COMP231

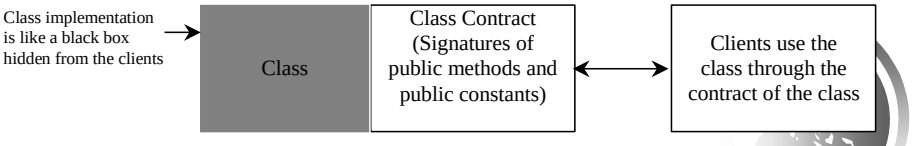
Advanced Programming

Chapter 10 Thinking in Objects

Compiled By: Dr. Majdi Mafarja
Fall Semester 2017/2018

Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



Designing the Loan Class

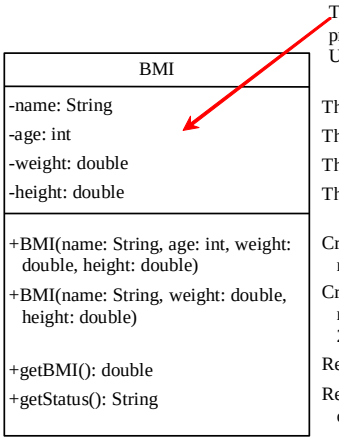
Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1)
-loanAmount: double	The loan amount (default: 1000).
-loanDate: Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years to this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

Loan TestLoanClass Run

Object-Oriented Thinking

Chapters 1-8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays. The studies of these techniques lay 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. From the improvements, you will gain the insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.

The BMI Class



The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The name of the person.

The age of the person.

The weight of the person in pounds.

The height of the person in inches.

Creates a BMI object with the specified name, age, weight, and height.

Creates a BMI object with the specified name, weight, height, and a default age 20.

Returns the BMI

Returns the BMI status (e.g., normal, overweight, etc.)

BMI UseBMIClass Run

Class Relationships

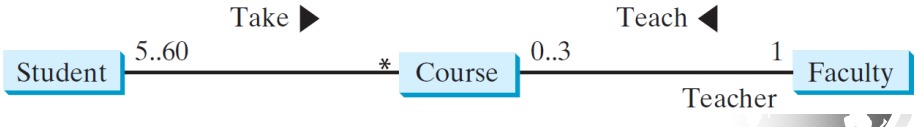
Association

Aggregation

Composition

Inheritance (Chapter 13)

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



```
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) { ... }
}
```

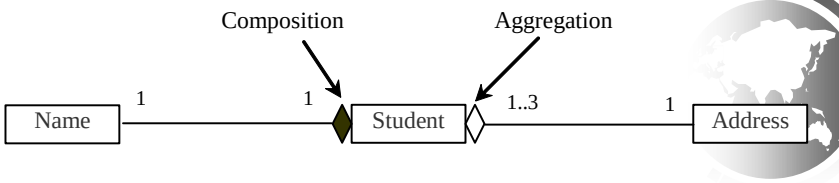
```
public class Faculty {
  private Course[]
    courseList;

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

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

Object Composition

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.



Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure 10.6 can be represented as follows:

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

Aggregated class

```
public class Student {
  private Name name;
  private Address address;

  ...
}
```

Aggregating class

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

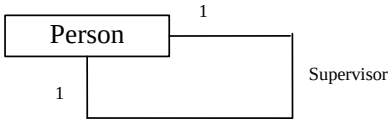
Aggregated class

Aggregation or Composition

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

Aggregation Between Same Class

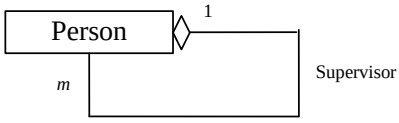
Aggregation may exist between objects of the same class. For example, a person may have a supervisor.




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


Aggregation Between Same Class

What happens if a person has several supervisors?




```
public class Person {
  ...
  private Person[] supervisors;
}
```

Example: The Course Class

Course	
-courseName: String	The name of the course.
-students: String[]	An array to store the students for the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course.
+dropStudent(student: String): void	Drops a student from the course.
+getStudents(): String[]	Returns the students in the course.
+getNumberOfStudents(): int	Returns the number of students in the course.

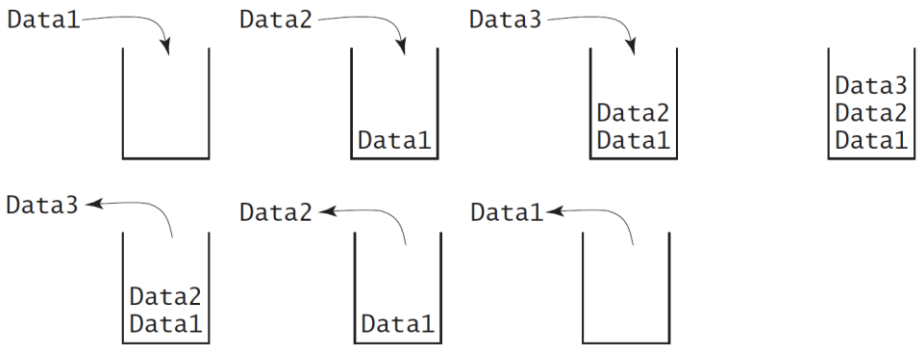
Course TestCourse Run

Example: The `StackOfIntegers` Class

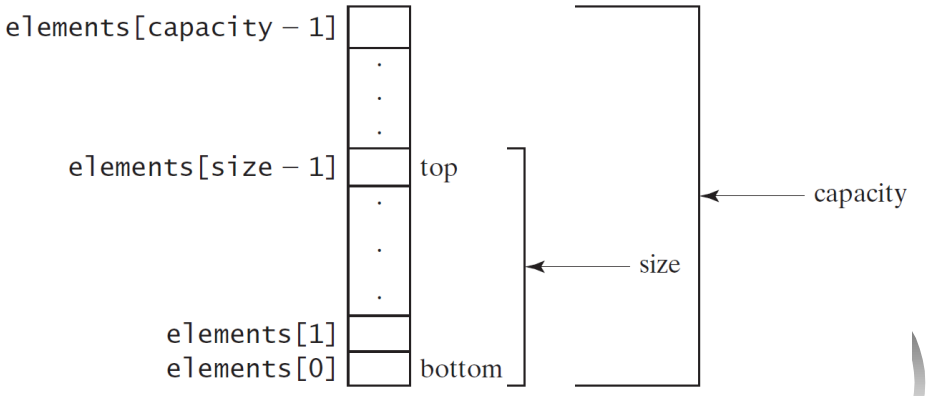
StackOfIntegers	
-elements: int[]	An array to store integers in the stack.
-size: int	The number of integers in the stack.
+StackOfIntegers()	Constructs an empty stack with a default capacity of 16.
+StackOfIntegers(capacity: int)	Constructs an empty stack with a specified capacity.
+empty(): boolean	Returns true if the stack is empty.
+peek(): int	Returns the integer at the top of the stack without removing it from the stack.
+push(value: int): int	Stores an integer into the top of the stack.
+pop(): int	Removes the integer at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

TestStackOfIntegers Run

Designing the StackOfIntegers Class



Implementing `StackOfIntegers` Class



StackOfIntegers

Wrapper Classes

- Boolean
- Character
- Short
- Byte
- Integer
- Long
- Float
- Double

NOTE: (1) The wrapper classes do not have no-arg constructors. (2) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.

The `Integer` and `Double` Classes

java.lang.Integer
-value: int +MAX_VALUE: int +MIN_VALUE: int
+Integer(value: int) +Integer(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longVlaue(): long +floatValue(): float +doubleValue():double +compareTo(o: Integer): int +toString(): String +valueOf(s: String): Integer +valueOf(s: String, radix: int): Integer +parseInt(s: String): int +parseInt(s: String, radix: int): int

java.lang.Double
-value: double +MAX_VALUE: double +MIN_VALUE: double
+Double(value: double) +Double(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longVlaue(): long +floatValue(): float +doubleValue():double +compareTo(o: Double): int +toString(): String +valueOf(s: String): Double +valueOf(s: String, radix: int): Double +parseDouble(s: String): double +parseDouble(s: String, radix: int): double

The `Integer` Class and the `Double` Class

- Constructors
- Class Constants `MAX_VALUE`, `MIN_VALUE`
- Conversion Methods

Numeric Wrapper Class Constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

public Integer(int value)

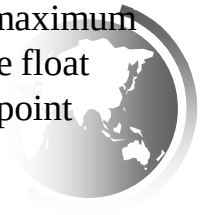
public Integer(String s)

public Double(double value)

public Double(String s)

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants MAX_VALUE and MIN_VALUE. MAX_VALUE represents the maximum value of the corresponding primitive data type. For Byte, Short, Integer, and Long, MIN_VALUE represents the minimum byte, short, int, and long values. For Float and Double, MIN_VALUE represents the minimum *positive* float and double values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45), and the maximum double floating-point number (1.79769313486231570e+308d).



Conversion Methods

Each numeric wrapper class implements the abstract methods doubleValue, floatValue, intValue, longValue, and shortValue, which are defined in the Number class. These methods "convert" objects into primitive type values.

The Static valueOf Methods

The numeric wrapper classes have a useful class method, valueOf(String s). This method creates a new object initialized to the value represented by the specified string. For example:

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

The Methods for Parsing Strings into Numbers

You have used the parseInt method in the Integer class to parse a numeric string into an int value and the parseDouble method in the Double class to parse a numeric string into a double value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):

```
Integer[] intArray = {new Integer(2),
  new Integer(4), new Integer(3)};
```

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

(a) New JDK 1.5 boxing (b)

```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing

BigInteger and BigDecimal

If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package. Both are *immutable*. Both extend the Number class and implement the Comparable interface.

BigInteger and BigDecimal

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

LargeFactorial Run

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```