

---

# Object-Oriented Programming

---

## Design Pattern

# Objectives

- Describe polymorphism
- Describe the procedure to override methods of the Object class
- Explain design patterns
- Describe the Singleton, Data Access Object (DAO), and Factory and Observer design patterns
- Describe delegation
- Explain composition and aggregation

# Introduction

- The concept of polymorphism can be applied to Java.
- In Java, a subclass can have its own unique behavior even while sharing certain common functionalities with the parent class.

# Implementing Polymorphism [1-4]

- A super class named `Car` has been created having two methods, `accelerate()` and `printDescription()`.
- A sub class named `LuxuryCar` is created based on `Car` and overrides the methods of the `Car` class.

## Code Snippet

```
class LuxuryCar extends Car {  
    // LuxuryCar defines an additional feature named perks  
    public String perks;  
    public LuxuryCar(int mileage, String color, String make, String perks)  
    {  
        super(mileage, color, make);  
        this.perks = perks;  
    }  
}
```

# Implementing Polymorphism [2-4]

```
public void accelerate() {  
    System.out.println("Luxury Car is Accelerating");  
}  
public void printDescription() {  
    super.printDescription();  
    System.out.println("The " + "Luxury car is a: " + this.perks +  
        ".");  
}  
}
```

# Implementing Polymorphism [3-4]

Code Snippet 2 creates two instances of type `Car`, instantiates them, and invokes the `accelerate()` and `printDescription()` methods on each instance respectively.

## Code Snippet

```
public class PolymorphismTest {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        Car objCar, objLuxuryCar;  
        objCar = new Car(80, "Red", "BMW");  
        objLuxuryCar = new LuxuryCar(120, "Yellow", "Ferrari", "Sports  
        Car");  
    }  
}
```

# Implementing Polymorphism [4-4]

```
objCar.accelerate();  
objCar.printDescription();  
//System.out.println("Now inside LuxuryCar");  
objLuxuryCar.accelerate();  
objLuxuryCar.printDescription();  
}  
}
```

# Overriding the Methods of Object

## Class [1 1 2]

■ Object is the root class.

■ Its methods can be overridden by any class (unless the methods are marked as final).

■ Following are the methods that can be overridden with a different functionality as compared to the root class:

□ `public boolean equals(Object obj)`

□ `public int hashCode()`

□ `public String toString()`

◆ The `equals()` method compares two objects to determine if they are equal.

◆ There are two different types of equality. They are:

□ Reference equality

□ Logical equality

◆ Reference equality is when the physical memory locations of the two strings are same.

◆ Logical equality is when data in the objects are the same.



# Overriding the Methods of Object

Code Snippet

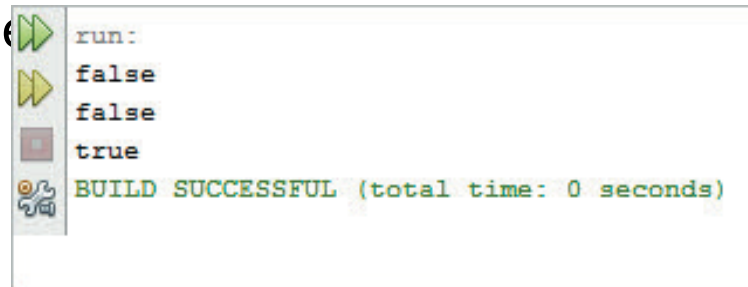
Class [2-12]

```
public class EqualityTest {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        String strAObj = new String("JOHN");  
        String strBObj = new String("JOHN");  
        String strCObj = new String("ANNA");  
        String strEObj = strAObj;  
        System.out.println(strAObj == strBObj);  
        System.out.println(strAObj == strCObj);  
        System.out.println(strAObj == strEObj);  
    }  
}
```

# Overriding the Methods of Object

## Class [3-12]

- The equality operator (==) compares the memory addresses of the two strings.
- When `strAObj` is compared to `strBObj`, the result is false, although their value is same, which is JOHN.
- A comparison between `strAObj` and `strCObj` returns false because the references of the two different String objects are different addresses.
- When `strAObj` is compared to `strEObj`, the result is true because they point to the same memory location.
- The Code Snippet



```
run:
false
false
true
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Overriding the Methods of Object

Code Snippet

Class [1-12]

```
public class LogicalEqualityTest {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // TODO code application logic here  
        String strAObj = new String("JOHN");  
        String strBObj = new String("JOHN");  
        String strCObj = new String("ANNA");  
        // Create a String reference and assign an existing  
        // String's reference  
        // to it so that both references point to the same  
        // String object in memory.  
        String strEObj = strAObj;
```

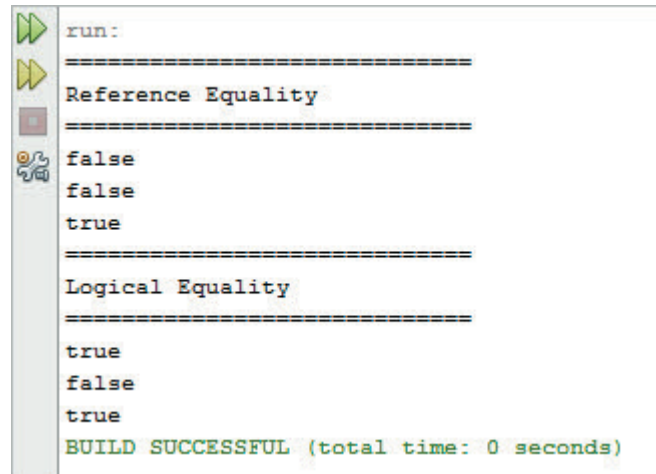
# Overriding the Methods of Object

```
// Print the results of the equality checks
System.out.println("=====");
System.out.println("Logical or Value Equality");
System.out.println("=====");
//Tests logical or value equality
System.out.println(strAObj.equals(strBObj));
System.out.println(strAObj.equals(strCObj));
System.out.println(strAObj.equals(strEObj));
}
}
```

# Overriding the Methods of Object

## Class [6-12]

- The `equals()` method is used to check for logical equality.
- Note that the `equals()` method is implicitly inherited from the `Object` class.
- The `String` class overrides the `equals()` method and compares the two `String` objects character by character.
- The Code Snippet displays the following output:



```
run:
=====
Reference Equality
=====
false
false
true
=====
Logical Equality
=====
true
false
true
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Overriding the Methods of Object

- The `hashCode()` method of `Object` class returns the object's memory address in hexadecimal format.
- It is used along with the `equals()` method in hash-based collections such as `Hashtable`.
- If two objects are equal, their hash code should also be equal.

## Code Snippet

```
public class Student
{
    private int ID;
    public int getID()
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter values");
        int ID = Integer.parseInt(sc.nextLine());
        return ID;
    }
}
```

# Overriding the Methods of Object

```
public boolean equals(Object obj)
{
    if (getID() == ((Student)obj).getID())
        return true;
    else
        return false;
}
public int hashCode()
{
    return ID;
}
/**
 * @param args the command line arguments
 */
```

# Overriding the Methods of Object

```
public static void main(String[] args)
{
    Student s1 = new Student();
    Student s2 = new Student();
    if (s1.equals(s2))
        System.out.println("The two ID values are equal");
    else
        System.out.println("The two ID values are not equal");
}
```



# Overriding the Methods of Object

## Class [10-12]

- The `Student` class defines three methods in addition to `main()` method.
- The `getID()` method accepts an ID value from the standard input.
- The overridden `equals()` method accepts a parameter of type `Object`.
- It compares the ID value of this object with the ID value of the current object.
- The overridden `hashCode()` method returns the ID value.
- In the `main()` method, two objects of `Student` class, namely, `s1` and `s2` are created.
- The `equals()` method is invoked on `s1` and the object `s2` is passed as a parameter.
- Depending on the user input, the output will be displayed accordingly.

# Overriding the Methods of Object

## Class [11-12]

- The `toString()` method of `Object` class returns a string representation of the object.
- It is typically used for debugging.

### Code Snippet

```
public class Exponent {  
    private double num, exp;  
    public Exponent(double num, double exp) {  
        this.num = num;  
        this.exp = exp;  
    }  
    /* Returns the string representation of this number.  
    The format of string is "Number + e Value" where Number is the  
    number value and e Value is the exponent part.
```

# Overriding the Methods of Object

```
@Override
public String toString() {
    return String.format(num + "E+" + exp);
}
/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    Exponent c1 = new Exponent(10, 15);
    System.out.println(c1);
}
}
```

# The instanceof Operator [1-5]

- The `instanceof` operator is used to compare an object to a specified type such as instance of a class and an instance of a

## Code Snippet

```
class Employee {  
    int empcode;  
    String name;  
    String dept;  
    int bonus;  
}  
class Manager extends Employee {  
    String name;  
    int mgrid;  
}
```

# The instanceof Operator [2-5]

```
public class Square {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        Employee emp1 = new Employee();  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter values");  
        emp1.name = sc.nextLine();  
        Employee m1 = new Manager();  
        m1.name = sc.nextLine();  
        if (emp1 instanceof Employee) {  
            emp1.bonus = 7000;  
            System.out.println(emp1.name + " is an employee and has bonus  
            "+emp1.bonus);  
        }  
    }  
}
```

# The instanceof Operator [3-5]

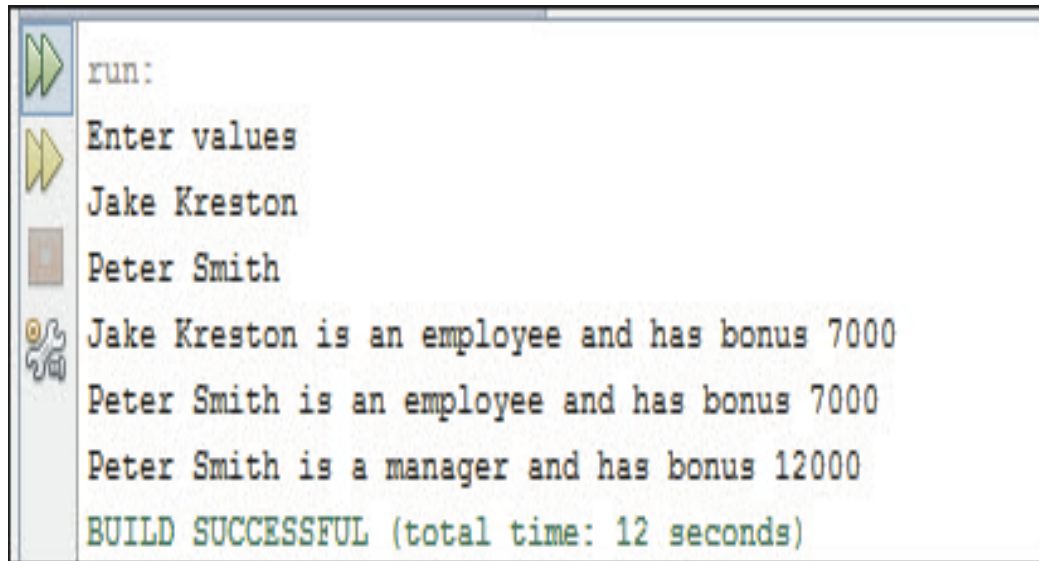
```
if (emp1 instanceof Manager) {  
    emp1.bonus = 12000;  
    System.out.println(emp1.name + " is a manager and has bonus  
    "+emp1.bonus);  
}  
if (m1 instanceof Employee) {  
    m1.bonus = 7000;  
    System.out.println(m1.name + " is an employee and has bonus "+  
    m1.bonus);  
}
```

# The instanceof Operator [4-5]

```
if (m1 instanceof Manager) {  
    m1.bonus = 12000;  
    System.out.println(m1.name + " is a manager and has bonus " +  
        m1.bonus);  
}  
}  
}
```

# The instanceof Operator [5-5]

- The code defines the following:
  - Parent class, `Employee`
  - Child class, `Manager` that inherits from the parent
- The following displays the output of the code:



```
run:
Enter values
Jake Kreston
Peter Smith
Jake Kreston is an employee and has bonus 7000
Peter Smith is an employee and has bonus 7000
Peter Smith is a manager and has bonus 12000
BUILD SUCCESSFUL (total time: 12 seconds)
```



# Design Patterns [1-10]

- A design pattern is a clearly defined solution to problems that occur frequently.
- Design pattern are based on the fundamental principles of object oriented design.
- Following are the different types of design patterns:
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- Singleton pattern is a type of creational pattern.
- The singleton design pattern provides complete information on such class implementations.

# Design Patterns [2-10]

- Consider the following when implementing the singleton design pattern:
  - ❑ The reference is finalized so that it does not reference a different instance.
  - ❑ The private modifier allows only same class access and restricts attempts to instantiate the singleton class.
  - ❑ The factory method provides greater flexibility. It is commonly used in singleton implementations.
  - ❑ The singleton class usually includes a private constructor that prevents a constructor to instantiate the singleton class.
  - ❑ To avoid using the factory method, a public variable can be used at the time of using a static reference.

# Design Patterns [3-10]

## Code Snippet

```
class SingletonExample {  
    private static SingletonExample singletonExample = null;  
    private SingletonExample() {  
    }  
    public static SingletonExample getInstance() {  
        if (singletonExample == null) {  
            singletonExample = new SingletonExample();  
        }  
        return singletonExample;  
    }  
    public void display() {  
        System.out.println("Welcome to Singleton Design  
        Pattern");  
    }  
}
```

# Design Patterns [4-10]

- The `SingletonExample` class contains a private static `SingletonExample` field.
- There is a private constructor.
- The public static `getInstance()` method returns the only `SingletonExample` instance.
- There is a public `sayHello()` method that can test the singleton.

# Design Patterns [5-10]

## Code Snippet

```
public class SingletonTest {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        SingletonExample singletonExample =  
        SingletonExample.getInstance();  
        singletonExample.display();  
    }  
}
```

- The `display()` method is called on the singleton class.
- The output of the program is "Welcome to Singleton Design Pattern".

# Design Patterns [6-10]

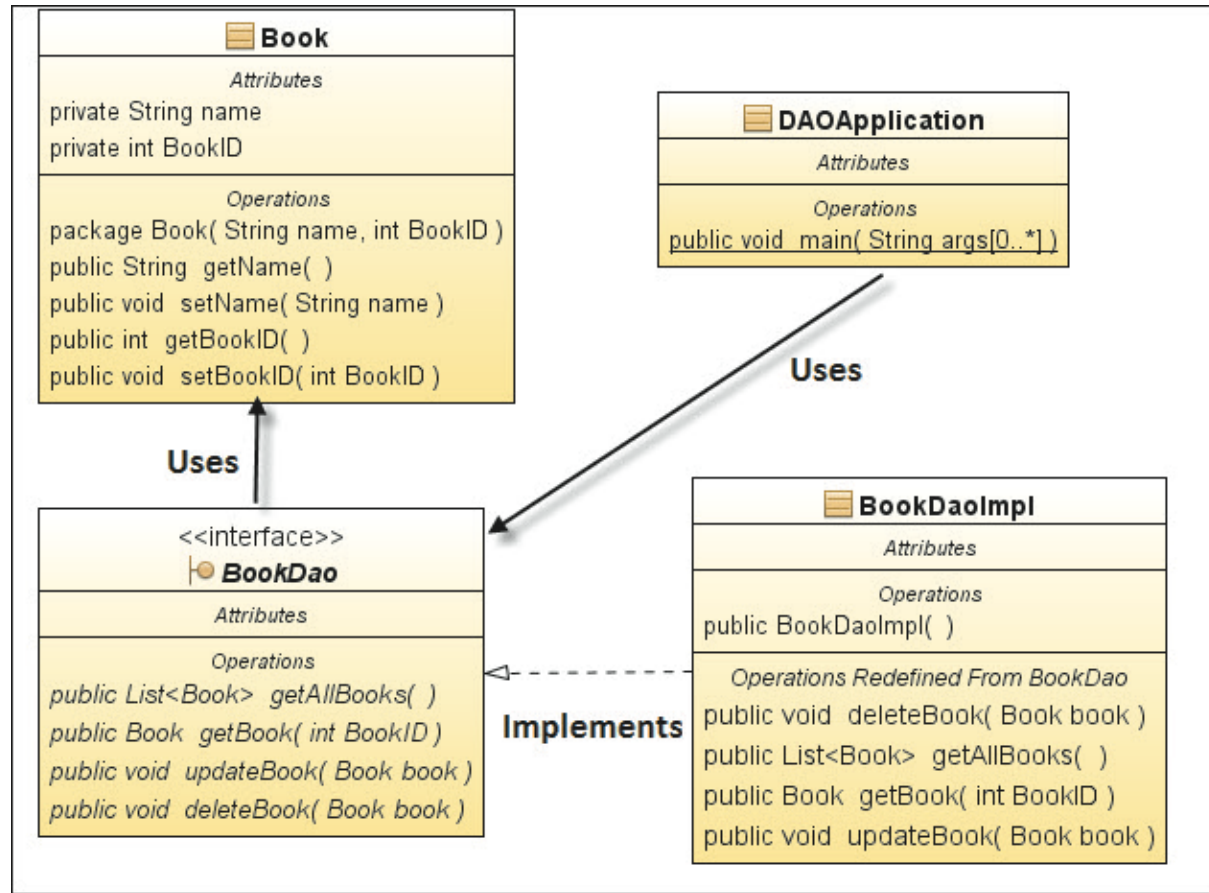
- In Java, interfaces include constant fields.
- They can be used as reference types.
- They are important components of many design patterns.
- An interface declaration includes the following:
  - Modifiers
  - The keyword interface
  - The interface name
  - A comma-separated list of parent interfaces that it can extend
  - The interface body

# Design Patterns [7-10]

- The Data Access Object (DAO) pattern is used when an application is created that needs to persist its data.
- The DAO pattern involves a technique for separating the business logic from persistence logic.
- The DAO pattern uses the following:
  - DAO Interface
  - DAO Concrete Class
  - Model Object or Value Object

# Design Patterns [8-10]

The following figure shows the structure of a DAO design pattern:





# Design Patterns [9-10]

## **Factory Pattern:**

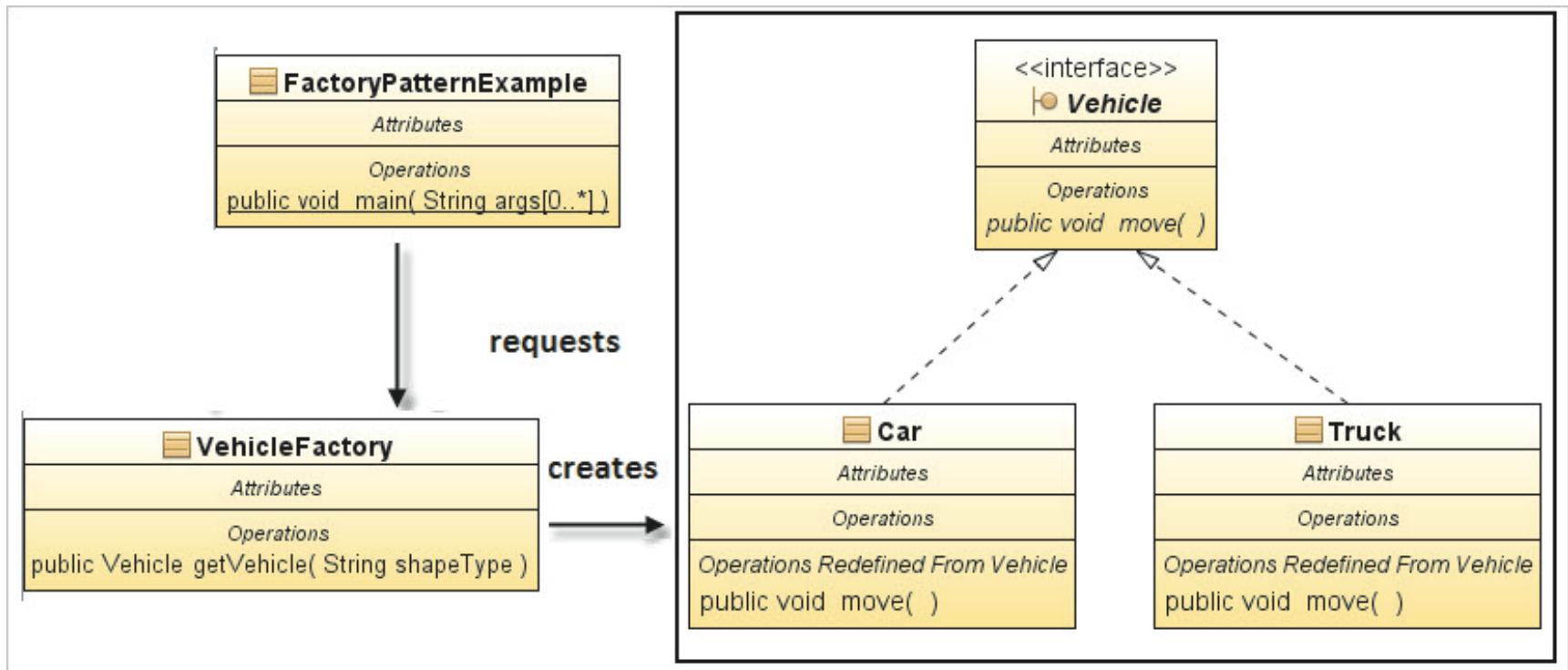
- It is one of the commonly used design patterns in Java.
- It belongs to the creational pattern category.
- This pattern does not perform direct constructor calls when invoking a method.

## **Observer Pattern:**

- This helps to observe the behavior of objects such as change in state or change in property.
- Here, an object called the subject maintains a collection of objects called observers.
- Whenever the subject changes, it notifies the observers.
- Observers can be added or removed from the collection of observers in the subject.

# Design Patterns [10-10]

The following figure shows the factory pattern diagram:



# Delegation

- Delegation is a relationship between objects.
- Here, one object forwards method calls to another object, which is called its delegate.
- Unlike inheritance, delegation does not create a super class.
- Delegation does not force to accept all the methods of the super class.
- Delegation supports code reusability and provides run-time flexibility.

# Composition and Aggregation [1-6]

- Composition refers to the process of composing a class from references to other objects.
- Composition forms the building blocks for data structures.
- Programmers can use object composition to create more complex objects.
- In aggregation, one class owns another class.
- In composition, when the owning object is destroyed, so are the objects within it but in aggregation, this is not true.
- Composition and aggregation are design concepts and not actual patterns.

# Composition and Aggregation [2-6]

## Code Snippet

```
// Composition
class House
{
    // House has door.
    // Door is built when House is built,
    // it is destroyed when House is destroyed.
    private Door dr;
};
```

To implement object composition, perform the following steps:

- ❑ Create a class with reference to other classes.
- ❑ Add the same signature methods that forward to the referenced object.

# Composition and Aggregation [3-6]

- Consider an example of a student attending a course.
  - The student 'has a' course.
  - The composition for the **Student** and **Course** classes is depicted in the following Code Snippet:

## Code Snippet

```
package composition;  
public class Course {  
    private String title;  
    private long score;  
    private int id;  
  
    public String getTitle() {  
        return title;  
    }  
}
```

# Composition and Aggregation [4-6]

```
public void setTitle(String title) {  
    this.title = title;  
}  
public long getScore() {  
    return score;  
}  
public void setScore(long score) {  
    this.score = score;  
}  
public int getId() {  
    return id;  
}  
public void setId(int id) {  
    this.id = id;  
}  
}
```

# Composition and Aggregation [5-6]

```
package composition;
public class Student {
    //composition has-a relationship
    private Course course;
    public Student(){
        this.course=new Course();
        course.setScore(1000);
    }
    public long getScore() {
        return course.getScore();
    }
}
/**
 * @param args the command line arguments
 */
```



# Composition and Aggregation [6-6]

```
public static void main(String[] args)
{
    Student p = new Student();
    System.out.println(p.getScore());
}
```

# Summary

- The development of application software is performed using a programming language that enforces a particular style of programming, also referred to as programming paradigm.
- In structured programming paradigm, the application development is decomposed into a hierarchy of subprograms.
- In object-oriented programming paradigm, applications are designed around data, rather than focusing only on the functionalities.
- The main building blocks of an OOP language are classes and objects. An object represents a real-world entity and a class is a conceptual model.
- Java is an OOP language as well a platform used for developing applications that can be executed on different platforms. Java platform is a software-only platform that runs on top of the other hardware-based platforms.
- The editions of Java platform are Java SE, Java EE, and Java ME.
- The components of Java SE platform are JDK and JRE. JRE provides JVM and Java libraries that are used to run a Java program. JDK includes the necessary development tools, runtime environment, and APIs for creating Java programs.