BIRZEIT UNIVERSITY

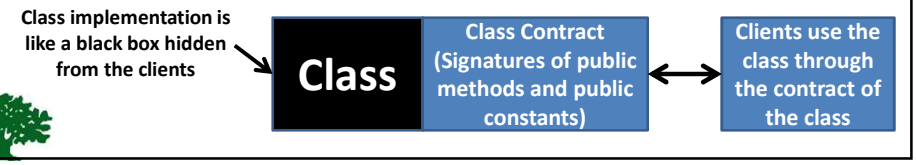
Thinking in Objects

Liang, Introduction to Java programming, 11th Edition, © 2017 Pearson Education, Inc.
All rights reserved

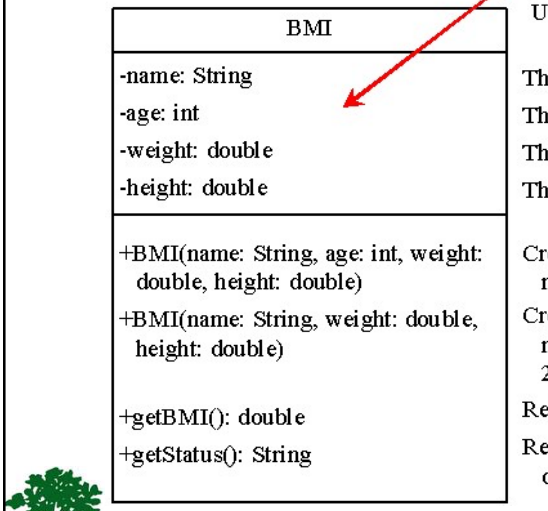
By: Mamoun Nawahdah (PhD)
2020

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.



Case Study 1: 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.)

3

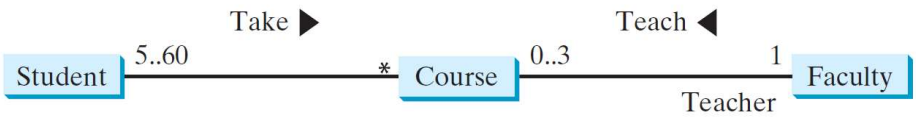
Case Study 2: Loan Class

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.

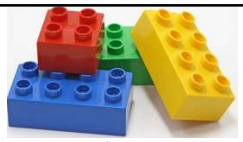
Class Relationships

- Association
- Aggregation
- Composition
- Inheritance (Chapter 11)

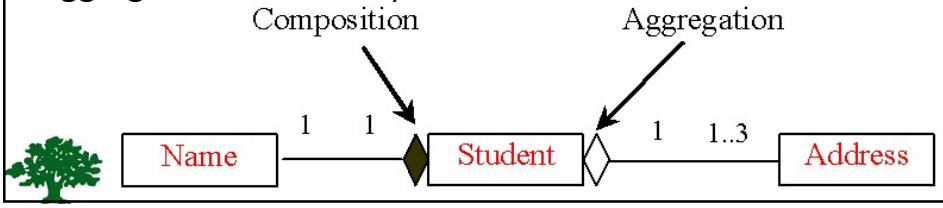
Association: is a general binary relationship that describes an **activity** between two classes.



Aggregation



- **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***.
- **Composition** is actually a special case of the aggregation relationship.



Class Representation

❖ An **aggregation** relationship is usually represented as a data field in the **aggregating** class.

❖ For example, the relationship in the previous figure can be represented as follows:

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

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

  ...
}
```

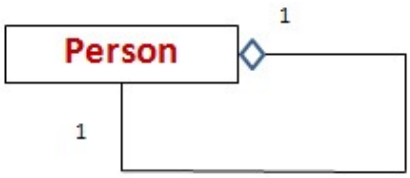
```
public class Address {
  ...
}
```

7

Aggregation Between Same Class

❖ Aggregation may exist between objects of the same class.

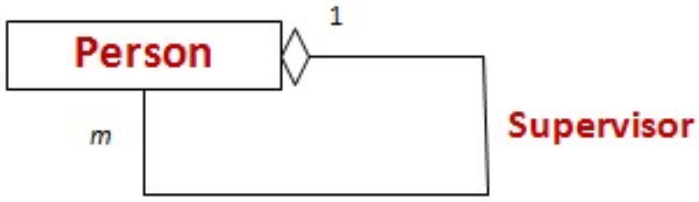
❖ For example, a **person** may have a **supervisor**:




```
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;
  …
}
```

9

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

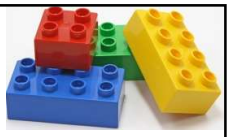
10

Designing a Class

- (**Coherence**) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.
- You can use a class for **students**, for example, but you should not combine **students** and **staff** in the same class, because students and staff have different entities.

11

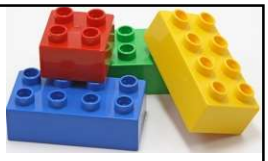
Designing a Class cont.



- (**Separating responsibilities**) A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
- Example: the classes **String**, **StringBuilder**, and **StringBuffer** all deal with strings, for example, but have different responsibilities:
  - **String** class deals with immutable strings.
  - **StringBuilder** class is for creating mutable strings.
  - **StringBuffer** class is similar to **StringBuilder** except that **StringBuffer** contains synchronized methods for updating strings.

12

Designing a Class cont.



❖ Classes are designed for **reuse**.

❖ Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it:

- Design the **properties** to ensure that the user can set properties in any order, with any combination of values.
- Design **methods** to function independently of their order of occurrence.

13

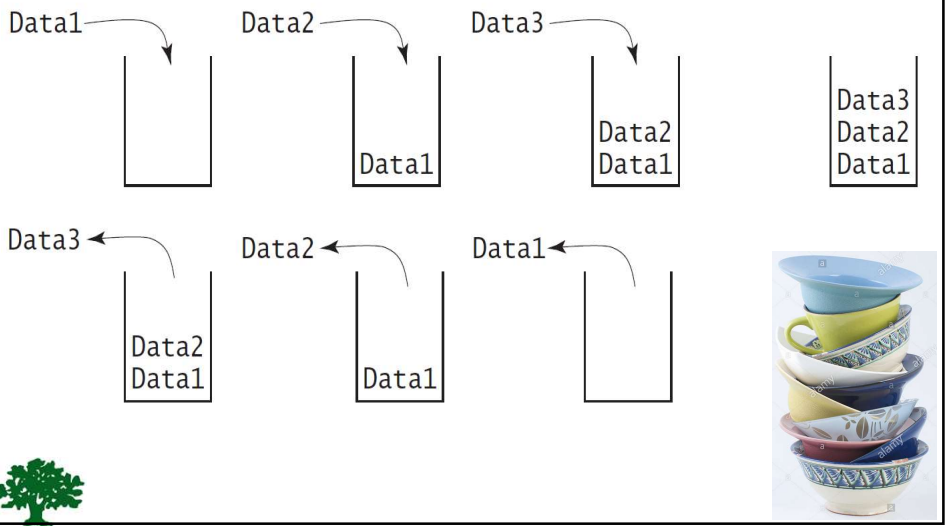
Designing a Class cont.

❖ **Follow standard Java programming style and naming conventions:**

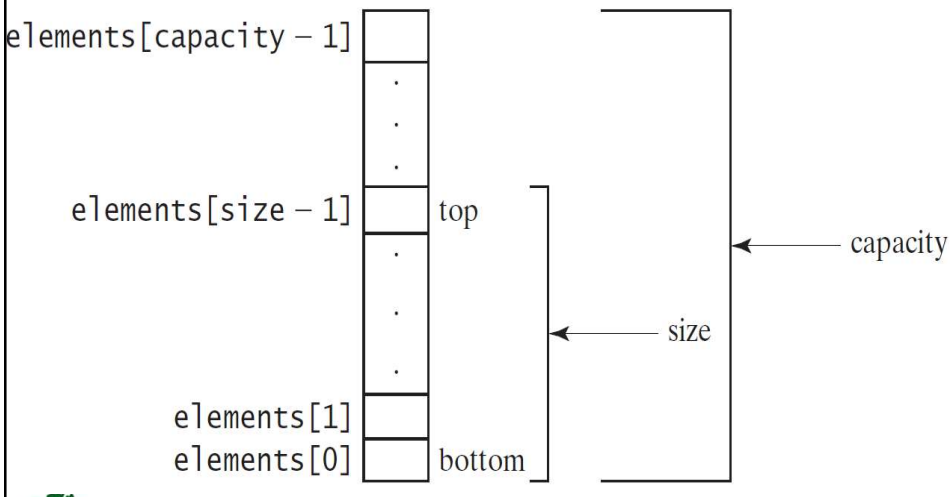
- Choose **informative names** for classes, data fields, and methods.
- Always place the data declaration before the constructor, and place constructors before methods.
- Always provide a constructor and **initialize** variables to avoid programming errors.

14

Designing Stack of Integers



Implementing StackOfIntegers



16

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.

Wrapper Classes

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

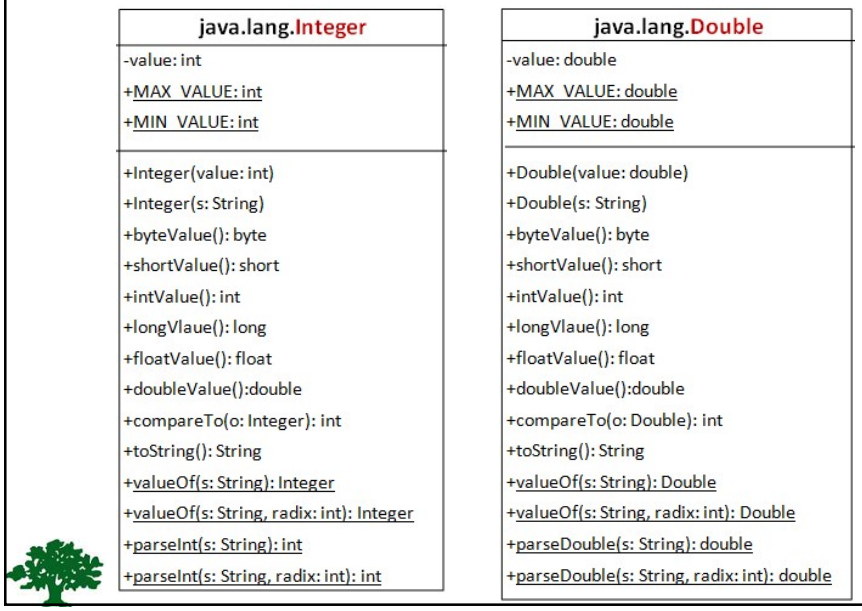
NOTE:

(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.

18

The Integer and Double Classes



19

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)

20

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.

21

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.

22

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");
```

23

The Methods for Parsing Strings into Numbers

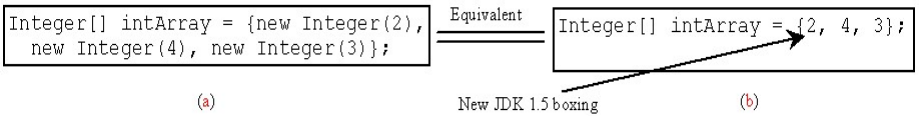
❖ You have used the **parseInt** method in the **Integer** class to parse a numeric string into an **int** value and the **parseDouble** method in the **Double** class to parse a numeric string into a **double** value.

❖ Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

24

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[] arr = {1, 2, 3};

System.out.println(arr[0] + arr[1] + arr[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***.

26

BigInteger and BigDecimal

```
BigInteger a = new BigInteger("92233720368547758O7");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

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

27

10.9 (*The* `Course` *class*) Revise the `Course` class as follows:

- The array size is fixed in Listing 10.6. Improve it to automatically increase the array size by creating a new larger array and copying the contents of the current array to it.
- Implement the `dropStudent` method.
- Add a new method named `clear()` that removes all students from the course.

Write a test program that creates a course, adds three students, removes one, and displays the students in the course.