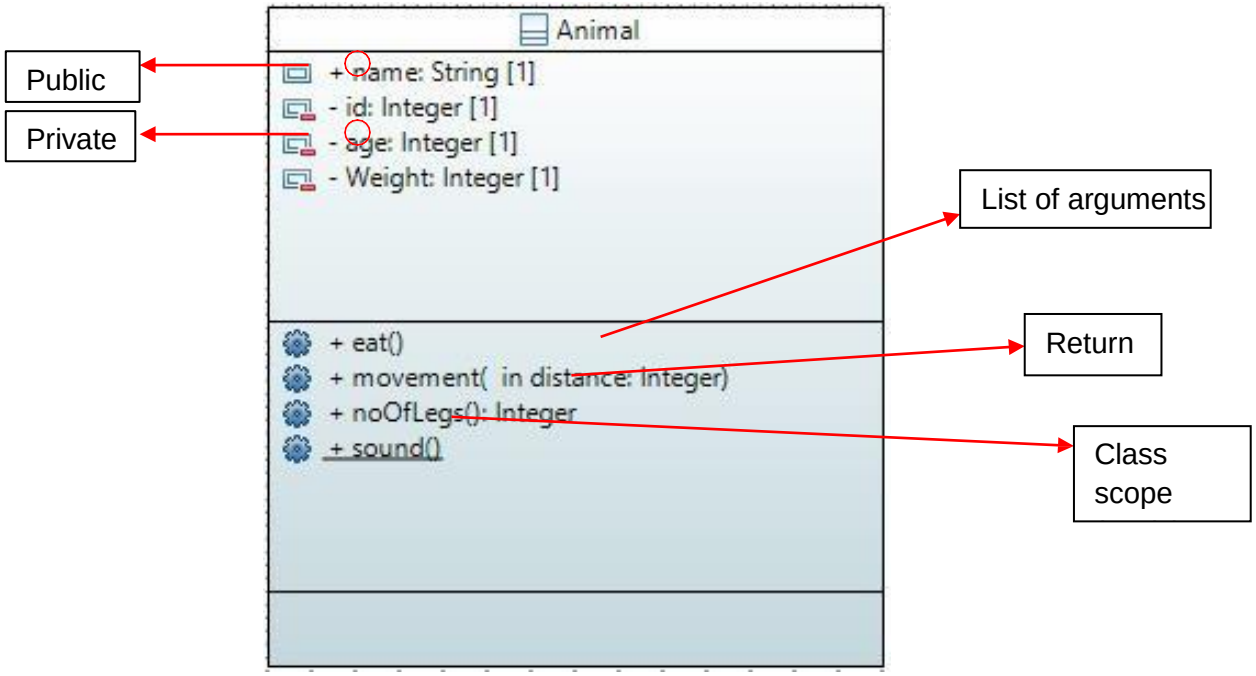
Example:

There are many animals in the zoo but you would model only one class called Animal, to represent entire collection of animals.



Class Information: Visibility and Scope

The class notation is a 3-piece rectangle with the class name, attributes, and operations. Attributes and operations can be labeled according to access and scope. Here is a new, expanded Animal class



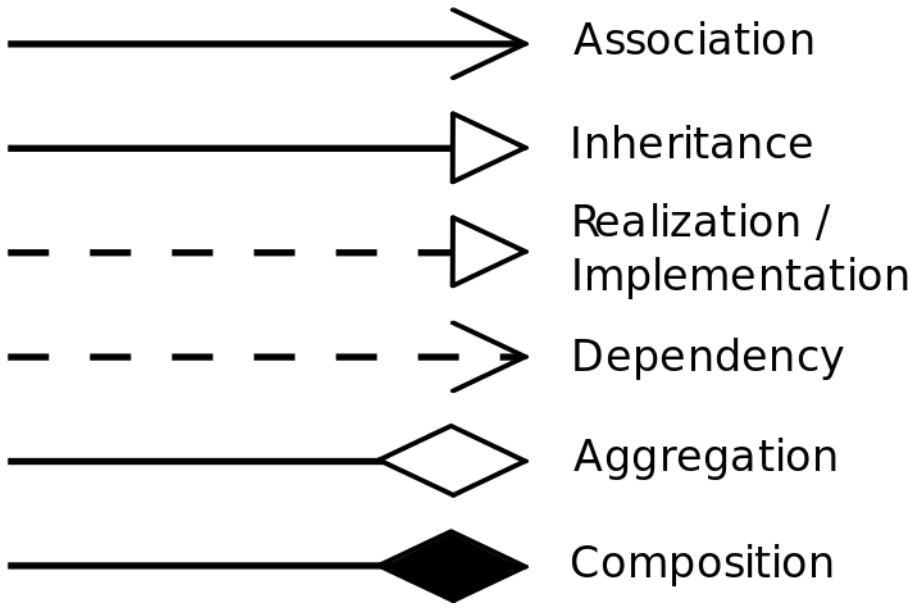
Symbol	Access
+	**public**
-	**private**
#	**protected**
~	**default**

UML Class Diagrams Relationship:

There are following key relationships between classes in a UML class diagram:

1. Association
2. Generalization/ Inheritance
3. Realization/ Implementation
4. Dependency
5. Aggregation
6. Composition

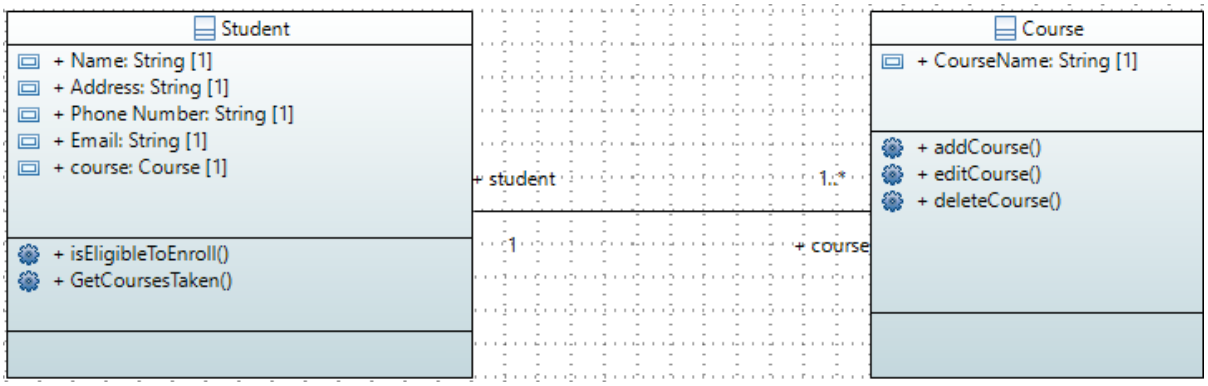
These six relationships are depicted below



1. **Association:**

There is an association between two classes if an instance of one class must know about the other in order to perform its work.

Example:



Student.class

```
public class Student {

	public Course course;
	public String Email;
	public String Address;
	public String Phone;
	public String Name;

	public Course getCourse() {
		return course;
	}

	public void setCourse(Course course)
{
		this.course = course;
	}

	public String getEmail() {
		return Email;
	}

	public void setEmail(String Email) {
		this.Email = Email;
	}

	public String getAddress() {
		return Address;
	}
	public void setAddress(String
Address) {
		this.Address = Address;
	}
}
```

Course.class

```
public class Course {


	public String CourseName;


	public String getCourseName() {
		return CourseName;
	}


	public void setCourseName(String
CourseName) {
		this.CourseName =
CourseName;
	}
	public void editCourse() {

	 }


	public void deleteCourse() {

	 }


	public void addCourse() {
		}

}
```

Main.java

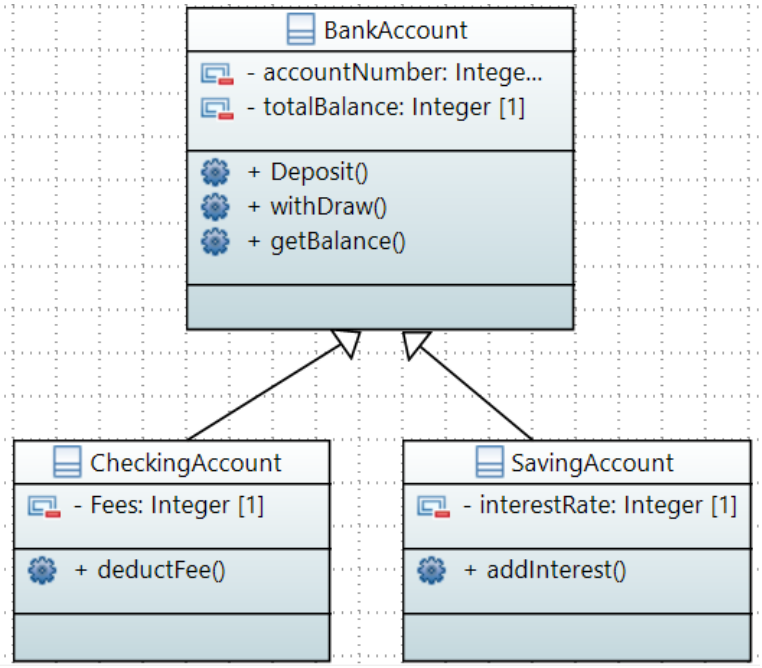
```
public class main {


		public static void main(String[] args) {
			Student student = new Student();
			Course sda = new Course();
			sda.setCourseName("SDA");

			student.setCourse(sda);

			Course savedCourse = student.getCourse();
System.out.println("Student Course: "+savedCourse.getCourseName());
		}

}
```

Multiplicities	Meaning
0..1	**zero or one instance. The notation *n* . . *m* indicates *n* to *m* instances.**
0..* *or* *	**no limit on the number of instances (including none).**
1	**exactly one instance**
1..*	**at least one instance**

2. **Generalization/ Inheritance**

   An inheritance link indicating one class is a superclass of the other. A generalization has a triangle pointing to the superclass.

   Example: CheckingAccount and SavingAccount classes are inheriting BankAccount class



BankAccount.java

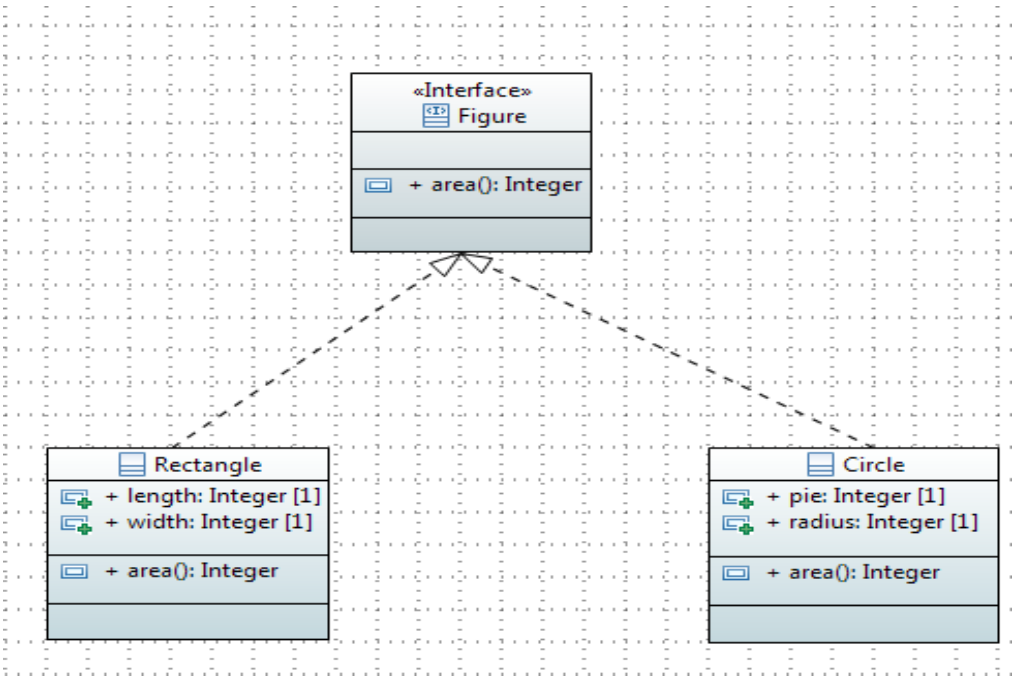
```
public class BankAccount {


	private Integer accountNumber;
	private Integer totalBalance;

	public Integer getAccountNumber() {
		return accountNumber;
	}

	public void setAccountNumber(Integer accountNumber)
{
		this.accountNumber = accountNumber;
	}

	public Integer getTotalBalance() {
		return totalBalance;
	}

	public void setTotalBalance(Integer totalBalance) {
		this.totalBalance = totalBalance;
	}

	public void getBalance() {
			}

	public void Deposit() {

	 }

	public void withDraw() {
			}

}
```

```
public class CheckingAccount extends
BankAccount {


        private Integer Fees;


        public Integer getFees() {
                return Fees;
        }


        public void setFees(Integer
Fees) {
                this.Fees = Fees;
        }
        public void deductFee() {
         }

    }

}
```

```
public class SavingAccount extends
BankAccount {


        private Integer interestRate;
        public Integer getInterestRate()
{
                return interestRate;
        }

        public void
setInterestRate(Integer interestRate) {
                this.interestRate =
interestRate;
        }


        public void addInterest() {
         }
  }
}
```

3. **Realization/Implementation:** Realization is the relationship between the interface and the implementing class.

Example: Figure interface might specify method for area. The Rectangle and Circles classes need to implement these methods, possibly in very different ways.



Shape.java

```
public interface Shape {


	public Integer area();

}
```

Circle.java

```
public class Circle implements Shape {


	public String length;
	public Integer width;
	public String getLength() {
		return length;
	}
	public void setLength(String length) {
		this.length = length;
	}

	public Integer getWidth() {
		return width;
	}

	public void setWidth(Integer width) {
		this.width = width;
	}

	public Integer area() {
		// TODO Auto-generated method
		return null;
	 }

}
```

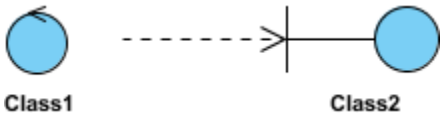
Rectangle.java

```
public class Rectangle implements Shape {
	public Integer length;
	public Integer width;

	public Integer getLength() {
		return length;
	}

	public void setLength(Integer length) {
		this.length = length;
	}

	public Integer getWidth() {
		return width;
	}

	public void setWidth(Integer width) {
		this.width = width;
	}

	public Integer area() {

		return null;
	 }

}
```

4. **Dependency**

A special type of association. Class1 depends on Class2

The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow



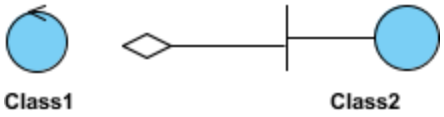
Example: Customer is dependent on Account as its makingDeposit using the class variable Account.

```
class Account{
public void deposit{
}
        }
 class Customer{
public void makeDeposit (Account acc)
{
acc.deposit();
}
        }
```

5. **Aggregation**

A special type of association.

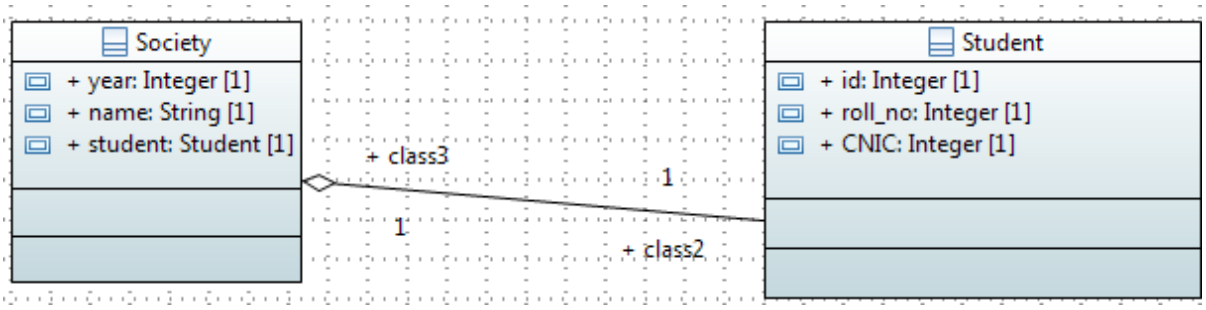
- It represents a "part of" relationship.
- Class2 is part of Class1.



- The HAS-A relationship is based on usage, rather than inheritance.
- class A has-a relationship with class B, if class A has a reference to an instance of class B

Example:

The Society class has an instance variable of type Student. As we have a variable of type Student in the Society class, it can use Society reference which is ad in this case, to invoke methods of the Student class.



Student.java

```
public classStudent{
int id, roll_no, CNIC;

public Student(int id, int roll_no, int CNIC) {
    this.id = id;
    this.roll_no = roll_no;
    this.CNIC = CNIC;
}
}
```

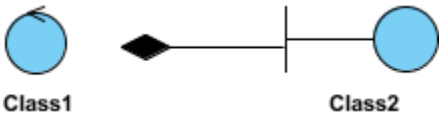
Society.java

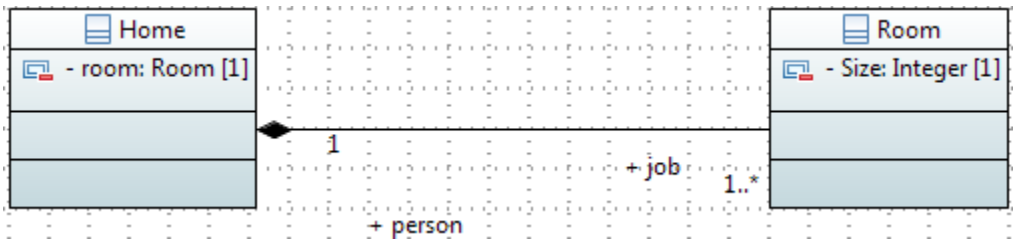
```
public class Society{
int year;
String name;
Student student;

public Society(int year, String name, int id, int roll_no, int CNIC) {
    this.year = year;
    this.name = name;
    this.student= new Student(id, roll_no, CNIC);
}
}
```

Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.

The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.





Room.java

```
public class Room {
	private long size;

	public long getSize() {
		return size;
	}
	public void setSize(long size) {
		this.size = size;
	}

}
```

Home.java

```
public class Home  {
	//composition has-a relationship
	private Room room;

	public  Home(){
		this.room=new  Room();
		job.setSize(1000L);
	}
	public long getSize() {
		return room.getSize();
	}

}
```

Main.java

```
public class TestHome {

	public static void main(String[] args) {
		Home home = new Home();
		long size = home.getSize();
	}

}
```