**CHAPTER 10: OBJECT-ORIENTED THINKING**

* 1. **CLASS ABSTRACTION AND ENCAPSULATION**

Class abstraction is 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.

Java provides many levels of abstraction, and **class abstraction** separates class implementation from how the class is used.

The creator of a class describes the functions of the class and lets the user know how the class can be used. The collection of public constructors, methods, and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the **class’s contract**.

a class is also known as an **abstract data type (ADT).**

Class abstraction separates class implementation from the use of the class.

* 1. **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.

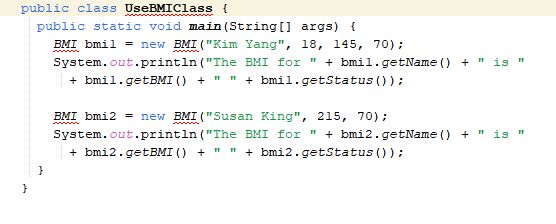
Classes provide more flexibility and modularity for building reusable software.

This section improves in gaining insights into the differences between procedural and object-oriented programming, and see the benefits of developing reusable code using objects and classes.

You can define a class named BMI as shown below:

|  |  |
| --- | --- |
| BMI | |
| Name: String | The name of the person. |
| Age: int | The age of the person. |
| Weight: double | The weight of the person in pounds. |
| Height: double | The height of the person in inches. |
| +BMI(name: String, age: int, weight: double, height: double) | Creates a BMI object with the specified name, age, weight, and height. |
| +BMI(name: String, weight: double, height: double) | Creates a BMI object with the specified name, weight, height, and a default age 20. |
| +getBMI(): double | Returns the BMI. |
| +getStatus(): String | Returns the BMI status (e.g., normal, overweight, etc.). |

The BMI class encapsulates BMI information.

****

The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Using objects improves software reusability and makes programs easier to develop and easier to maintain.

* 1. **CLASS RELATIONSHIPS**

To design classes, you need to explore the relationships among classes. The common relationships among classes are association, aggregation, composition, and inheritance.

* + 1. **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 graphical notation, as shown below:

**Student**

**Course**

**Faculty**

Take

Teach

Teacher

5-60

1

\*

This 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 5 to 60 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.

* + 1. **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**.

We refer aggregation between two objects as **composition** if the existence of the aggregated object is dependent on the aggregating object.

In other words, if a relationship is composition, the aggregated object cannot exist on its own.

* 1. **PROCESSING PRIMITIVE DATA TYPES VALUES AS OBJECTS**

A primitive-type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.

Primitive data type values are not objects in Java.

Java offers a convenient way to incorporate, or wrap, a primitive data type value into an object (e.g., wrapping an **int** into an **Integer** object, wrapping a **double** into a **Double** object, and wrapping a **char** into a **Character** object).

By using a wrapper class, you can process primitive data type values as objects.

Java provides **Boolean, Character, Double, Float, Byte, Short, Integer,** and **Long** wrapper classes in the **java.lang** package for primitive data types.

Numeric wrapper classes are very similar to each other. Each contains the methods **doubleValue(),** **floatValue(),** **intValue(),** **longValue(),** **shortValue(),** and **byteValue().** These methods “convert” objects into primitive-type values.

The key features of Integer and Double are shown below.

|  |  |
| --- | --- |
| java.lang.Integer | Java.lang.Double |
| –value: int  +MAX\_VALUE: int  +MIN\_VALUE: int | –value: double  +MAX\_VALUE: double  +MIN\_VALUE: double |
| +Integer(value: int)  +Integer(s: String)  +byteValue(): byte  +shortValue(): short  +intValue(): int  +longValue(): 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 | +Double(value: double)  +Double(s: String)  +byteValue(): byte  +shortValue(): short  +intValue(): int  +longValue(): 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 wrapper classes provide constructors, constants, and conversion methods for manipulating various data types.

* 1. **AUTOMATIC CONVERSION BETWEEN PRIMITIVE TYPES AND WRAPPER CLASS TYPES**

A primitive-type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context.

Converting a primitive value to a wrapper object is called **boxing**. The reverse conversion is called **unboxing**.

Java allows primitive types and wrapper classes to be converted automatically.

The compiler will automatically box a primitive value that appears in a context requiring an object, and unbox an object that appears in a context requiring a primitive value. This is called **autoboxing** and **autounboxing**.

The following statement in (a) is the same as in (b) due to autounboxing:

Integer intObject = Integer.valueOf(2);

Integer intObject = 2;

Equivalent

(a)

(b)

autoboxing

* 1. **THE BigInteger and BigDecimal classes**

The BigInteger and BigDecimal classes can be used to represent integers or decimal numbers of any size and precision.

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.

The largest integer of the long type is **Long.MAX\_VALUE (i.e., 9223372036854775807).**

An instance of **BigInteger** can represent an integer of any size.

You can use new **BigInteger(String)** or **BigInteger.valueOf(long)** to create an instance of **BigInteger** and new **BigDecimal(String)** or **BigDecimal.valueOf(double)** to create an instance of **BigDecimal**, use the **add, subtract, multiply, divide,** and **remainder** methods to perform arithmetic operations, and use the **compareTo** method to compare two big numbers.

For example, the following code creates two BigInteger objects and multiplies them:

BigInteger a = new BigInteger("9223372036854775807");

BigInteger b = new BigInteger("2");

BigInteger c = a.multiply(b);

System.out.println(c);

The output is 18446744073709551614.

There is no limit to the precision of a **BigDecimal** object. The divide method may throw an **ArithmeticException** if the result cannot be terminated.

However, you can use the overloaded divide(BigDecimal d, int scale, RoundingMode roundingMode) method to specify a scale and a rounding mode to avoid this exception, where **scale** is the maximum number of digits after the decimal point.

* 1. **THE STring CLASS**

A **String** object is immutable; its contents cannot be changed once the string is created.

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use the syntax:

String newString = new String(stringLiteral);

The argument **stringLiteral** is a sequence of characters enclosed in double quotes.

The following statement creates a String object message for the string literal "Welcome to Java":

String message = **new** String("**Welcome to Java**");

* + 1. **Immutable Strings and Interned Strings**

A **String** object is immutable; its contents cannot be changed.

Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an **interned string**.

* + 1. **Replacing and Splitting Strings**

The **String** class provides the methods for replacing and splitting strings, as shown below:

|  |  |
| --- | --- |
| java.lang.String | |
| +replace(oldChar: char, newChar: char): String | Returns a new string that replaces all matching characters in this string with the new character. |
| +replaceFirst(oldString: String, newString: String): String | Returns a new string that replaces the first matching substring in this string with the new substring. |
| +replaceAll(oldString: String, newString: String): String | Returns a new string that replaces all matching substrings in this string with the new substring. |
| +split(delimiter: String): String[] | Returns an array of strings consisting of the substrings split by the delimiter. |

The **String** class contains the methods for replacing and splitting strings.

Once a string is created, its contents cannot be changed. The methods replace, replaceFirst, and replaceAll return a new string derived from the original string ( without changing the original string!).

Several versions of the replace methods are provided to replace a character or a substring in the string with a new character or a new substring.

* + 1. **Matching, Replacing, and Splitting by Patterns**

A **regular expression** (abbreviated **regex**) is a string that describes a pattern for matching a set of strings.

You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature.

The following statement evaluates to true:

"**440–02–4534**".matches**("\\d{3}–\\d{2}–\\d{4}")**

Here, **\\d** represents a single digit, and **\\d{3}** represents three digits.

The following statement splits the string into an array of strings delimited by punctuation marks.

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

for (**int** i = **0**; i < tokens.length; i++)

System.out.println(tokens[i]);

Here the regular expression **[.,:;?]** specifies a pattern that matches **., ,, :, ;, or ?.** Each of these characters is a delimiter for splitting the string. Thus, the string is split into **Java, C, C#,** and **C++** which are stored in array tokens.

* + 1. **Conversion between Strings and Arrays**

Strings are not arrays, but a string can be converted into an array and vice versa. To convert a string into an array of characters, use the **toCharArray** method.

the following statement converts the string Java to an array:

**char[] chars = "Java".toCharArray();**

* + 1. **Converting Characters and Numeric Values to Strings**

Recall that you can use **Double.parseDouble(str)** or **Integer.parseInt(str)** to convert a string to a double value or an **int** value, and you can convert a character or a number into a string by using the string concatenating operator.

Another way of converting a number into a string is to use the overloaded static **valueOf** method. This method can also be used to convert a character or an array of characters into a string, as shown below:

|  |  |
| --- | --- |
| java.lang.String | |
| +valueOf(c: char): String | Returns a string consisting of the character c. |
| +valueOf(data: char[]): String | Returns a string consisting of the characters in the array. |
| +valueOf(d: double): String | Returns a string representing the double value. |
| +valueOf(f: float): String | Returns a string representing the float value. |
| +valueOf(i: int): String | Returns a string representing the int value. |
| +valueOf(l: long): String | Returns a string representing the long value. |
| +valueOf(b: boolean): String | Returns a string representing the boolean value. |

The **String** class contains the static methods for creating strings from primitive-type values.

* + 1. **Formatting Strings**

The **String** class contains the static **format** method to return a formatted string. The syntax to invoke this method is

String.format(format, item1, item2, ..., itemk);

This method is similar to the **printf** method except that the **format** method returns a formatted string, whereas the **printf** method displays a formatted string.

* 1. **THE StringBuilder AND StringBuffer CLASSES**

The **StringBuilder** and **StringBuffer** classes are similar to the String class except that the String class is immutable.

The **StringBuilder** and **StringBuffer** classes can be used wherever a string is used. They are more flexible than String. You can add, insert, or append new contents into **StringBuilder** and **StringBuffer** objects, whereas the value of a String object is fixed once the string is created.

The **StringBuilder** class is similar to **StringBuffer** except that the methods for modifying the buffer in **StringBuffer** are **synchronized**, which means that only one task is allowed to execute the methods.

|  |  |
| --- | --- |
| java.lang.StringBuilder | |
| +StringBuilder() | Constructs an empty string builder with capacity 16. |
| +StringBuilder(capacity: int) | Constructs a string builder with the specified capacity. |
| +StringBuilder(s: String) | Constructs a string builder with the specified string. |

The **StringBuilder** class contains the constructors for creating instances of **StringBuilder**.

* + 1. **Modifying Strings in the StringBuilder**

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed below:

|  |  |
| --- | --- |
| java.lang.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–1. |
| +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 into the builder at the specified index. |
| +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–1with the specified string. |
| +reverse(): StringBuilder | Reverses the characters in the builder. |
| +setCharAt(index: int, ch: char): void | Sets a new character at the specified index in this builder. |

The **StringBuilder** class contains the methods for modifying string builders.

* + 1. **The toString, capacity, length, setLength, and charAt Methods**

The **StringBuilder** class provides the additional methods for manipulating a string builder and obtaining its properties, as shown below:

|  |  |
| --- | --- |
| java.lang.StringBuilder | |
| +toString(): String | Returns a string object from the string builder. |
| +capacity(): int | Returns the capacity of this string builder. |
| +charAt(index: int): char | Returns the character at the specified index. |
| +length(): int | Returns the number of characters in this builder. |
| +setLength(newLength: int): void | Sets a new length in this builder. |
| +substring(startIndex: int): String | Returns a substring starting at startIndex. |
| +substring(startIndex: int, endIndex: int): String | Returns a substring from startIndex to endIndex – 1. |
| +trimToSize(): void | Reduces the storage size used for the string builder. |

The **StringBuilder** class contains the methods for modifying string builders.

**Note**: The length of the string builder is always less than or equal to the capacity of the builder.

The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder.

**THE END!**