National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2024, Lab Manual – 09

Course Code: CL-217	**Course : Object Oriented Programming Lab**
Instructor(s) :	**Ali Fatmi**

LAB - 9

Nested Classes and Exception Handling

Nested Classes

Java inner class or nested class is a class that is declared inside the class or interface.

It can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Member inner class
- Anonymous inner class
- Local inner class

Java Member Inner class

A class that is created inside a class but outside a method is called member inner class. It is also known as a regular inner class. It can be declared with access modifiers like public, default, private, and protected.

Example:
In this example, we are creating a msg() method in the member inner class that is accessing the private data member of the outer class.

```java
class JavaOuterClass
{
  // private variable of the outer class
  private int value = 30;
  // inner class
  class JavaInnerClass
  {
    // public variable of the inner class
    public int getValue()
    {
      System.out.println("This is the getValue method of the inner class:");
      return value;
    }
  } //inner class end here
  public static void main(String args[])
  {
    //Creating object of outer class
    JavaOuterClass outer = new JavaOuterClass();
    // Creating object of inner class
    JavaOuterClass.JavaInnerClass inner = outer.new JavaInnerClass();
    System.out.println("Value:" + inner.getValue());
  }
}
```

Output:

```
This is the getValue method of the inner class:
Value:30
```

How to instantiate Member Inner class in Java?

An object or instance of a member's inner class always exists within an object of its outer class. The new operator is used to create the object of member inner class with slightly different syntax.The general form of syntax to create an object of the member inner class is as follows:

Syntax:

OuterClassReference.new MemberInnerClassConstructor();

Example:

obj.new Inner();

Here, OuterClassReference is the reference of the outer class followed by a dot which is followed by the new operator.

Java Local inner class

A class i.e., created inside a method, is called local inner class in java. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

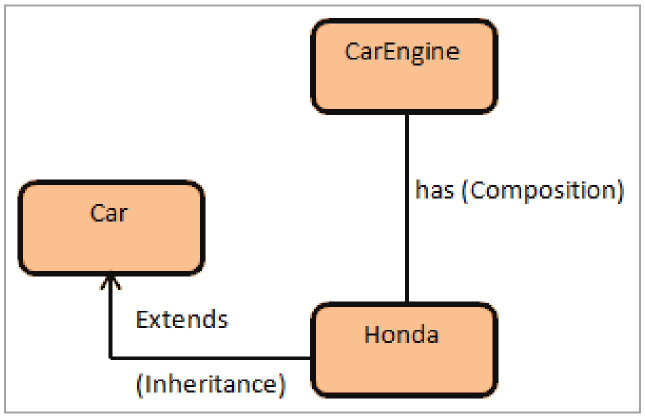
Java local inner class example

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

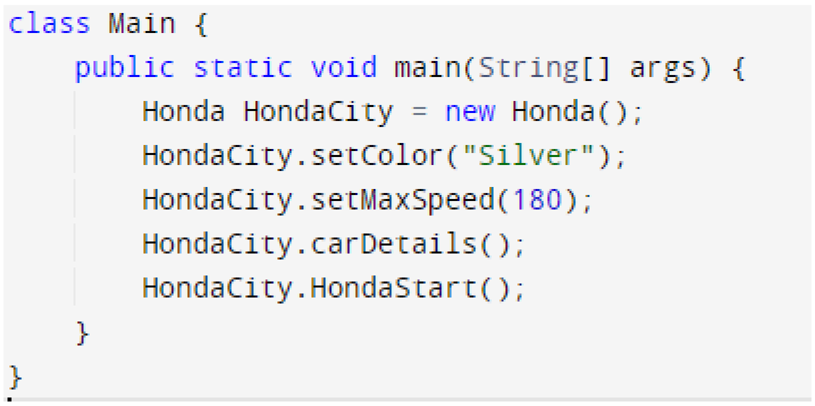
Output:

```
30
```

Containership (Has a Relationship)

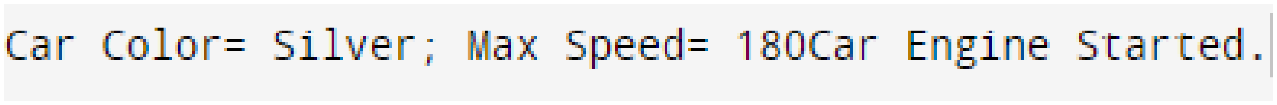


```
class CarEngine {
    public void startEngine(){
        System.out.println("Car Engine Started.");
    }
    public void stopEngine(){
        System.out.println("Car Engine Stopped.");
    }
}

class Car {
    private String color;
    private int max_Speed;
    public void carDetails(){
        System.out.println("Car Color= "+color + "; Max Speed= " + max_Speed);
    }
    //set car color
    public void setColor(String color) {
        this.color = color;
    }
    //set car max_Speed
    public void setMaxSpeed(int max_Speed) {
        this.max_Speed = max_Speed;
    }
}

class Honda extends Car{
    public void HondaStart(){
        CarEngine Honda_Engine = new CarEngine();   //composition
        Honda_Engine.startEngine();
        }
}
```



```java
class Main {
    public static void main(String[] args) {
        Honda HondaCity = new Honda();
        HondaCity.setColor("Silver");
        HondaCity.setMaxSpeed(180);
        HondaCity.carDetails();
        HondaCity.HondaStart();
    }
}
```

Output:



```
Car Color= Silver; Max Speed= 180Car Engine Started.
```

Exception Handling

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

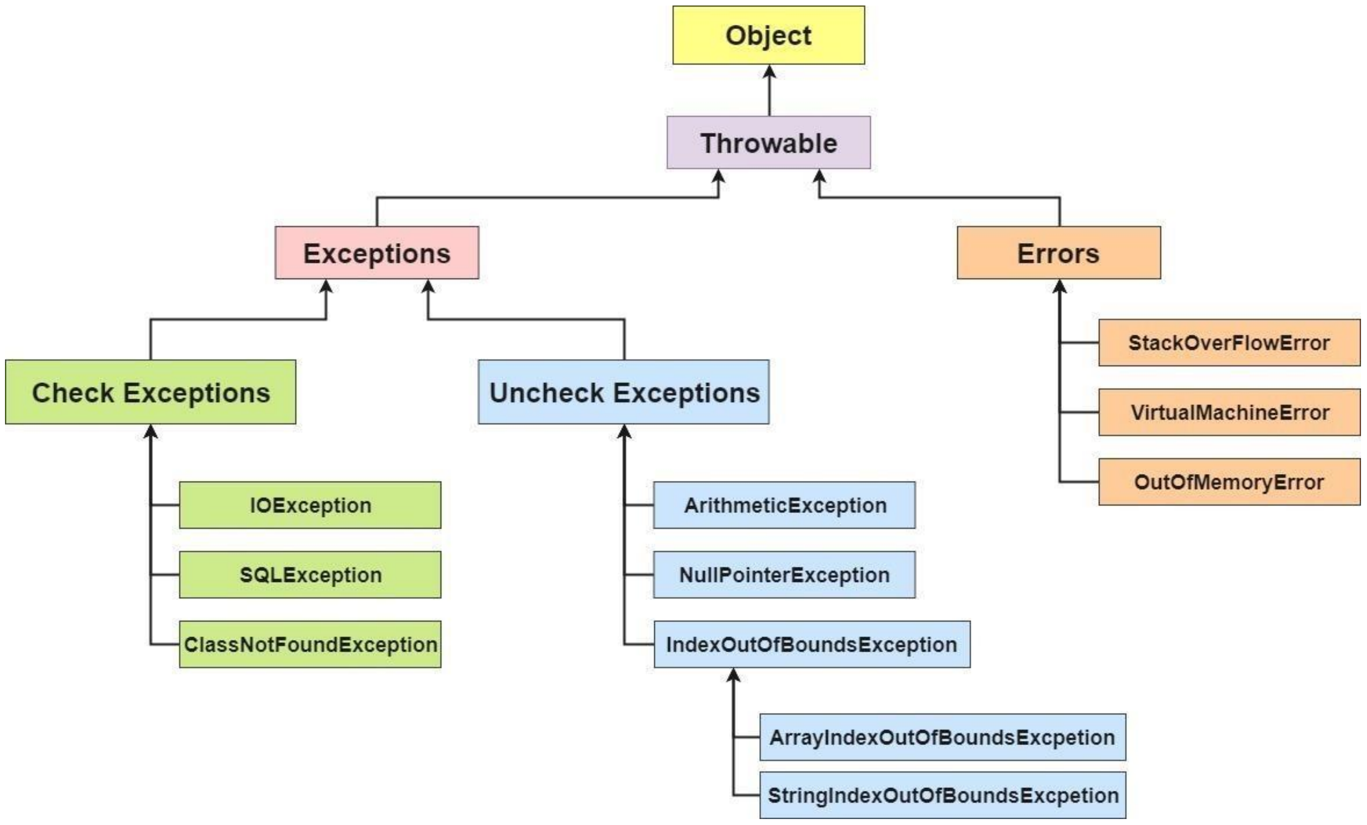
statement 8;

statement 9;

statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed.

Hierarchy of Java Exception classes

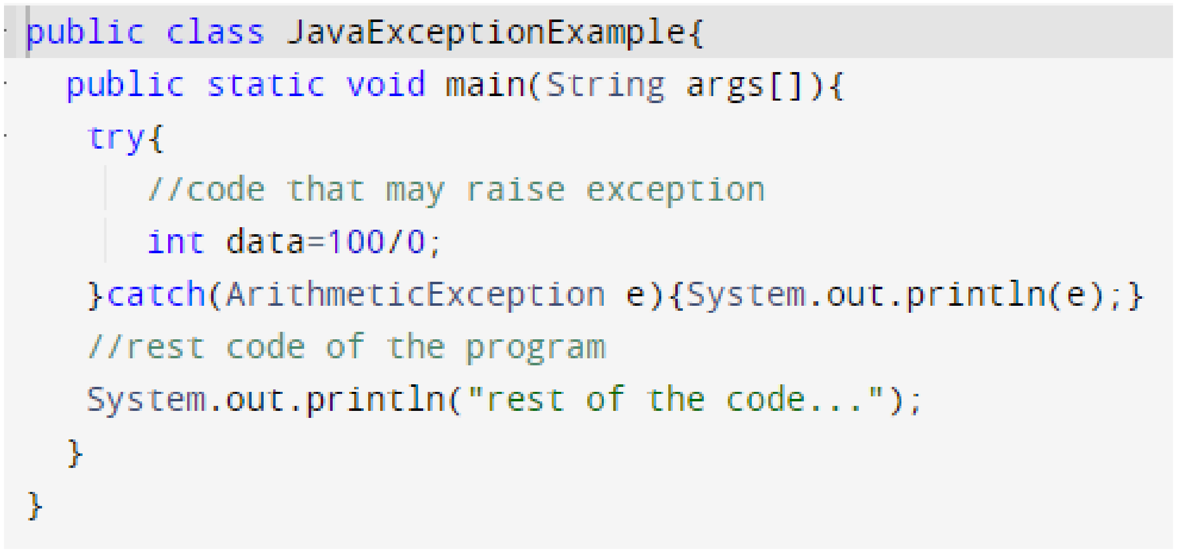


Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

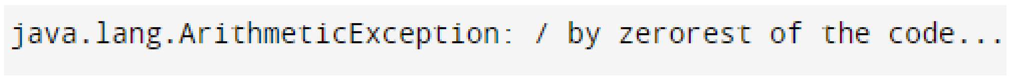
Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Simple Try Catch Example:



```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:



```
java.lang.ArithmeticException: / by zerorest of the code...
```

<u>Common Scenarios of Java Exceptions</u>

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

int a=50/0;//ArithmeticException

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

String s=null;

System.out.println(s.length());//NullPointerException

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

String s="abc";

int i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

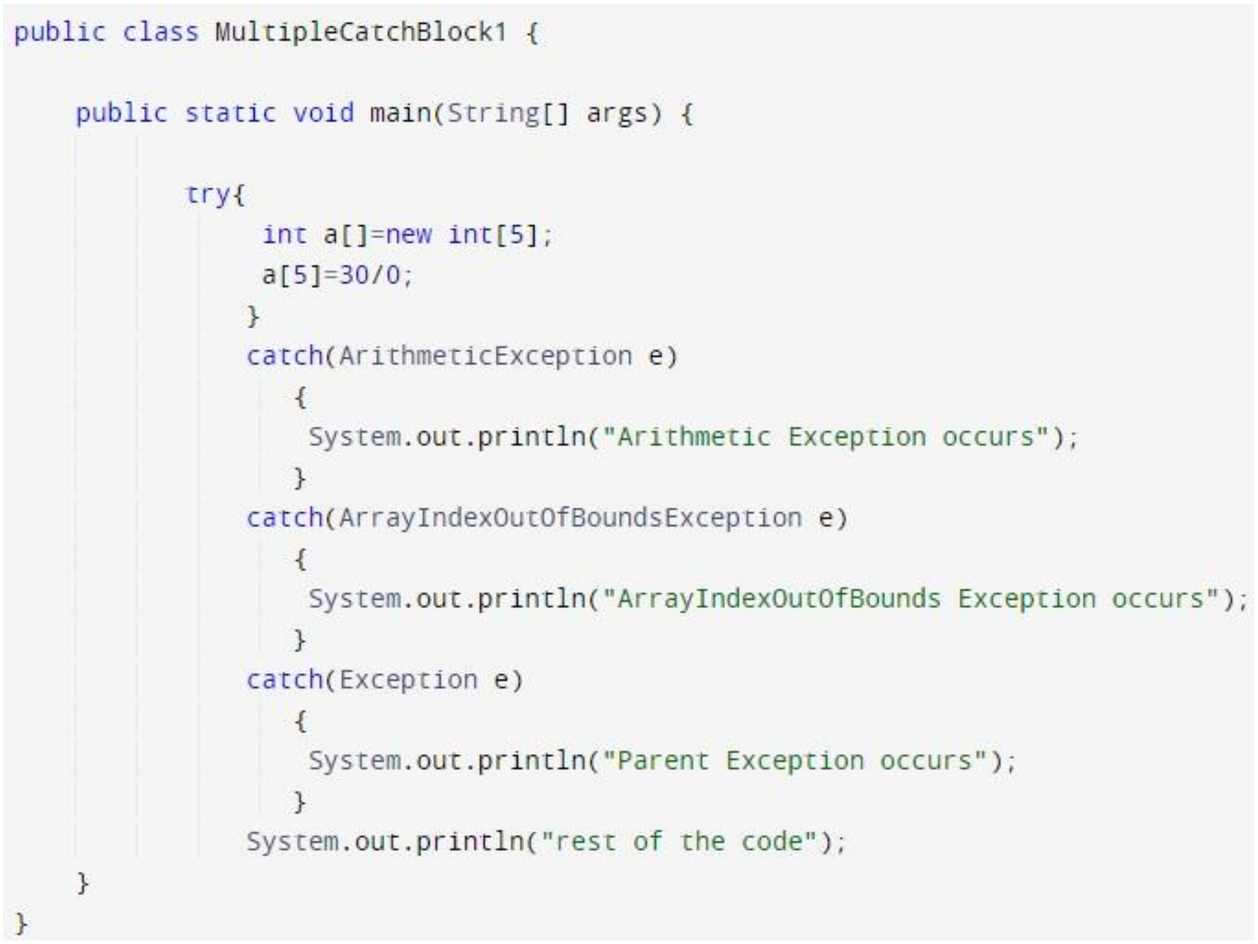
Multi - Catch:

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember:

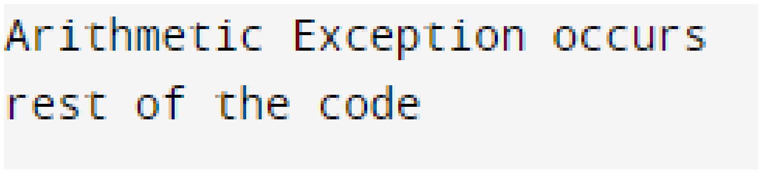
- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception or a compile – time error occurs.

Multi - Catch Example:



```
public class MultipleCatchBlock1 {

    public static void main(String[] args) {

            try{
                 int a[]=new int[5];
                 a[5]=30/0;
                }
                catch(ArithmeticException e)
                   {
                    System.out.println("Arithmetic Exception occurs");
                   }
                catch(ArrayIndexOutOfBoundsException e)
                   {
                    System.out.println("ArrayIndexOutOfBounds Exception occurs");
                   }
                catch(Exception e)
                   {
                    System.out.println("Parent Exception occurs");
                   }
                System.out.println("rest of the code");
    }
}
```

Output:



```
Arithmetic Exception occurs
rest of the code
```

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Finally Example: the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and then the program terminates abnormally.

```java
public class TestFinallyBlock1{
      public static void main(String args[]){

      try {

        System.out.println("Inside the try block");

        //below code throws divide by zero exception
       int data=25/0;
       System.out.println(data);
      }
      //cannot handle Arithmetic type exception
      //can only accept Null Pointer type exception
      catch(NullPointerException e){
        System.out.println(e);
      }

      //executes regardless of exception occured or not
      finally {
        System.out.println("finally block is always executed \n");
      }

      System.out.println("rest of the code...");
      }
}
```

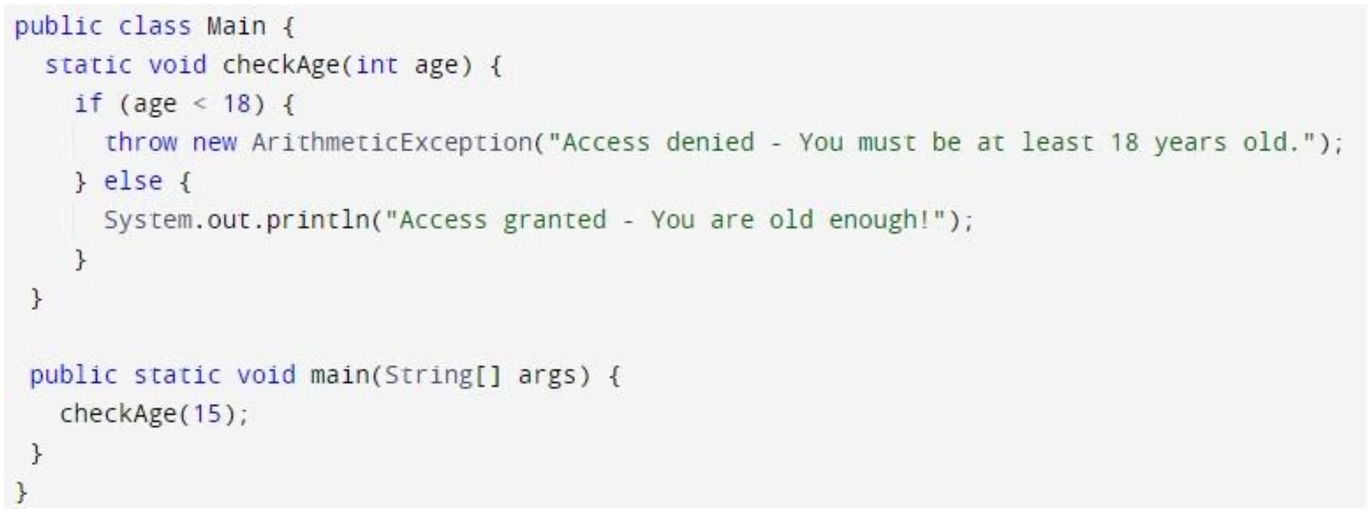
Output:

```
Inside the try block
finally block is always executed

Exception in thread "main" java.lang.ArithmeticException: / by zero
at TestFinallyBlock1.main(TestFinallyBlock1.java:12)
```

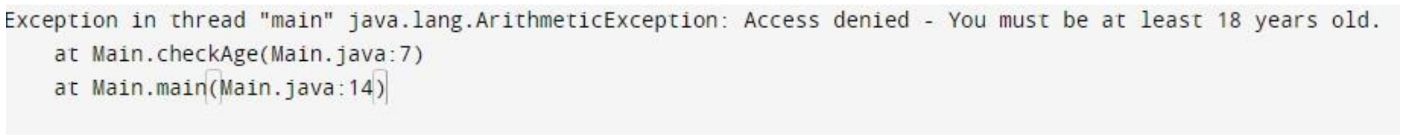
Throw Keyword:

The throw keyword is used to create a custom error.

The throw statement is used together with an exception type. There are many exception types available in Java: ArithmeticException, ClassNotFoundException, ArrayIndexOutOfBoundsException, SecurityException, etc.



```java
public class Main {
  static void checkAge(int age) {
    if (age < 18) {
      throw new ArithmeticException("Access denied - You must be at least 18 years old.");
    } else {
      System.out.println("Access granted - You are old enough!");
    }
  }

  public static void main(String[] args) {
    checkAge(15);
  }
}
```

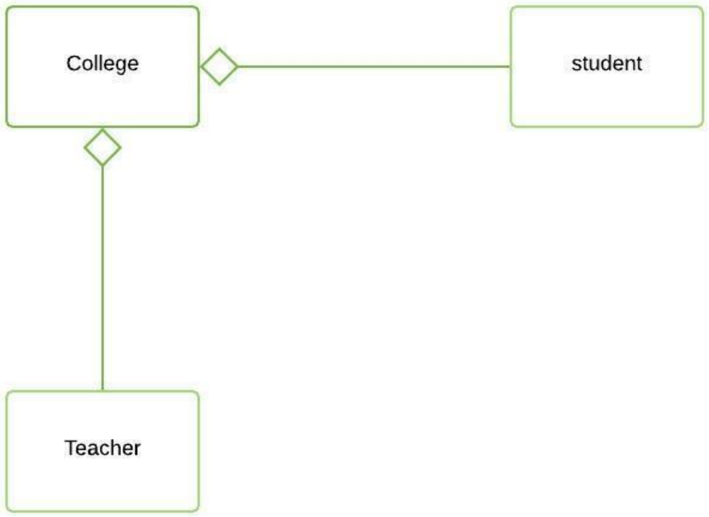


```
Exception in thread "main" java.lang.ArithmeticException: Access denied - You must be at least 18 years old.
    at Main.checkAge(Main.java:7)
    at Main.main(Main.java:14)
```

Aggregation

It is a special form of Association where:

- It represents Has-A’s relationship.
- It is a **unidirectional association** i.e. a one-way relationship. For example, a department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both entries can survive individually** which means ending one entity will not affect the other entity.



```java
// Java program to illustrate
// Concept of Aggregation

// Importing required classes
import java.io.*;
import java.util.*;

// Class 1
// Student class
class Student {

    // Attributes of Student
    private String studentName;
    private int studentId;

    // Constructor of Student class
    public Student(String studentName, int studentId)
    {
        this.studentName = studentName;
        this.studentId = studentId;
    }

    public int getstudentId() {
      return studentId;
    }

    public String getstudentName() {
      return studentName;
    }
}

// Class 2
// Department class
// Department class contains list of Students
```

```java
class Department {

    // Attributes of Department class
    private String deptName;
    private List<Student> students;

    // Constructor of Department class
    public Department(String deptName, List<Student> students)
    {
        this.deptName = deptName;
        this.students = students;
    }

    public List<Student> getStudents() {
      return students;
    }

    public void addStudent(Student student)
    {
        students.add(student);
    }
}

// Class 3
// Institute class
// Institute class contains the list of Departments
class Institute {

    // Attributes of Institute
    private String instituteName;
    private List<Department> departments;

    // Constructor of Institute class
    public Institute(String instituteName,
                     List<Department> departments)
    {
        // This keyword refers to current instance itself
        this.instituteName = instituteName;
        this.departments = departments;
    }

    public void addDepartment(Department department)
    {
        departments.add(department);
    }

    // Method of Institute class
    // Counting total students in the institute
    public int getTotalStudentsInInstitute()
    {
        int noOfStudents = 0;
        List<Student> students = null;
        for (Department dept : departments) {
            students = dept.getStudents();

            for (Student s : students) {
                noOfStudents++;
            }
        }
        return noOfStudents;
    }
}

// Class 4
// main class
class AggregationExample {

    // main driver method
    public static void main(String[] args)
    {
        // Creating independent Student objects
        Student s1 = new Student("Parul", 1);
        Student s2 = new Student("Sachin", 2);
        Student s3 = new Student("Priya", 1);
        Student s4 = new Student("Rahul", 2);

        // Creating an list of CSE Students
        List<Student> cse_students = new ArrayList<Student>();
        cse_students.add(s1);
        cse_students.add(s2);

        // Creating an initial list of EE Students
        List<Student> ee_students = new ArrayList<Student>();
        ee_students.add(s3);
        ee_students.add(s4);

        // Creating Department object with a Students list
        // using Aggregation (Department "has" students)
        Department CSE = new Department("CSE", cse_students);
        Department EE = new Department("EE", ee_students);

        // Creating an initial list of Departments
        List<Department> departments = new ArrayList<Department>();
        departments.add(CSE);
        departments.add(EE);

        // Creating an Institute object with Departments list
        // using Aggregation (Institute "has" Departments)
        Institute institute = new Institute("BITS", departments);

        // Display message for better readability
        System.out.print("Total students in institute: ");

        // Calling method to get total number of students
        // in the institute and printing on console
        System.out.print(
            institute.getTotalStudentsInInstitute());
    }
}
```

Output

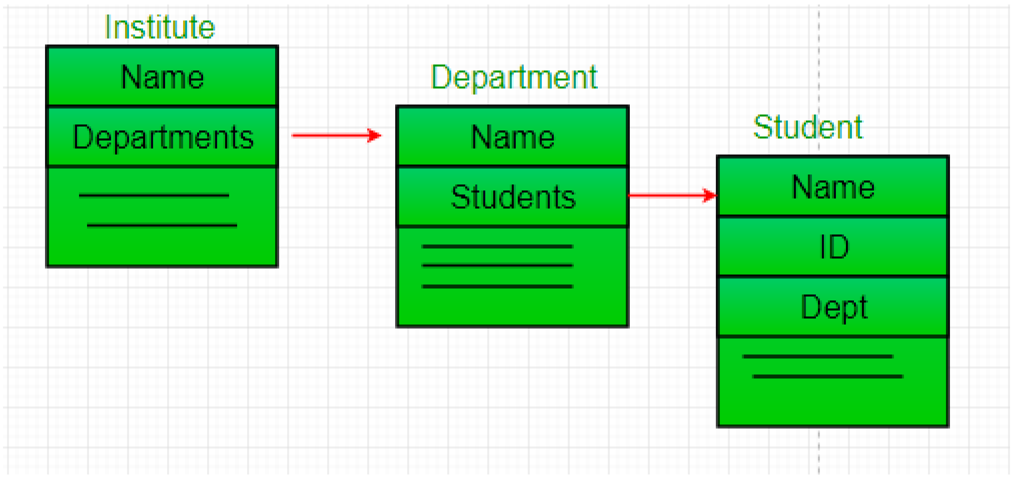
```
Total students in institute: 4
```

Explanation of the above Program:

In this example,

- There is an Institute which has no. of departments like CSE, EE. Every department has no. of students.
- So, we make an Institute class that has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class.
- That means Institute class is associated with Department class through its Object(s).
- And Department class has also a reference to Object or Objects (i.e. List of Objects) of the Student class means it is associated with the Student class through its Object(s).

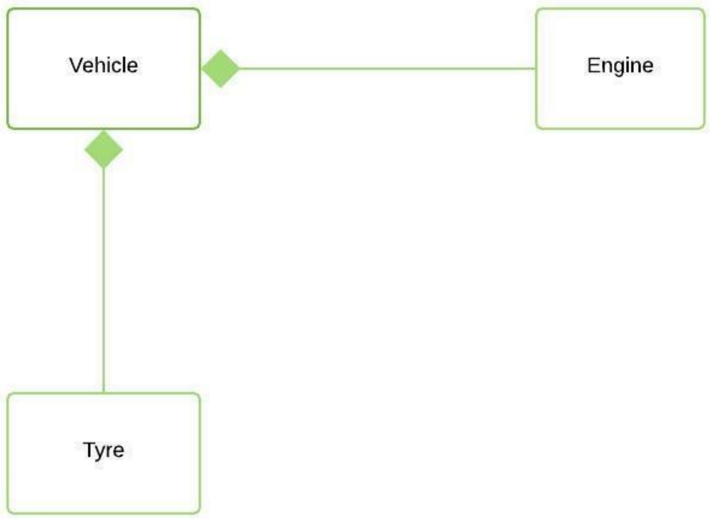
It represents a **Has-A** relationship. In the above example: Student **Has-A** name. Student **Has-A** ID. Department **Has-A** Students as depicted from the below media.



Composition

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

- It represents **part-of** relationship.
- In composition, both entities are dependent on each other.
- When there is a composition between two entities, the composed object **cannot exist** without the other entity.



```
// Java program to illustrate
// Concept of Composition

// Importing required classes
import java.io.*;
import java.util.*;

// Class 1
// Department class
class Department {

    // Attributes of Department
    public String departmentName;

    // Constructor of Department class
    public Department(String departmentName)
    {
        this.departmentName = departmentName;
    }

    public String getDepartmentName()
    {
        return departmentName;
    }
}

// Class 2
// Company class
class Company {

    // Reference to refer to list of books
    private String companyName;
    private List<Department> departments;

    // Constructor of Company class
    public Company(String companyName)
```

```java
    {
        this.companyName = companyName;
        this.departments = new ArrayList<Department>();
    }

    // Method
    // to add new Department to the Company
    public void addDepartment(Department department)
    {
        departments.add(department);
    }

    public List<Department> getDepartments()
    {
        return new ArrayList<>(departments);
    }

    // Method
    // to get total number of Departments in the Company
    public int getTotalDepartments()
    {
        return departments.size();
    }
}

// Class 3
// Main class
class CompositonExample {

    // Main driver method
    public static void main(String[] args)
    {
        // Creating a Company object
        Company techCompany = new Company("Tech Corp");

        techCompany.addDepartment(new Department("Engineering"));
        techCompany.addDepartment(new Department("Operations"));
        techCompany.addDepartment(new Department("Human Resources"));
        techCompany.addDepartment(new Department("Finance"));

        int d = techCompany.getTotalDepartments();
        System.out.println("Total Departments: " + d);

        System.out.println("Department names: ");
        for (Department dept : techCompany.getDepartments()) {
            System.out.println("- " + dept.getDepartmentName());
        }
    }
}
```

Output

```
Total Departments: 4
```

```
Department names:

- Engineering

- Operations

- Human Resources

- Finance
```

Explanation of the above Program:

In the above example,

- A **company** can have no. of **departments**.
- All the departments are part-of the Company.
- So, if the Company gets destroyed then all the Departments within that particular Company will be destroyed, i.e. Departments can not exist independently without the Company.
- That’s why it is composition. Department is **Part-of** Company.

Lab Tasks

1. - Create a class named as “Job” that has role, ID and salary as private attributes.
   - Make get and set for all the attributes.
   - Create a class named as “Person” that has Job object as a member.
   - Make a constructor that initializes the Job object and call the set salary function and set the value of your choice. Invoke the get salary function also using the job object.
   - In the main program, create a Person object and display the salary.

2. - Create a class named as “CPU” that has an attribute double price.
   - Create a nested class “Processor” that has attributes double cores and String manufacturer. The class has a method double getCache( ) that returns 4.2.
   - Create another nested protected class “RAM” that has attributes double memory and String manufacturer. The class has a method double getClockSpeed ( ) that returns 5.3.
   - In the main program, create objects of the outer class as well as both the inner classes. Call both the functions.

3. – Create a class named as “Car” that has attributes carname and cartype. Make a parameterized constructor to set these attributes. Make a private method getCarname( ) that returns car name.
   - Create a class named as “Engine” that has an attribute engine type.
   - Make a set engine function that first checks if the car type is equal “4T”. If the condition matches, it checks if the car name is equal “Mehran” and set the engine type to small or else set the engine type to large. If not set the engine type to “Bigger”.
   - The class has a method getEngineType that returns engine type.
   - In the main program, create objects of the outer class as well as for the inner class. Call the functions as appropriate.

4. Write a Java program, that has an array of size (your choice). Access an element greater than the size of the array and handle the exception using exception handling.

5. Write a Java program that takes has an array to store the marks of a student for five subjects. Input each subject marks from the user and the total marks too. Add all the marks and divide by the total marks. Use exception handling with multiple catch blocks.

6. Write a Java program that takes as input two numbers. Divide one number by another. Handle any exceptions that can occur using finally block as well.