Chapter 10
Thinking in Objects
Motivations

- You see the advantages of object-oriented programming from the preceding chapter. This chapter will demonstrate how to solve problems using the object-oriented paradigm.
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 lets 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loan()</td>
<td>Constructs a default Loan object.</td>
</tr>
<tr>
<td>Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)</td>
<td>Constructs a loan with specified interest rate, years, and loan amount.</td>
</tr>
<tr>
<td>getAnnualInterestRate(): double</td>
<td>Returns the annual interest rate of this loan.</td>
</tr>
<tr>
<td>getNumberOfYears(): int</td>
<td>Returns the number of the years of this loan.</td>
</tr>
<tr>
<td>getLoanAmount(): double</td>
<td>Returns the amount of this loan.</td>
</tr>
<tr>
<td>getLoanDate(): Date</td>
<td>Returns the date of the creation of this loan.</td>
</tr>
<tr>
<td>setAnnualInterestRate(annualInterestRate: double): void</td>
<td>Sets a new annual interest rate to this loan.</td>
</tr>
<tr>
<td>setNumberOfYears(numberOfYears: int): void</td>
<td>Sets a new number of years to this loan.</td>
</tr>
<tr>
<td>setLoanAmount(loanAmount: double): void</td>
<td>Sets a new amount to this loan.</td>
</tr>
<tr>
<td>getMonthlyPayment(): double</td>
<td>Returns the monthly payment of this loan.</td>
</tr>
<tr>
<td>getTotalPayment(): double</td>
<td>Returns the total payment of this loan.</td>
</tr>
</tbody>
</table>

The annual interest rate of the loan (default: 2.5).
The number of years for the loan (default: 1)
The loan amount (default: 1000).
The date this loan was created.
Thinking in Objects

• Procedural paradigm focuses on designing methods
• The Objected-Oriented paradigm couples data and methods together into objects
• Software design using the Objected-Oriented programming paradigm focuses on objects and operations on objects
The BMI Class

<table>
<thead>
<tr>
<th>BMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>-name: String</td>
</tr>
<tr>
<td>-age: int</td>
</tr>
<tr>
<td>-weight: double</td>
</tr>
<tr>
<td>-height: double</td>
</tr>
<tr>
<td>+BMI(name: String, age: int, weight: double, height: double)</td>
</tr>
<tr>
<td>+BMI(name: String, weight: double, height: double)</td>
</tr>
<tr>
<td>+getBMI(): double</td>
</tr>
<tr>
<td>+getStatus(): String</td>
</tr>
</tbody>
</table>

The name of the person.
The age of the person.
The weight of the person in pounds.
The height of the person in inches.

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

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.)
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.
Aggregation Between Same Class

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

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

    // ...
}
```
Aggregation Between Same Class

- A person may have several supervisors

```java
public class Person {
    ...
    private Person[] supervisors;
}
```
## Example: The Course Class

<table>
<thead>
<tr>
<th>Course</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>- courseName: String</td>
<td>The name of the course.</td>
</tr>
<tr>
<td>- students: String[]</td>
<td>An array to store the students for the course.</td>
</tr>
<tr>
<td>- numberOfStudents: int</td>
<td>The number of students (default: 0).</td>
</tr>
<tr>
<td>+ Course(courseName: String)</td>
<td>Creates a course with the specified name.</td>
</tr>
<tr>
<td>+ getCourseName(): String</td>
<td>Returns the course name.</td>
</tr>
<tr>
<td>+ addStudent(student: String): void</td>
<td>Adds a new student to the course.</td>
</tr>
<tr>
<td>+ dropStudent(student: String): void</td>
<td>Drops a student from the course.</td>
</tr>
<tr>
<td>+ getStudents(): String[]</td>
<td>Returns the students in the course.</td>
</tr>
<tr>
<td>+ getNumberOfStudents(): int</td>
<td>Returns the number of students in the course.</td>
</tr>
</tbody>
</table>
Example: The **StackOfIntegers** Class

<table>
<thead>
<tr>
<th>StackOfIntegers</th>
</tr>
</thead>
<tbody>
<tr>
<td>-elements: int[]</td>
</tr>
<tr>
<td>-size: int</td>
</tr>
<tr>
<td>+StackOfIntegers()</td>
</tr>
<tr>
<td>+StackOfIntegers(capacity: int)</td>
</tr>
<tr>
<td>+empty(): boolean</td>
</tr>
<tr>
<td>+peek(): int</td>
</tr>
<tr>
<td>+push(value: int): int</td>
</tr>
<tr>
<td>+pop(): int</td>
</tr>
<tr>
<td>+getSize(): int</td>
</tr>
</tbody>
</table>

An array to store integers in the stack.
The number of integers in the stack.
Constructs an empty stack with a default capacity of 16.
Constructs an empty stack with a specified capacity.
Returns true if the stack is empty.
Returns the integer at the top of the stack without removing it from the stack.
Stores an integer into the top of the stack.
Removes the integer at the top of the stack and returns it.
Returns the number of elements in the stack.
Designing the StackOfIntegers Class
Implementing `StackOfIntegers` Class

```
elements[capacity - 1]
  .
  .
  .

elements[size - 1]  top
  .
  .

elements[1]

elements[0]  bottom

capacity

size
```
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

<table>
<thead>
<tr>
<th>java.lang.Integer</th>
<th>java.lang.Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>-value: int</td>
<td>-value: double</td>
</tr>
<tr>
<td>+MAX_VALUE: int</td>
<td>+MAX_VALUE: double</td>
</tr>
<tr>
<td>+MIN_VALUE: int</td>
<td>+MIN_VALUE: double</td>
</tr>
<tr>
<td>+Integer(value: int)</td>
<td>+Double(value: double)</td>
</tr>
<tr>
<td>+Integer(s: String)</td>
<td>+Double(s: String)</td>
</tr>
<tr>
<td>+byteValue(): byte</td>
<td>+byteValue(): byte</td>
</tr>
<tr>
<td>+shortValue(): short</td>
<td>+shortValue(): short</td>
</tr>
<tr>
<td>+intValue(): int</td>
<td>+intValue(): int</td>
</tr>
<tr>
<td>+longValue(): long</td>
<td>+longValue(): long</td>
</tr>
<tr>
<td>+floatValue(): float</td>
<td>+floatValue(): float</td>
</tr>
<tr>
<td>+doubleValue():double</td>
<td>+doubleValue():double</td>
</tr>
<tr>
<td>+compareTo(o: Integer): int</td>
<td>+compareTo(o: Double): int</td>
</tr>
<tr>
<td>+toString(): String</td>
<td>+toString(): String</td>
</tr>
<tr>
<td>+valueOf(s: String): Integer</td>
<td>+valueOf(s: String): Double</td>
</tr>
<tr>
<td>+valueOf(s: String, radix: int): Integer</td>
<td>+valueOf(s: String, radix: int): Double</td>
</tr>
<tr>
<td>+parseInt(s: String): int</td>
<td>+parseInt(s: String): int</td>
</tr>
<tr>
<td>+parseInt(s: String, radix: int): int</td>
<td>+parseInt(s: String, radix: int): int</td>
</tr>
</tbody>
</table>
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`
  - `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 following abstract methods
  - `doubleValue`
  - `floatValue`
  - `intValue`
  - `longValue`
  - `shortValue`

• 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.

Example:

```java
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};
```

New JDK 1.5 boxing

```
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);

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

- Constructing a String:
  - String message = "Welcome to Java";
  - String message = new String("Welcome to Java");
  - String s = new String();

- Obtaining String length and Retrieving Individual Characters in a string
- String Concatenation (concat)
- Substrings (substring(index), substring(start, end))
- Comparisons (equals, compareTo)
- String Conversions
- Finding a Character or a Substring in a String
- Conversions between Strings and Arrays
- Converting Characters and Numeric Values to Strings