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.
- Before studying these examples, we first introduce several language features for supporting these examples
Objectives

- To apply class abstraction to develop software (§10.2).
- To explore the differences between the procedural paradigm and object-oriented paradigm (§10.3).
- To discover the relationships between classes (§10.4).
- To design programs using the object-oriented paradigm (§§10.5–10.6).
- To create objects for primitive values using the wrapper classes (*Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Character*, and *Boolean*) (§10.7).
- To simplify programming using automatic conversion between primitive types and wrapper class types (§10.8).
- To use the *BigInteger* and *BigDecimal* classes for computing very large numbers with arbitrary precisions (§10.9).
- To use the *String* class to process immutable strings (§10.10).
- To use the *StringBuilder* and *StringBuffer* classes to process mutable strings (§10.11).

Class Abstraction & 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

**Example:** The use of a PC does not require the knowledge of its internal workings
A specific loan can be viewed as an object of a Loan class. The interest rate, amount, and period are its properties, and computing the monthly & total payments are its methods. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, amount, and period. You can then use the methods to find the monthly & total payments. As a user of the Loan class, you don’t need to know how these methods are implemented.

Example: The Loan Class

<table>
<thead>
<tr>
<th>Loan</th>
<th>The annual interest rate of the loan (default: 2.5).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The number of years for the loan (default: 1)</td>
</tr>
<tr>
<td></td>
<td>The loan amount (default: 1000).</td>
</tr>
<tr>
<td></td>
<td>The date this loan was created.</td>
</tr>
<tr>
<td></td>
<td>Constructs a default Loan object.</td>
</tr>
<tr>
<td></td>
<td>Constructs a loan with specified interest rate, years, and loan amount.</td>
</tr>
<tr>
<td></td>
<td>Returns the annual interest rate of this loan.</td>
</tr>
<tr>
<td></td>
<td>Returns the number of the years of this loan.</td>
</tr>
<tr>
<td></td>
<td>Returns the amount of this loan.</td>
</tr>
<tr>
<td></td>
<td>Sets the amount of this loan.</td>
</tr>
<tr>
<td></td>
<td>Sets a new number of years to this loan.</td>
</tr>
<tr>
<td></td>
<td>Sets a new number to this loan.</td>
</tr>
<tr>
<td></td>
<td>Returns the monthly payment of this loan.</td>
</tr>
<tr>
<td></td>
<td>Returns the total payment of this loan.</td>
</tr>
</tbody>
</table>

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.
Example: The BMI Class

In procedural programming, data & operations on the data are separate. It requires passing data to methods. The OO approach mirrors the real world, in which objects are associated with both data & operations. Using objects improves reusability, making programs easier to develop and maintain. An OO program can be viewed as a collection of cooperating objects. This example is an OO version of the procedural program of Listing 3.4

```
BMI
- name: String
- age: int
- weight: double
- height: double

+BMI(name: String, age: int, weight: double, height: double)
+BMI(name: String, weight: double, height: double)
+ getBMI(): double
+ getStatus(): String
```

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.)

Class Relationships

1. Association
2. Aggregation
3. Composition
4. Inheritance

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

A "uses" B. B exists independently (conceptually) from A

Example: A Company is an aggregation of People

A Company is a composition of Accounts

When a Company ceases to do business its Accounts cease to exist but its People continue to exist

Composition

A "owns" B. B has no meaning or purpose in the system without A

Example: A Text Editor owns a Buffer (composition)

A Text Editor uses a File (aggregation)

When Text Editor is closes, the Buffer is destroyed but the File is not

Aggregation & 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
Aggregation & Composition

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

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

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

public class Address {
  ...
}
```

Aggregation Between Same Class

Aggregation may exist between objects of the same class.

```java
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?

Person

Supervisor

1

m

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

Example: The Course Class

The Course class encapsulates the internal implementation. The user can create a Course object and manipulate it through the public methods addStudent, dropStudent, getNumOfStudents and getStudents. However, the user doesn’t need to know how these methods are implemented.

| Course | | |
|-----------------|-----------------|
| -courseName: String | -students: String[] |
| -numberOfStudents: int | +Course(courseName: String) |
| +getCourseName(): String | +addStudent(student: String): void |
| +dropStudent(student: String): void | +getStudents(): String[] |
| +getNumberOfStudents(): int | |

This example uses an array to store students, but you could use a different data structure to store students. The program that uses Course does not need to change as long as the contract of the public methods remains unchanged.
Example: Stack of Integers

Stack of Integers

- `elements[capacity - 1]`
- `elements[size - 1]` (top)
- `elements[1]`
- `elements[0]` (bottom)
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.

---

**Wrapper Classes**

Wrapper classes are used when an **object**
representation of a primitive data type is required

- Wrapper classes do not have **no-arg constructors**
- The instances of all wrapper classes are **immutable**

1. **Boolean**
2. **Character**
3. **Short**
4. **Byte**
5. **Integer**
6. **Long**
7. **Float**
8. **Double**
The Integer & Double Classes

- 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:

  ```java
  public Integer(int value)
  public Integer(String s)
  public Double(double value)
  public Double(String s)
  ```

**MIN_VALUE & MAX_VALUE**

- **MAX_VALUE** represents the maximum value of the corresponding numeric 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 smallest **positive** float and double values

- **MAX_VALUE** for **Integer** is **2,147,483,647**

- **MIN_VALUE** for **Float** is **1.4E-45**

- **MAX_VALUE** for **Double** is **1.79769313486231570e+308d**
Conversion to Primitive Data Types

Each numeric wrapper class implements the abstract methods:

- `doubleValue()`
- `floatValue()`
- `intValue()`
- `longValue()`
- `shortValue()`

which are defined in the `Number` abstract class. These methods convert objects into primitive type values.

Conversion from Numeric String to Object

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.

Examples:

```java
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```
Parsing Strings into Numbers

- We 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.

- `valueOf()` returns an object. `parseInt()` and `parseDouble()` return a primitive data type.

Automatic Conversion Between Primitive Types & Wrapper Class Types

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

```
Integer[] intArray = {new Integer(2), new Integer(4), new Integer(3)};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

(a) **Boxing** (automatic primitive type to object conversion)

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

(b) **Unboxing** (automatic object to primitive type conversion)
BigInteger & BigDecimal Classes

- If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes of the java.math package

- Both are immutable

- Both extend the Number abstract class and implement the Comparable interface

```
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);
System.out.println(c);
```
The String Class

- Constructing a string:
  ```java
  String message = "Welcome to Java";
  String message = new String("Welcome to Java");
  String s = new String();
  ```
- `length()`
- `concat()`
- `equals()`
- `compareTo()`
- `charAt()`
- `substring(index), substring(start, end)`
- Finding a character or a substring in a string
- String conversions
- Conversions between strings and arrays
- Converting characters and numeric values to strings

Constructing Strings

```java
String newString = new String(stringLiteral);
String message = new String("Welcome to Java");
```

Since strings are used frequently, Java provides a **shorthand** initializer for creating a string:

```java
String message = "Welcome to Java";
```
Strings are Immutable

A String object is immutable; its contents cannot be changed

Does the following code change the contents of the string?

```java
String s = "Java";
s = "HTML";
```

Trace Code

```java
String s = "Java";
s = "HTML";
```

After executing `String s = "Java";`

- `s` is a `String` object for "Java"
- Contents cannot be changed

After executing `s = "HTML";`

- `s` is a `String` object for "HTML"
- This string object is now unreferenced
Trace Code

```java
String s = "Java";
s = "HTML";
```

After executing `s = "Java";`

```java
String object for "Java"
```

After executing `s = "HTML";`

```java
String object for "HTML"
```

This string object is now unreferenced

Contents cannot be changed

Interned Strings

- Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a **unique instance for string literals with the same character sequence**

- Such an instance is called **interned**
Example: Interned Strings

- A new object is created if you use the `new` operator
- If you use the string initializer, **no new object is created** if the interned object is already created

```java
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

display

- `s1 == s2` is false
- `s1 == s3` is true

Trace Code

```java
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
```
Trace Code

String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";

\[
\begin{array}{c}
\text{s1} \\
\text{String} \\
\text{Interned string object for "Welcome to Java"} \\
\text{s2} \\
\text{String} \\
\text{A string object for "Welcome to Java"} \\
\text{s3} \\
\end{array}
\]
Replacing & Splitting Strings

java.lang.String

+replace(oldChar: char, newChar: char): String
+replaceFirst(oldString: String, newString: String): String
+replaceAll(regex: String, newString: String): String
+split(delimiter: String): String[]

Returns a new string that replaces all matching oldChar in this string with the newChar.

Returns a new string that replaces the first matching oldString in this string with the newString substring.

Returns a new string that replace all matching regex in this string with the newString.

Returns an array of strings consisting of the substrings split by the delimiter.

Examples: Replacing Substrings

Replace all occurrences of a character in a string
"Welcome".replace('e', 'A') returns a new string, WAlcomA

Replace first occurrence of a substring in a string
"Welcome".replaceFirst("e", "AB") returns a new string, WABlcome

Replace all occurrences of a single-character substring in a string
"Welcome".replace("e", "AB") returns a new string, WABlcomAB

Replace all occurrences of a multi-character substring in a string
"Welcome".replace("el", "AB") returns a new string, WABcome
**Example: Splitting a String**

```java
String[] tokens = "Java#HTML#Perl".split("#", 0);
for (int i = 0; i < tokens.length; i++)
    System.out.print(tokens[i] + " ");
```

displays
```
Java HTML Perl
```

**Matching, Replacing & Splitting by Patterns**

- You can match, replace, or split a string by specifying a pattern
- This is an extremely useful and powerful feature, commonly known as *regular expression*. For this reason, two simple patterns are used in this section

```java
"Java".matches("Java");
"Java".equals("Java");

"Java is fun".matches("Java.*");
"Java is cool".matches("Java.*");
```
Matching, Replacing & Splitting by Patterns

The `replaceAll()`, `replaceFirst()`, and `split()` methods can be used with a regular expression. For example, the following statement returns a new string that replaces $, +, or # in "a+b$#c" by the string NNN.

```java
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

Here the regular expression `[$+#]` specifies a pattern that matches $, +, or #. So, the output is aNNNbNNNNNc.

Matching, Replacing & Splitting by Patterns

The following statement splits the string into an array of strings delimited by the punctuation marks [.,::;?]

```java
String[] tokens = "Java,C?C#,C++".split("[.,::;?]");

for (int i = 0; i < tokens.length; i++)
    System.out.println(tokens[i]);
```

displays
Java
C
C#
C++
Convert Characters & Numbers to Strings

- The `String` class provides several `static` `valueOf()` methods for converting a character, an array of characters, and numeric values to strings
- These methods have the same name `valueOf()` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`

  **Example:** To convert a double value to a `String`, use `String.valueOf(5.44)`. The return value is a string consisting of characters 5, 4, 4.

StringBuilder & StringBuffer Classes

- The `StringBuilder` and `StringBuffer` classes are an alternative to the `String` class
- In general, they can be used wherever a `String` is used
- They are more flexible than `String`. You can `add`, `insert`, or `append` new contents into a `StringBuilder` or `StringBuffer`, whereas the value of a `String` object is fixed once it is created
- `StringBuffer` is appropriate for threaded applications, and `StringBuilder` for all others
**StringBuilder Constructors**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StringBuilder()</code></td>
<td>Constructs an empty string builder with capacity 16.</td>
</tr>
<tr>
<td><code>StringBuilder(capacity: int)</code></td>
<td>Constructs a string builder with the specified capacity.</td>
</tr>
<tr>
<td><code>StringBuilder(s: String)</code></td>
<td>Constructs a string builder with the specified string.</td>
</tr>
</tbody>
</table>

**StringBuilder**

+ `append(data: char[]): StringBuilder`  
  Appends a char array into this string builder.

+ `append(data: char[], offset: int, len: int): StringBuilder`  
  Appends a subarray in data into this string builder.

+ `append(v: aPrimitiveType): StringBuilder`  
  Appends a primitive type value as a string to this builder.

+ `append(s: String): StringBuilder`  
  Appends a string to this string builder.

+ `delete(startIndex: int, endIndex: int): StringBuilder`  
  Deletes characters from startIndex to endIndex.

+ `deleteCharAt(index: int): StringBuilder`  
  Deletes a character at the specified index.

+ `insert(index: int, data: char[], offset: int, len: int): StringBuilder`  
  Inserts a subarray of the data in the array to the builder at the specified index.  
  Inserts data into this builder at the position offset.

+ `insert(offset: int, data: char[]): StringBuilder`  
  Inserts data into this builder at the position offset.

+ `insert(offset: int, b: aPrimitiveType): StringBuilder`  
  Inserts a value converted to a string into this builder.

+ `insert(offset: int, s: String): StringBuilder`  
  Inserts a string into this builder at the position offset.

+ `replace(startIndex: int, endIndex: int, s: String): StringBuilder`  
  Replaces the characters in this builder from startIndex to endIndex with the specified string.  
  Reverses the characters in the builder.

+ `reverse(): StringBuilder`  
  Sets a new character at the specified index in this builder.
Examples: Modifying Strings in the Builder

```java
stringBuilder.append("Welcome to Java") initializes the builder

stringBuilder.insert(11, "HTML and ") changes the builder to Welcome to HTML and Java

stringBuilder.delete(8, 11) changes the builder to Welcome Java

stringBuilder.deleteCharAt(8) changes the builder to Welcome o Java

stringBuilder.reverse() changes the builder to avaJ ot emocleW

stringBuilder.replace(11, 15, "HTML") changes the builder to Welcome to HTML

stringBuilder.setCharAt(0, 'w') sets the builder to welcome to Java
```

```java
toString() capacity() charAt() length() setLength()
```

```
java.lang.StringBuilder
+toString(): String
+capacity(): int
+charAt(index: int): char
+length(): int
+setLength(newLength: int): void
+substring(startIndex: int): String
+substring(startIndex: int, endIndex: int): String
+trimToSize(): void
```

Returns a string object from the string builder.

Returns the capacity of this string builder.

Returns the character at the specified index.

Returns the number of characters in this builder.

Sets a new length in this builder.

Returns a substring starting at startIndex.

Returns a substring from startIndex to endIndex-1.

Reduces the storage size used for the string builder.
Example: Checking Palindromes
Ignoring Non-alphanumeric Characters

Problem: Check whether a string is a palindrome (a string that reads the same forward and backward). Ignore the non-alphanumeric characters while doing so.