COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

ADVANCED PROGRAMMING COMP231

Lecturer :Farid Mohammad

Object Oriented Thinking

Class Abstraction and Encapsulation

Class abstraction is the separation of class implementation from the use of a class.

The details of implementation are encapsulated and hidden from the user.
This is known as class encapsulation.

Listing 2.9, ComputeLoan.java, presented a program for computing loan payments.

That program cannot be reused in other programs because the code for computing the payments is in the main method.

One way to fix this problem is to **define static methods** for computing the **monthly payment and total payment.**

However, this solution has limitations.

Suppose you wish to associate a date with the loan. There is no good way to tie a date with a loan without using objects.

The traditional procedural programming paradigm is action-driven, and data are separated from actions.

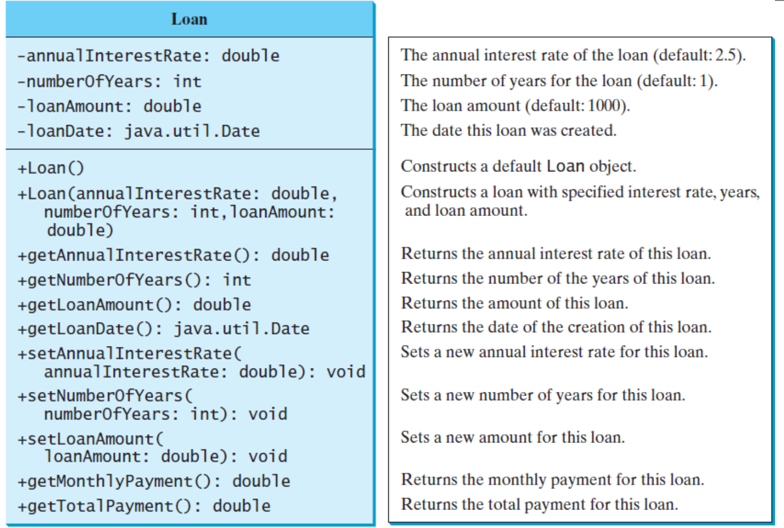
To tie a date with a loan, you can define a **loan class with a date**

LISTING 2.9 ComputeLoan.java

```
1  import java.util.Scanner;                                            import class
2
3  public class ComputeLoan {
4    public static void main(String[] args) {
5      // Create a Scanner
6      Scanner input = new Scanner(System.in);                          create a Scanner
7
8      // Enter annual interest rate in percentage, e.g., 7.25%
9      System.out.print("Enter annual interest rate, e.g., 7.25%: ");
10     double annualInterestRate = input.nextDouble();                  enter interest rate
11
12     // Obtain monthly interest rate
13     double monthlyInterestRate = annualInterestRate / 1200;
14
15     // Enter number of years
16     System.out.print(
17       "Enter number of years as an integer, e.g., 5: ");
18     int numberOfYears = input.nextInt();                             enter years
19
20     // Enter loan amount
21     System.out.print("Enter loan amount, e.g., 120000.95: ");
22     double loanAmount = input.nextDouble();                          enter loan amount
23
```

A loan object now contains data and actions

Figure 10.2 shows the UML class diagram for the Loan class.



Thinking in Objects

The procedural paradigm focuses on designing methods.

The object-oriented paradigm couples data and methods together into objects.

Software design using the object-oriented paradigm focuses on objects and operations on objects.

Chapters 1–8 introduced **fundamental** programming techniques for problem solving using loops, methods, and arrays.

Knowing these techniques lays a solid foundation for object-oriented programming.

Classes **provide** more **flexibility** and **modularity** for building reusable software.

This section improves the solution for a problem introduced in Chapter 3 using the object-oriented approach.

Listing 3.4, ComputeAndInterpretBMI.java, presented a program for computing body mass index.

The code cannot be **reused** in other programs, because the code is in the `main` method.

To make it reusable, **define** a static method to compute body mass index as follows:

```
public static double getBMI(double weight, double height)
```

LISTING 3.4 ComputeAndInterpretBMI.java

```
1 import java.util.Scanner;
2
3 public class ComputeAndInterpretBMI {
4 public static void main(String[] args) {
5 Scanner input = new Scanner(System.in);
6
7 // Prompt the user to enter weight in pounds
8 System.out.print("Enter weight in pounds: ");
9 double weight = input.nextDouble();
10
11 // Prompt the user to enter height in inches
12 System.out.print("Enter height in inches: ");
13 double height = input.nextDouble();
14
15 final double KILOGRAMS_PER_POUND = 0.45359237; //
Constant
16 final double METERS_PER_INCH = 0.0254; // Constant
17
18 // Compute BMI
19 double weightInKilograms = weight *
KILOGRAMS_PER_POUND;
20 double heightInMeters = height * METERS_PER_INCH;
21 double bmi = weightInKilograms /
22 (heightInMeters * heightInMeters);
23
24 // Display result
25 System.out.println("BMI is " + bmi);
26 if (bmi < 18.5)
27 System.out.println("Underweight");
28 else if (bmi < 25)
29 System.out.println("Normal");
30 else if (bmi < 30)
31 System.out.println("Overweight");
32 else
33 System.out.println("Obese");
34 }
35 }
```

This method is useful for computing body mass index for a specified weight and height.
However, it has limitations.

Suppose you need to associate the weight and height with a person's name and birth date.

You could declare separate variables to store these values, but these values would not be tightly coupled.

The **ideal way to couple them is to create an object that contains them all**.

Since these values are tied to **individual** objects, they should be stored in **instance data fields**.

You can define a class named **BMI** as shown in Figure 10.3.

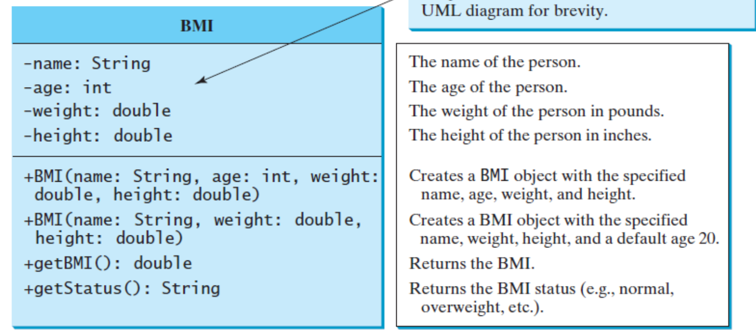


Figure 10.3 The BMI class encapsulates BMI information.

Listing 10.3 UseBMIClass.java

```java
public class UseBMIClass {
public static void main(String[] args) {
BMI bmi1 = new BMI("Kim Yang", 18, 145, 70);
System.out.println("The BMI for " + bmi1.getName() + " is "
+ bmi1.getBMI() + " " + bmi1.getStatus());

BMI bmi2 = new BMI("Susan King", 215, 70);
System.out.println("The BMI for " + bmi2.getName() + " is "
+ bmi2.getBMI() + " " + bmi2.getStatus());
 }
 }
```

```java
public class BMI {
 private String name;
 private int age;
 private double weight; // in pounds
 private double height; // in inches
 public static final double KILOGRAMS_PER_POUND = 0.45359237;
 public static final double METERS_PER_INCH = 0.0254;

 public BMI(String name, int age, double weight, double height) {
  this.name = name;
  this.age = age;
  this.weight = weight;
  this.height = height;
  }

  public BMI(String name, double weight, double height) {
  this(name, 20, weight, height);
  }

  public double getBMI() {
  double bmi = weight * KILOGRAMS_PER_POUND /
  ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
  return Math.round(bmi * 100) / 100.0;
  }

  public String getStatus() {
  double bmi = getBMI();
  if (bmi < 18.5)
  return "Underweight";
  else if (bmi < 25)
  return "Normal";
  else if (bmi < 30)
  return "Overweight";
  else
  return "Obese";
  }

  public String getName() {
  return name;
  }

  public int getAge() {
  return age;
  }

  public double getWeight() {
  return weight;
  }
```

	public double getHeight() {
	return height;
	}
	}

Class Relationships

To design classes, you need to explore the relationships among classes.	**What is Class Relationship**
The common relationships among classes are: ***Association***	
Aggregation	
Composition	
Inheritance	

Association

Association is a general binary relationship that describes an **activity between two classes**.

For example, a student taking a course is an **association**
between
the **Student** class
and the **Course** class,

and a faculty member teaching a course is an association between
the **Faculty**
class
and the **Course** class.

These associations can be represented in UML graphical notation,
as shown in Figure 10.4.

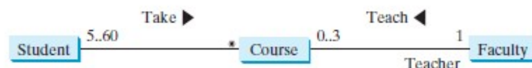


Figure 10.4 This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from five to sixty students, and a course is taught by only one faculty member.

An association is illustrated by
a solid line between two classes **with an optional label** that describes the relationship.

In Figure 10.4, the labels are *Take* and *Teach*
.
Each relationship may have an optional small black **triangle** that indicates the **direction** of the relationship.

In this figure, the direction indicates that a student takes a course (as opposed to a course taking a student).

Each class involved in the relationship may **have a role** name that describes the role it plays in the relationship.

In Figure 10.4, ***teacher* is the role name for Faculty.**

Each class involved in an association may specify a ***multiplicity***, which is placed at the side of the class to specify **how many** of the class's objects are involved in the relationship in UML.

The character * means an unlimited number of objects, and the interval **m..n** indicates that the number of objects is between **m** and **n**, inclusively.

In Figure 10.4, each student may take any number of courses,
and
each course must have at least five and at most sixty students.

Each course is taught by only one faculty member, and a faculty member may **teach from zero to three courses per semester.**

In Java code, you can implement associations by using data fields and methods.

For example,
The relationships in Figure 10.4 may be implemented using the classes in Figure 10.5.

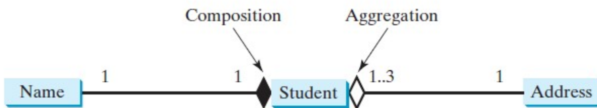
```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

FIGURE 10.5 The association relations are implemented using data fields and methods in classes.

Aggregation and Composition

Aggregation is a **special form of association** that represents an ownership relationship between two objects. Aggregation models ***has-a*** **relationships.** The owner object is called an ***aggregating object***, and its class is called an *aggregating class*. The subject object is called an *aggregated object*, and its class is called an *aggregated class*.	
An object can be owned by several other aggregating objects. If an object is **exclusively owned** by an aggregating object, the relationship between the object and its aggregating object is referred to as a ***composition***. For example, "a student has a name" is a **composition** relationship between the **Student** class and the **Name** class, whereas "a student has an address" is an aggregation relationship between the **Student** class and the **Address** class, since an address can be shared by **several** students. In UML, a **filled** diamond is attached to an aggregating class (in this case, **Student**) to denote the **composition** relationship with an aggregated class (**Name**), and an **empty diamond** is attached to an aggregating class (**Student**) to denote the **aggregation** relationship with an aggregated class (**Address**), as shown in Figure 10.6.	
 FIGURE 10.6 Each student has a name and an address.	
An aggregation relationship is usually represented as a **data field** in the aggregating class. For example, the relationships in Figure 10.6 may be implemented using the classes in Figure 10.7. The relation "a student has a name" and "a student has an address" are implemented in the data field **name** and **address** in the **Student** class.	

```java
public class Name {
  ...
}
```

Aggregated class

```java
public class Student {
  private Name name;
  private Address address;
  ...
}
```

Aggregating class

```java
public class Address {
  ...
}
```

Aggregated class

FIGURE 10.7 The composition relations are implemented using data fields in classes.

Aggregation may exist between objects of the same class.

For example, a **person** may have a **supervisor**. This is illustrated in Figure 10.8.

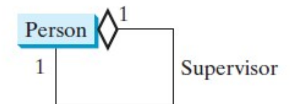


FIGURE 10.8 A person may have a supervisor.

In the relationship "a person has a supervisor," a supervisor can be represented as a data field in the Person class, as follows:

```java
public class Person {
// The type for the data is the class itself
private Person supervisor;
...
}
```

If a person can have **several supervisors**, as shown in Figure 10.9a, you may **use an array to** store supervisors, as shown in Figure 10.9b.

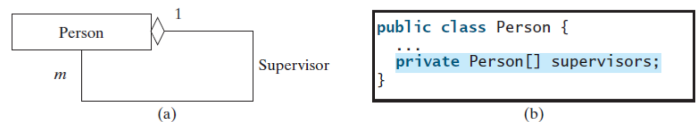


FIGURE 10.9 A person can have several supervisors.

Note

Since aggregation and composition relationships are represented using classes in the same way, we will not differentiate them and call both compositions for simplicity.

aggregation or composition