Object-Oriented Programming [1]

A Study Guide for Students of Sorsogon State University - Bulan Campus[2]

Jarrian Vince G. Gojar[3]

November 13, 2024

[1] A course in the Bachelor of Science in Computer Science

[2] This book is a study guide for students of Sorsogon State University - Bulan Campus taking up the course Object-Oriented Programming.

[3] https://github.com/godkingjay

Sorsogon State University - Bulan Campus

Contents

List of Figures

List of Tables

List of Codes

Preface

"The only way to learn a new programming language is by writing programs in it."

– Dennis Ritchie

Jarrian Vince G. Gojar
https://github.com/godkingjay

1

Introduction to Java Programming Language and Object-Oriented Programming

1.1 Introduction

Java is a general-purpose, concurrent, class-based, object-oriented programming language that was designed and developed by Sun Microsystems in the early 1990s. It is currently owned by Oracle Corporation. Java is one of the most popular programming languages in use, particularly for client-server web applications.

1.1.1 History of Java

Before Java, the primary programming language was C++. C++ is a powerful programming language, but it is complex and difficult to learn. It is also platform-dependent, which means that C++ programs must be recompiled for each operating system. Java was designed to be easy to use and is therefore easier to write, compile, debug, and learn than C++.

Java was developed by James Gosling, Mike Sheridan, and Patrick Naughton at Sun Microsystems in the early 1990s. It was first released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1991 and first released in 1995. As of May 2007, in compliance with the specifications of the Java Community Process, Sun made available most of their Java technologies as free software under the GNU General Public License. Sun's vice-president Rich Green said that Sun's ideal role with regard to Java was as an evangelist.

Sun Microsystems released the first public implementation as Java 1.0 in 1995. It promised Write Once, Run Anywhere (WORA), providing no-cost run-times on popular platforms. Fairly secure and featuring configurable security, it allowed network- and file-access restrictions. Major web browsers soon incorporated the ability to run Java applets within web pages, and Java quickly became popular.

In 2006, for marketing purposes, Sun renamed new J2 versions as Java EE, Java ME, and Java SE, respectively.

In 2006, Sun released much of its Java virtual machine (JVM) as free and open-source software (FOSS), under the terms of the GNU General Public License (GPL). Sun's software strategy had been to use the language to sell hardware, so FOSS would have put Sun at a disadvantage.

After Oracle Corporation acquired Sun in 2009, Oracle has described itself as the "steward of Java technology with a relentless commitment to fostering a community of participation and transparency".

In 2011, Oracle Corporation sued Google for having distributed a new implementation of Java embedded in the Android operating system. Google had not acquired any licenses from Sun or Oracle. The American court system ruled in favor of Google stating that the implementation of Java in Android was considered fair use.

In September 2017, Mark Reinhold, chief Architect of the Java Platform, proposed to change the release train to "one feature release every six months". Therefore, Oracle announced that it would be moving away from a release model that produces a major release every two years, and instead moving to a model that produces a feature release every six months. The first version was released in March 2018.

In 2019, the Oracle Corporation announced that it would be moving Java EE to the Eclipse Foundation, to make the process of developing Java EE faster, more agile, and more open.

In recent years, Java has been one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers.

As of today, Java is one of the most popular programming languages in use, particularly for client-server web applications.

1.1.2 Java Virtual Machine(JVM)

The Java Virtual Machine (JVM) is an abstract computing machine that enables a computer to run a Java program. There are three notions of the JVM: specification, implementation, and instance. The specification is a document that formally describes what is required of a JVM implementation. Having a single specification ensures all implementations are interoperable. A JVM implementation is a computer program that meets the requirements of the JVM specification. An instance of a JVM is an implementation running in a process that executes a computer program compiled into Java bytecode.

1.1.3 Java Development Kit(JDK)

The Java Development Kit (JDK) is a software development kit used to develop Java applications and applets. It is one of the most widely used Java software development kits. It consists of the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), and other tools needed in Java development.

To install the JDK, you need to download the JDK from the Oracle website. You can download the JDK from the following URL: https://www.oracle.com/ph/java/technologies/downloads/

1.1.4 Java IDE (Integrated Development Environment)

An Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally

consists of a source code editor, build automation tools, and a debugger. Most modern IDEs have intelligent code completion.

There are many IDEs available for Java development. Some of the most popular IDEs are:

- Eclipse
- NetBeans
- IntelliJ IDEA
- JDeveloper
- BlueJ
- DrJava
- JCreator
- JGrasp
- MyEclipse
- Oracle JDeveloper
- Visual Studio Code
- Xcode
- Android Studio
- Code::Blocks
- CodeLite

One of the most popular IDEs for Java development is Eclipse. Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages through the use of plugins, including Ada, ABAP, C, C++, and more.

To install Eclipse, you need to download the Eclipse IDE from the Eclipse website. You can download Eclipse from the following URL: https://www.eclipse.org/downloads/

1.1.5 Summary

Java is a general-purpose, concurrent, class-based, object-oriented programming language that was designed and developed by Sun Microsystems in the early 1990s. It is currently owned by Oracle Corporation. Java is one of the most popular programming languages in use, particularly for client-server web applications.

1.2 Basic Syntax

Java is a case-sensitive language. This means that the language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters. The following are the keywords in Java.These are reserved words that have special meaning to the compiler and cannot be used for other purposes:

- abstract
- assert
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- default
- do
- double
- else
- enum
- extends
- final
- finally
- float
- for
- goto
- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- package
- private
- protected
- public
- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- try
- void
- volatile
- while

1.2.1 Java Program Structure

A Java program is a collection of classes. A class is a blueprint from which objects are created. A class can contain fields (variables) and methods (functions). A Java program must have a class definition. A class definition includes the following components:

- Modifiers
- Class name
- Superclass (if any)
- Interfaces (if any)
- Body
- Fields
- Methods
- Constructors
- Blocks
- Nested classes and interfaces
- Annotations
- Comments
- Package statement
- Import statement
- Main method
- Statements and expressions
- Variables
- Literals
- Arrays
- Enumerations

```java
// BasicSyntax.java
package com.oop.BasicSyntax;

public class BasicSyntax {
  public static void main(String[] args) {
    System.out.println("This is the basic syntax of Java programming
        language.");
  }
}
```

Code 1.1: Basic Syntax of a Java Program

The code above shows the basic syntax of a Java program. The program prints the message "This is the basic syntax of Java programming language."

1.2.1.1 package

```java
package com.oop.BasicSyntax;
```

Code 1.2: Package Declaration

The line 'package com.oop.BasicSyntax;' in Java is used to declare the package to which the current Java file belongs. In this case, the Java file is part of the 'com.oop.BasicSyntax' package. Packages are used to organize classes and interfaces into namespaces, making it easier to manage and maintain large Java projects.

The package declaration must be the first line in a Java file, and it is optional. If a package declaration is not included in a Java file, the file is considered to be part of the default package.

The package name must be a valid Java identifier, and it should follow the Java naming conventions. The package name can be a simple name or a compound name separated by periods. For example, 'com.oop.BasicSyntax' is a simple package name, and 'com.oop.BasicSyntax.util' is a compound package name.

The package has the following syntax:

- [package] [package_name];

The package declaration includes the following components:

- package: The keyword that indicates the start of the package declaration.
- [package_name]: The name of the package to which the Java file belongs.

1.2.1.2 class

```java
public class BasicSyntax {
  // Class body
}
```

Code 1.3: Class Declaration

The line "public class BasicSyntax {}" in Java is used to declare a class named "BasicSyntax". A class is a blueprint for creating objects in Java. It defines the structure and behavior of objects by specifying fields (variables) and methods

The class has the following syntax:

- [access_modifier] class [class_name] { [class_body] }

The code block inside the class is called the class body. The class body contains the fields, methods, constructors, and other components of the class.

The class declaration has the following components:

- [access_modifier]: The access modifier that specifies the visibility of the class.
- class: The keyword that indicates the start of the class declaration.
- [class_name]: The name of the class. The class name should be a valid Java identifier.
- { [class_body] }: The code block inside the class. The class body contains the fields, methods, constructors, and other components of the class.

The classes in Java can have the following access modifiers:

- public: The class is accessible by any other class.
- protected: The class is accessible by classes in the same package and subclasses in other packages.
- default: The class is accessible only by classes in the same package.
- private: The class is accessible only within the same class.
- final: The class cannot be subclassed.
- abstract: The class cannot be instantiated.
- strictfp: The class follows the strict floating-point rules defined by the IEEE 754 standard.

For coding conventions, the class name should be a valid Java identifier, and it should follow the Java naming conventions. The class name should be in CamelCase format, starting with an uppercase letter.

The class body is enclosed in curly braces {}. The class body contains the fields, methods, constructors, and other components of the class.

1.2.1.3 main method

```
1 public static void main(String[] args) {
2   // Method body
3 }
```

Code 1.4: Main Method or Driver Method

The line "public static void main(String[] args) {}" in Java is used to declare the main method of the class. The main method is the entry point of a Java program. When a Java program is executed, the main method is called by the Java Virtual Machine (JVM) to start the program.

The main method has the following syntax:

- public static void main(String[] args) { [method_body] }

The main method is a special method in Java that has the following components:

- public: The access modifier that indicates the main method is accessible by any other class.
- static: The keyword that indicates the main method belongs to the class and not to the instance of the class.
- void: The return type of the main method. The main method does not return any value.
- main: The name of the main method. The main method is the entry point of a Java program.
- String[] args: The parameter of the main method. The args parameter is an array of strings that stores the command-line arguments passed to the Java program.
- [method_body]: The code block inside the main method. The method body contains the statements and expressions that define the behavior of the main method.

The main method is the entry point of a Java program. When a Java program is executed, the Java Virtual Machine (JVM) calls the main method to start the program.

```
1 System.out.println("This is the basic syntax of Java programming
    language.");
```

Code 1.5: Print Message to Console

The line “System.out.println("This is the basic syntax of Java programming language.");” in Java is used to print a message to the console. The “System.out.println()” method is used to print a message to the standard output stream (console).

The “System.out.println()” method has the following syntax:

- System.out.println([message]);

It has the following components:

- System: The class name. The System class is a predefined class in Java that provides access to the system resources.
- out: The static field of the System class. The out field is an instance of the PrintStream class that provides methods to write data to the standard output stream (console).
- println(): The method of the PrintStream class. The println() method is used to print a message to the standard output stream (console).
- [message]: The message to be printed. The message can be a string, a number, or any other value that can be converted to a string.

1.2.2 Summary

The basic syntax of a Java program includes the package declaration, class declaration, and main method. The package declaration is used to declare the package to which the Java file belongs. The class declaration is used to declare a class in Java. The main method is the entry point of a Java program and is called by the Java Virtual Machine (JVM) to start the program.

1.3 Lexical Structure

The lexical structure of a programming language defines the set of valid tokens that can be used to write programs in the language. The lexical structure of Java includes the following elements:

- Unicode: Java programs are written using the Unicode character set, which supports a wide range of characters from different languages.
- Lexical Translations: Java programs are translated into Unicode characters using the Unicode escape sequences.
- Unicode Escapes: Unicode escape sequences are used to represent Unicode characters in Java programs.
- Line Terminators: Line terminators are used to mark the end of a line in a Java program.
- Input Elements and Tokens: Input elements are the smallest individual units of a Java program, and tokens are the meaningful sequences of input elements.
- White Space: White space characters are used to separate tokens and improve the readability of a Java program.
- Comments: Comments are used to document Java programs and improve their readability.
- Identifiers: Identifiers are used to name classes, methods, variables, and other elements in a Java program.
- Keywords: Keywords are reserved words in Java that have special meanings and cannot be used as identifiers.
- Literals: Literals are fixed values that are used in Java programs, such as numbers, characters, and strings.
- Separators: Separators are used to separate tokens in a Java program, such as parentheses, braces, and commas.
- Operators: Operators are used to perform operations on operands in a Java program, such as addition, subtraction, and comparison.

1.3.1 Unicode

Unicode is a character encoding standard that supports a wide range of characters from different languages and scripts. Java programs are written using the Unicode character set, which allows developers to use characters from various languages in their code.

Unicode escape sequences are used to represent Unicode characters in Java programs. For example, the Unicode escape sequence \u0041 represents the character 'A' in the Latin alphabet.

```
1  char unicodeCharacter = '\u0041';
2   System.out.println("Unicode Character: " + unicodeCharacter);
```

Code 1.6: Unicode Character

The code above declares a variable named unicodeCharacter of type char and assigns the Unicode character \u0041 to it. The code then prints the Unicode character 'A' to the console.

Read more about Unicode at https://unicode.org/

1.3.2 Line Terminators

Line terminators are used to mark the end of a line in a Java program. The line terminator can be a carriage return (CR), a line feed (LF), or a carriage return followed by a line feed (CRLF).

The line terminator is used to separate lines of code in a Java program and improve the readability of the code.

- CR (Carriage Return): Moves the cursor to the beginning of the line.

- LF (Line Feed): Moves the cursor to the next line.
- CRLF (Carriage Return Line Feed): Moves the cursor to the beginning of the next line.

```
1  // Carriage Return (CR)
2  System.out.print("Hello, World!\r");
3
4  // Line Feed (LF)
5  System.out.print("Hello, World!\n");
6
7  // Carriage Return Line Feed (CRLF)
8  System.out.print("Hello, World!\r\n");
```

Code 1.7: Line Terminators

The code above demonstrates the use of line terminators in Java. The code prints the message "Hello, World!" to the console using different line terminators.

1.3.3 White Space

White space characters are used to separate tokens in a Java program and improve the readability of the code. White space characters include spaces, tabs, and line terminators.

White space is ignored by the Java compiler, so developers can use it liberally to format their code and make it more readable.

```
1  // Using White Space to Improve Readability
2  int number = 10;
3  int result = number * 2;
4  System.out.println("Result: " + result);
5
6  if (result > 10) {
7    System.out.println("Result is greater than 10");
8  } else {
9    System.out.println("Result is less than or equal to 10");
10 }
```

Code 1.8: Using White Space to Improve Readability

The code above demonstrates the use of white space characters in Java. The code uses spaces and line breaks to format the code and make it more readable.

```
1  int number=10;int result=number*2;System.out.println("Result:
       "+result);if(result>10){System.out.println("Result is greater than
       10");}else{System.out.println("Result is less than or equal to 10");}
```

Code 1.9: Code Without White Space

The code above is the same as the previous code but without white space characters. The code is difficult to read and understand because it lacks proper formatting.

1.3.4 Comments

Comments are used to document Java programs and improve their readability. Comments are ignored by the Java compiler and are not executed as part of the program.

There are three types of comments in Java:

- Single-Line Comments: Single-line comments start with // and continue until the end of the line.
- Multi-Line Comments: Multi-line comments start with /* and end with *. They can span multiple lines.
- Javadoc Comments: Javadoc comments start with /** and end with *. They are used to generate documentation from the source code.

Comments are an essential part of writing maintainable code, as they help other developers understand the purpose and functionality of the code.

1.3.5 Identifiers

Identifiers are used to name classes, methods, variables, and other elements in a Java program. Identifiers must follow certain rules and conventions:

- An identifier can only contain letters, digits, underscores (_), and dollar signs ($).
- An identifier must start with a letter, underscore, or dollar sign.
- Identifiers are case-sensitive, so uppercase and lowercase letters are considered different.
- Identifiers should be descriptive and follow the naming conventions of the Java programming language.

Good identifiers help make the code more readable and maintainable by providing meaningful names for the elements in the program.

```
1  // Valid Identifiers
2  int number;
3  String firstName;
4  double averageScore;
5  boolean isComplete;
6  void printMessage();
7
8  // Invalid Identifiers
9  int 1number; // Cannot start with a digit
10 String first-name; // Cannot contain hyphens
11 double average score; // Cannot contain spaces
12 boolean isComplete?; // Cannot contain special characters
13 void print Message(); // Cannot contain spaces
```

Code 1.10: Valid and Invalid Identifiers

The code above demonstrates valid and invalid identifiers in Java. Valid identifiers follow the rules and conventions of the Java programming language, while invalid identifiers violate these rules.

1.3.6 Keywords

Keywords are reserved words in Java that have special meanings and cannot be used as identifiers. Keywords are an essential part of the Java programming language and are used to define the syntax and structure of Java programs.

The following are the keywords in Java:

- abstract
- assert
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- default
- do
- double
- else
- enum
- extends
- final
- finally
- float
- for
- goto
- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- package
- private
- protected
- public
- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- try
- void
- volatile
- while

Keywords are an essential part of the Java programming language and are used to define the syntax and structure of Java programs. They have special meanings and cannot be used as identifiers.

1.3.7 Literals

Literals are fixed values that are used in Java programs. There are several types of literals in Java:

- Integer Literals: Integer literals are whole numbers without a decimal point.
- Floating-Point Literals: Floating-point literals are numbers with a decimal point or an exponent.
- Character Literals: Character literals are single characters enclosed in single quotes.
- String Literals: String literals are sequences of characters enclosed in double quotes.
- Boolean Literals: Boolean literals are either true or false.
- Null Literal: The null literal represents the absence of a value.

Literals are used to represent fixed values in Java programs and are an essential part of writing code.

```
1  // Integer Literal
2  int integerLiteral = 10;
3  System.out.println("Integer Literal: " + integerLiteral);
4
5  // Floating-Point Literal
6  double floatingPointLiteral = 3.14;
7  System.out.println("Floating-Point Literal: " + floatingPointLiteral);
8
9  // Character Literal
10 char characterLiteral = 'A';
```

```
System.out.println("Character Literal: " + characterLiteral);

// String Literal
String stringLiteral = "Hello, World!";
System.out.println("String Literal: " + stringLiteral);

// Boolean Literal
boolean booleanLiteral = true;
System.out.println("Boolean Literal: " + booleanLiteral);

// Null Literal
Object nullLiteral = null;
System.out.println("Null Literal: " + nullLiteral);
```

Code 1.11: Literals in Java

The code above demonstrates the use of literals in Java. The code declares variables of different types and assigns them literal values. The code then prints the variables to the console.

1.3.8 Separators

Separators are used to separate tokens in a Java program. The following are the separators in Java:

- Parentheses: ( )
- Braces: { }
- Brackets: [ ]
- Semicolon: ;
- Comma: ,
- Period: .
- Ellipsis: ...

Separators are an essential part of the Java syntax and are used to define the structure of Java programs.

```
public class Separators {
  public static void main(String[] args) {
    // Using Parentheses to Group Expressions
    int result1 = (10 + 5) * 2;
    System.out.println("Result 1: " + result1);

    // Using Braces to Define
    if (result1 > 10) {
      System.out.println("Result 1 is greater than 10");
    } else {
      System.out.println("Result 1 is less than or equal to 10");
    }

    // Using Brackets to Access Array Elements
    int[] numbers = { 1, 2, 3, 4, 5 };
    System.out.println("First Element: " + numbers[0]);
```

```
17
18      // Using Semicolon to Terminate Statements
19      int x = 10;
20      int y = 20;
21      int z = x + y;
22      System.out.println("Sum: " + z);
23
24      // Using Comma to Separate Variables
25      int a = 1, b = 2, c = 3;
26      System.out.println("Values: " + a + ", " + b + ", " + c);
27
28      // Using Period to Access Members
29      String text = "Hello, World!";
30      int length = text.length();
31      System.out.println("Length: " + length);
32
33      // Using Ellipsis to Indicate Variable Arguments
34      int sum = add(1, 2, 3, 4, 5);
35      System.out.println("Sum: " + sum);
36    }
37
38    /**
39    * Method to Calculate the Sum of Variable Arguments
40    *
41    * @param numbers the variable arguments to be added
42    * @return the sum of the variable arguments
43    */
44    public static int add(int... numbers) {
45      int sum = 0;
46      for (int number : numbers) {
47        sum += number;
48      }
49      return sum;
50    }
51  }
```

Code 1.12: Separators in Java

The code above demonstrates the use of separators in Java. The code uses parentheses, braces, brackets, semicolons, commas, periods, and ellipses to define the structure of the program and separate tokens.

Parentheses are used to group expressions and define the order of operations in Java programs. Braces are used to define blocks of code, such as classes, methods, and control structures. Brackets are used to access elements in arrays. Semicolons are used to terminate statements. Commas are used to separate variables. Periods are used to access members of classes and objects. Ellipses are used to indicate variable arguments in methods or by spreading an array into individual elements.

1.3.9 Operators

Operators are used to perform operations on operands in a Java program. There are several types of operators in Java:

- Arithmetic Operators: Arithmetic operators are used to perform mathematical operations, such as addition, subtraction, multiplication, and division.
- Relational Operators: Relational operators are used to compare values and determine the relationship between them, such as equality, inequality, and greater than or less than.
- Logical Operators: Logical operators are used to combine multiple conditions and determine the truth value of a compound expression.
- Assignment Operators: Assignment operators are used to assign values to variables.
- Bitwise Operators: Bitwise operators are used to perform bitwise operations on integer operands.
- Conditional Operator: The conditional operator is used to evaluate a boolean expression and return one of two values based on the result.
- instanceof Operator: The instanceof operator is used to test if an object is an instance of a particular class.
- Unary Operators: Unary operators are used to perform operations on a single operand.
- Shift Operators: Shift operators are used to shift the bits of an integer value to the left or right.
- Ternary Operator: The ternary operator is a shorthand form of the if-else statement.

1.3.9.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on operands. The following are the arithmetic operators in Java:

- Addition (+): Adds two operands.
- Subtraction (-): Subtracts the second operand from the first.
- Multiplication (*): Multiplies two operands.
- Division (/): Divides the first operand by the second.
- Modulus (%): Returns the remainder of the division of the first operand by the second.

```
int operand1 = 10;
int operand2 = 5;
int sum1 = operand1 + operand2;
int difference = operand1 - operand2;
int product = operand1 * operand2;
int quotient = operand1 / operand2;
int remainder = operand1 % operand2;

System.out.println("Sum: " + sum1);
System.out.println("Difference: " + difference);
System.out.println("Product: " + product);
System.out.println("Quotient: " + quotient);
System.out.println("Remainder: " + remainder);

// Output
// Sum: 15
// Difference: 5
// Product: 50
```

```
// Quotient: 2
// Remainder: 0
```

Code 1.13: Arithmetic Operators

1.3.9.2 Relational Operators

Relational operators are used to compare values and determine the relationship between them. The following are the relational operators in Java:

- Equal to (==): Checks if two operands are equal.
- Not equal to (!=): Checks if two operands are not equal.
- Greater than (>): Checks if the first operand is greater than the second.
- Less than (<): Checks if the first operand is less than the second.
- Greater than or equal to (>=): Checks if the first operand is greater than or equal to the second.
- Less than or equal to (<=): Checks if the first operand is less than or equal to the second.

```
boolean isEqual = operand1 == operand2;
boolean isNotEqual = operand1 != operand2;
boolean isGreater = operand1 > operand2;
boolean isLess = operand1 < operand2;
boolean isGreaterOrEqual = operand1 >= operand2;
boolean isLessOrEqual = operand1 <= operand2;

System.out.println("Is Equal: " + isEqual);
System.out.println("Is Not Equal: " + isNotEqual);
System.out.println("Is Greater: " + isGreater);
System.out.println("Is Less: " + isLess);
System.out.println("Is Greater or Equal: " + isGreaterOrEqual);
System.out.println("Is Less or Equal: " + isLessOrEqual);

// Output
// Is Equal: false
// Is Not Equal: true
// Is Greater: true
// Is Less: false
// Is Greater or Equal: true
// Is Less or Equal: false
```

Code 1.14: Relational Operators

1.3.9.3 Logical Operators

Logical operators are used to combine multiple conditions and determine the truth value of a compound expression. The following are the logical operators in Java:

- AND (&&): Returns true if both operands are true.
- OR (||): Returns true if at least one of the operands is true.
- NOT (!): Returns true if the operand is false.

```java
boolean condition1 = true;
boolean condition2 = false;
boolean andResult = condition1 && condition2;
boolean orResult = condition1 || condition2;
boolean notResult1 = !condition1;
boolean notResult2 = !condition2;

System.out.println("AND Result: " + andResult);
System.out.println("OR Result: " + orResult);
System.out.println("NOT Result 1: " + notResult1);
System.out.println("NOT Result 2: " + notResult2);
// Output
// AND Result: false
// OR Result: true
// NOT Result 1: false
// NOT Result 2: true
```

Code 1.15: Logical Operators

1.3.9.4 Assignment Operators

Assignment operators are used to assign values to variables. The following are the assignment operators in Java:

- Assignment (=): Assigns the value of the right operand to the left operand.
- Addition Assignment (+=): Adds the value of the right operand to the left operand and assigns the result to the left operand.
- Subtraction Assignment (-=): Subtracts the value of the right operand from the left operand and assigns the result to the left operand.
- Multiplication Assignment (*=): Multiplies the value of the right operand with the left operand and assigns the result to the left operand.
- Division Assignment (/=): Divides the value of the left operand by the right operand and assigns the result to the left operand.
- Modulus Assignment (%=): Computes the remainder of the division of the left operand by the right operand and assigns the result to the left operand.

```java
int value = 10;
value += 5;
value -= 3;
value *= 2;
value /= 4;
value %= 3;

// Output
// Value: 0
```

Code 1.16: Assignment Operators

1.3.9.5 Bitwise Operators

Bitwise operators are used to perform bitwise operations on integer operands. The following are the bitwise operators in Java:

- AND (&): Performs a bitwise AND operation on the operands.
- OR (|): Performs a bitwise OR operation on the operands.
- XOR (^): Performs a bitwise XOR operation on the operands.
- Complement (~): Performs a bitwise complement operation on the operand.
- Left Shift («): Shifts the bits of the operand to the left by a specified number of positions.
- Right Shift (»): Shifts the bits of the operand to the right by a specified number of positions.

```
int operand3 = 5;
int operand4 = 3;
int andResult1 = operand3 & operand4;
int orResult1 = operand3 | operand4;
int xorResult = operand3 ^ operand4;
int complementResult = ~operand3;
int leftShiftResult = operand3 << 1;
int rightShiftResult = operand3 >> 1;

System.out.println("AND Result 1: " + andResult1);
System.out.println("OR Result 1: " + orResult1);
System.out.println("XOR Result: " + xorResult);
System.out.println("Complement Result: " + complementResult);
System.out.println("Left Shift Result: " + leftShiftResult);
System.out.println("Right Shift Result: " + rightShiftResult);

// Output
// AND Result 1: 1
// OR Result 1: 7
// XOR Result: 6
// Complement Result: -6
// Left Shift Result: 10
// Right Shift Result: 2
```

Code 1.17: Bitwise Operators

1.3.9.6 Conditional Operators

The conditional operator is used to evaluate a boolean expression and return one of two values based on the result. The conditional operator has the following syntax:

- condition ? value1 : value2

If the condition is true, the value of value1 is returned. If the condition is false, the value of value2 is returned.

```
int number1 = 10;
int number2 = 20;
```

```
int max = (number1 > number2) ? number1 : number2;

System.out.println("Max: " + max);

// Output
// Max: 20
```

Code 1.18: Conditional Operators

1.3.9.7 instanceof Operators

The instanceof operator is used to test if an object is an instance of a particular class. The instanceof operator has the following syntax:

- object instanceof class

If the object is an instance of the specified class or one of its subclasses, the instanceof operator returns true. Otherwise, it returns false.

```
Object object = new Object();
boolean isInstance = object instanceof Object;

System.out.println("Is Instance: " + isInstance);

// Output
// Is Instance: true
```

Code 1.19: instanceof Operators

1.3.9.8 Unary Operators

Unary operators are used to perform operations on a single operand. The following are the unary operators in Java:

- Unary Plus (+): Returns the value of the operand.
- Unary Minus (-): Negates the value of the operand.
- Increment (++): Increments the value of the operand by 1.
- Decrement (–): Decrements the value of the operand by 1.
- Logical Complement (!): Returns the logical complement of the operand.

```
int operand5 = 10;
int increment = ++operand5;
int decrement = --operand5;
boolean isTrue = true;
boolean notResult3 = !isTrue;

System.out.println("Increment: " + increment);
System.out.println("Decrement: " + decrement);
System.out.println("NOT Result 3: " + notResult3);

```

```
// Output
// Increment: 11
// Decrement: 10
// NOT Result 3: false
```

Code 1.20: Unary Operators

1.3.9.9 Shift Operators

Shift operators are used to shift the bits of an integer value to the left or right. The following are the shift operators in Java:

- Left Shift («): Shifts the bits of the operand to the left by a specified number of positions.
- Right Shift (»): Shifts the bits of the operand to the right by a specified number of positions.

```
int operand6 = 5;
int leftShiftResult1 = operand6 << 1;
int rightShiftResult1 = operand6 >> 1;

System.out.println("Left Shift Result 1: " + leftShiftResult1);
System.out.println("Right Shift Result 1: " + rightShiftResult1);

// Output
// Left Shift Result 1: 10
// Right Shift Result 1: 2
```

Code 1.21: Shift Operators

1.3.9.10 Ternary Operators

The ternary operator is a shorthand form of the if-else statement. The ternary operator has the following syntax:

- condition ? value1 : value2

If the condition is true, the value of value1 is returned. If the condition is false, the value of value2 is returned.

```
int number3 = 10;
int number4 = 20;
int min = (number3 < number4) ? number3 : number4;

System.out.println("Min: " + min);

// Output
// Min: 10
```

Code 1.22: Ternary Operators

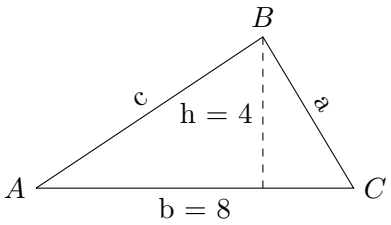
1.3.10 Summary

The lexical structure of Java defines the set of valid tokens that can be used to write programs in the language. The lexical structure includes elements such as Unicode, lexical translations, Unicode escapes, line terminators, input elements and tokens, white space, comments, identifiers, keywords, literals, separators, and operators.

1.3.11 Coding Exercises

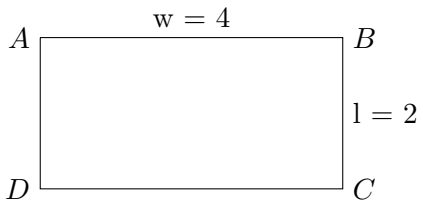
1. Write a program that prints "Hello, World!" to the console.

2. Write a program that calculates the area of a triangle given the base and height.



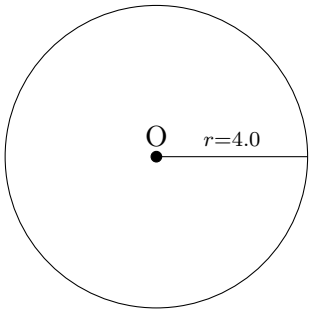
$$b = 8, \text{ the base of the triangle} \tag{1.1}$$
$$h = 4, \text{ the height of the triangle} \tag{1.2}$$
$$A = \frac{b \cdot h}{2}, \text{ formula for the area of a triangle} \tag{1.3}$$

3. Write a program that calculates the area of a rectangle given the length and width.



$$l = 2, \text{ the length of the rectangle} \tag{1.4}$$
$$w = 4, \text{ the width of the rectangle} \tag{1.5}$$
$$A = l \cdot w, \text{ formula for the area of a rectangle} \tag{1.6}$$

4. Write a program that calculates the area of a circle given the radius.



$$r = 4.0, \text{ the radius of the circle} \tag{1.7}$$
$$\pi = 3.141592653589793, \text{ the mathematical constant pi} \tag{1.8}$$
$$A = \pi \cdot r^2, \text{ formula for the area of a circle} \tag{1.9}$$

1.4 Data Types

Data types specify the different sizes and values that can be stored in a variable. There are two types of data types in Java:

- Primitive Data Types: These data types are predefined by the language and are named by a reserved keyword. They are used to store simple values.
- Non-Primitive Data Types: These data types are not predefined by the language and are created by the programmer. They are also known as reference types because they refer to objects.

1.4.1 Primitive Data Types

Primitive data types are predefined by the language and are named by a reserved keyword. They are used to store simple values. There are eight primitive data types in Java:

Table 1.1: Primitive Data Types in Java

Data Type	Bytes	Description
byte	1	Stores whole numbers from -128 to 127
short	2	Stores whole numbers from -32,768 to 32,767
int	4	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8	Stores fractional numbers. Sufficient for storing 15 decimal digits
char	2	Stores a single character/letter or ASCII values
boolean	1	Stores true or false values

The table above shows the eight primitive data types in Java, along with their sizes in bytes and descriptions.

1.4.1.1 byte

The “byte” data type is an 8-bit signed integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The “byte” data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.

```
1  byte byteVariable = 100;
```

Code 1.23: Byte Data Type

1.4.1.2 short

The “short” data type is a 16-bit signed integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). The “short” data type is used to save memory in large arrays, mainly in place of integers, since a short is two times smaller than an int.

```
short shortVariable = 1000;
```

Code 1.24: Short Data Type

1.4.1.3 int

The “int” data type is a 32-bit signed integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). The “int” data type is generally used as the default data type for integral values unless there is a concern about memory.

```
int intVariable = 100000;
```

Code 1.25: Int Data Type

1.4.1.4 long

The “long” data type is a 64-bit signed integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). The “long” data type is used when you need a range of values more than those provided by int.

```
long longVariable = 100000000L;
```

Code 1.26: Long Data Type

To indicate that a value is of the “long” data type, you must append an “L” to the value. The “L” tells the compiler that the value is a long literal. If you do not append an “L” to the value, the compiler will treat it as an int literal. This can lead to a compilation error if the value is too large to be represented as an int.

1.4.1.5 float

The “float” data type is a single-precision 32-bit IEEE 754 floating-point. It should never be used for precise values, such as currency. This is because it will result in a loss of precision. The “float” data type is used when you need a fractional value with a large range.

```
float floatVariable = 234.5f;
```

Code 1.27: Float Data Type

To indicate that a value is of the “float” data type, you must append an “f” to the value. The “f” tells the compiler that the value is a float literal. If you do not append an “f” to the value, the compiler will treat it as a double literal.

1.4.1.6 double

The “double” data type is a double-precision 64-bit IEEE 754 floating-point. It can store fractional values with a large range. The “double” data type is used when you need a fractional value with a large range.

```
double doubleVariable = 123.4;
```

Code 1.28: Double Data Type

Unlike the “float” data type, the “double” data type does not require any appending characters to indicate that a value is of the “double” data type. The compiler will treat any value with a decimal point as a double literal.

1.4.1.7 char

The “char” data type is a single 16-bit Unicode character. It has a minimum value of \u0000 (or 0) and a maximum value of \u0041 (or 65,535 inclusive).

```
char charVariable = 'A';
```

Code 1.29: Char Data Type

When you assign a character literal to a char variable, you must enclose the character in single quotes. Character literals are enclosed in single quotes, such as 'A' or '7'. If you enclose a character in double quotes, it will be treated as a string literal.

```
char charVariable = 65;
```

Code 1.30: Char Data Type using ASCII Value

A number from 0 to 65535 can also be used to represent a character. For example, 65 represents the character 'A', and 97 represents the character 'a'. This is because the ASCII value of 'A' is 65, and the ASCII value of 'a' is 97.

```
char charVariable = '\u0041';
```

Code 1.31: Char Data Type using Unicode Escape

You can also use Unicode escape sequences to represent characters. For example, '\u0041' represents the character 'A' in the Latin alphabet.

1.4.1.8 boolean

The “boolean” data type represents one bit of information, but its “size” isn't precisely defined. It can only take the values true or false.

```
boolean booleanVariable = true;
```

Code 1.32: Boolean Data Type

Aside from representing true or false, a boolean can also represent values with two states, such as on or off, yes or no, or 0 or 1.

1.4.2 Non-Primitive Data Types

Non-primitive data types are not predefined by the language and are created by the programmer. They are also known as reference types because they refer to objects. There are two types of non-primitive data types:

- Reference Data Types: Reference variables are created using defined classes. They are used to access objects.
- Array Data Types: Array data types are used to store multiple values in a single variable.

1.4.2.1 Reference Data Types

Reference variables are created using defined classes. They are used to access objects.

1.4.2.1.1 String

The "String" class represents a sequence of characters. Strings are immutable, which means they cannot be changed after they are created.

```
String stringVariable = new String("Hello, World!");
```

Code 1.33: String Data Type

1.4.2.1.2 ArrayList

The "ArrayList" class is a resizable array that implements the List interface. It allows you to add, remove, and access elements based on their index.

```
List<String> arrayListVariable = new ArrayList<String>();
arrayListVariable.add("Apple");
arrayListVariable.add("Banana");
```

Code 1.34: ArrayList Data Type

1.4.2.1.3 HashMap

The "HashMap" class is a hash table-based implementation of the Map interface. It allows you to store key-value pairs and access the values based on their keys.

```
Map<String, String> hashMapVariable = new HashMap<String, String>();
hashMapVariable.put("Key1", "Value1");
hashMapVariable.put("Key2", "Value2");
```

Code 1.35: HashMap Data Type

1.4.2.1.4 HashSet

The "HashSet" class is an implementation of the Set interface. It stores unique elements and does not allow duplicates.

```
Set<String> hashSetVariable = new HashSet<String>();
hashSetVariable.add("Apple");
hashSetVariable.add("Banana");
```

Code 1.36: HashSet Data Type

1.4.2.1.5 LinkedList

The "LinkedList" class is a doubly-linked list implementation of the List and Deque interfaces. It allows you to add, remove, and access elements based on their index.

```
List<String> linkedListVariable = new LinkedList<String>();
linkedListVariable.add("Apple");
linkedListVariable.add("Banana");
```

Code 1.37: LinkedList Data Type

1.4.2.1.6 Queue

The "Queue" interface represents a collection of elements that can be added or removed in a specific order. The "LinkedList" class implements the Queue interface.

```
Queue<String> queueVariable = new LinkedList<String>();
queueVariable.add("Apple");
queueVariable.add("Banana");
```

Code 1.38: Queue Data Type

1.4.2.1.7 Stack

The "Stack" class represents a last-in, first-out (LIFO) stack of elements. It extends the Vector class with five operations that allow a vector to be treated as a stack.

```
Stack<String> stackVariable = new Stack<String>();
stackVariable.push("Apple");
stackVariable.push("Banana");
```

Code 1.39: Stack Data Type

1.4.2.1.8 TreeMap

The "TreeMap" class is a Red-Black tree-based implementation of the Map interface. It provides an efficient means of storing key-value pairs in sorted order.

```
Map<String, String> treeMapVariable = new TreeMap<String, String>();
treeMapVariable.put("Key1", "Value1");
treeMapVariable.put("Key2", "Value2");
```

Code 1.40: TreeMap Data Type

1.4.2.1.9 TreeSet

The "TreeSet" class is an implementation of the Set interface. It stores unique elements in sorted order.

```
Set<String> treeSetVariable = new TreeSet<String>();
treeSetVariable.add("Apple");
treeSetVariable.add("Banana");
```

Code 1.41: TreeSet Data Type

1.4.2.2 Array Data Types

Array data types are used to store multiple values in a single variable. To create an array, you must specify the data type of the elements and the size of the array. To access an element in an array, you must use the index of the element.

```
int[] intArrayVariable = new int[5];
intArrayVariable[0] = 10;
intArrayVariable[1] = 20;
intArrayVariable[2] = 30;
intArrayVariable[3] = 40;
intArrayVariable[4] = 50;

String[] stringArrayVariable = new String[2];
stringArrayVariable[0] = "Hello";
stringArrayVariable[1] = "World";

System.out.print("Int Array: ");
for (int i = 0; i < intArrayVariable.length; i++) {
  System.out.print(intArrayVariable[i] + " ");
}
System.out.println();

System.out.print("String Array: ");
for (int i = 0; i < stringArrayVariable.length; i++) {
  System.out.print(stringArrayVariable[i] + " ");
}
System.out.println();

// Output
// Int Array: 10 20 30 40 50
```

Code 1.42: Array Data Types

The code above demonstrates the use of array data types in Java. The code creates two arrays, one for integers and one for strings. It assigns values to the elements of the arrays and then prints the arrays to the console.

1.4.3 Summary

Data types specify the different sizes and values that can be stored in a variable. There are two types of data types in Java: primitive data types and non-primitive data types.

Primitive data types are predefined by the language and are used to store simple values. There are eight primitive data types in Java: byte, short, int, long, float, double, char, and boolean.

Non-primitive data types are not predefined by the language and are created by the programmer. They are used to access objects and store multiple values in a single variable. Non-primitive data types include reference data types and array data types.

1.4.4 Coding Exercises

1. Create a Java program that declares and initializes a variable of the following data types: byte, short, int, long, float, double, char, boolean, String, and an array of integers. Print the values of the variables to the console.

1.5 Basic Object-Oriented Programming Concepts

Object-Oriented Programming (OOP) is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.

The main principles of OOP are:

- Encapsulation: Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- Inheritance: Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- Polymorphism: Polymorphism is the ability of an object to take on many forms.
- Abstraction: Abstraction is a concept of object-oriented programming that allows hiding the implementation details and showing only the functionality to the user.

In Object-Oriented Programming (OOP), a class is a blueprint for creating objects. A class provides initial values for state (member variables or attributes) and implementations of behavior (member functions or methods). A class is a user-defined data type that can be used in a program and works as an object constructor or a blueprint for creating objects.

```
1  // Animal.java
2
3  public class Animal {
4      String name;
5      String species;
6      int age;
7      int weight;
8      String color;
9      String habitat;
10
```

```
    int x;
    int y;

    public Animal(String name, String species, int age, int weight, String
        color, String habitat, int x, int y) {
        this.name = name;
        this.species = species;
        this.age = age;
        this.weight = weight;
        this.color = color;
        this.habitat = habitat;
        this.x = x;
        this.y = y;
    }

    void makeSound() {
        System.out.println(this.name + " makes a sound.");
    }

    void move(int x, int y) {
        this.x = x;
        this.y = y;
        System.out.println("The animal moves to (" + x + ", " + y + ").");
    }

    void eat() {
        System.out.println("The animal eats food.");
    }

    void sleep() {
        System.out.println("The animal sleeps.");
    }
}
```

Code 1.43: Animal Class

The "Animal" class is a blueprint for creating objects of type "Animal". It provides initial values for state (member variables or attributes) and implementations of behavior (member functions or methods).

The syntax for creating a class in Java is:

```
public class ClassName {
    // member variables or attributes
    // member functions or methods
}
```

Code 1.44: Class Syntax

```
public class Animal {
    String name;
    String species;
    int age;
    int weight;
    String color;
    String habitat;
    int x;
    int y;

    // Rest of the code...
}
```

Code 1.45: Class Attributes

The "Animal" class has member variables or attributes that represent the state of an animal, such as its name, species, age, weight, color, habitat, and position (x, y). These attributes represents the real-world properties of an animal.

```
public class Animal {
    // Member variables or attributes...

    public Animal(String name, String species, int age, int weight, String
        color, String habitat, int x, int y) {
        this.name = name;
        this.species = species;
        this.age = age;
        this.weight = weight;
        this.color = color;
        this.habitat = habitat;
        this.x = x;
        this.y = y;
    }

    // Rest of the code...
}
```

Code 1.46: Class Constructor

A constructor is a special method that is used to initialize objects. It is called when an object of a class is created. It has the same name as the class and does not have a return type. Constructors are used to set initial values for the member variables of an object.

```
public class Animal {
    // Member variables or attributes...

    // Constructor...

    void makeSound() {
```

```
7           System.out.println(this.name + " makes a sound.");
8       }
9
10      void move(int x, int y) {
11          this.x = x;
12          this.y = y;
13          System.out.println("The animal moves to (" + x + ", " + y + ").");
14      }
15
16      void eat() {
17          System.out.println("The animal eats food.");
18      }
19
20      void sleep() {
21          System.out.println("The animal sleeps.");
22      }
23  }
```

Code 1.47: Class Methods

The "Animal" class has member functions or methods that represent the behavior of an animal, such as making a sound, moving to a new position, eating food, and sleeping. These methods represent the actions that an animal can perform.

1.5.1 Why use Object-Oriented Programming?

Object-oriented programming (OOP) has several advantages over procedural programming. Some of the key benefits of OOP are:

- **Modularity:** OOP programs are divided into objects, which can be developed independently. This makes the code more modular, reusable, and easier to maintain.
- **Code Reusability:** OOP promotes code reusability through inheritance and polymorphism. Classes can be reused in different parts of the program without modification.
- **Encapsulation:** OOP encapsulates the data and methods within a class, which protects the data from being accessed or modified by unauthorized code.
- **Abstraction:** OOP provides abstraction by hiding the implementation details and showing only the functionality to the user. This simplifies the complexity of the program.
- **Inheritance:** OOP allows classes to inherit the properties and methods of other classes, which promotes code reusability and reduces the complexity of the program.
- **Polymorphism:** OOP allows methods to do different things based on the object that they are acting upon. This provides flexibility and code reusability.

OOP is widely used in software development because of its ability to model real-world entities, promote code reusability, and simplify the complexity of the program. It is a powerful paradigm that helps in developing scalable, maintainable, and flexible software applications.

1.5.2 Inheritance

Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object. It allows a class to inherit the properties and methods of another class. The class which inherits the properties and methods is known as the child class, and the class whose properties and methods are inherited is known as the parent class. Inheritance is an important

feature of OOP that allows code reusability and reduces the complexity of a program.

```java
// Mammal.java

public class Mammal extends Animal {
    String fur;
    boolean milk;
    boolean isWarmBlooded;
    boolean canGiveBirth;

    Mammal(String name, String species, int age, int weight, String color,
        String habitat, int x, int y) {
        super(name, species, age, weight, color, habitat, x, y);
    }

    Mammal(String name, String species, int age, int weight, String color,
        String habitat, int x, int y, String fur, boolean milk, boolean
        isWarmBlooded, boolean canGiveBirth) {
        super(name, species, age, weight, color, habitat, x, y);
        this.fur = fur;
        this.milk = milk;
        this.isWarmBlooded = isWarmBlooded;
        this.canGiveBirth = canGiveBirth;
    }

    void giveBirth() {
        if (this.age >= 2 && this.canGiveBirth) {
            System.out.println(this.name + " is giving birth.");
        } else {
            System.out.println(this.name + " is too young to give birth.");
        }
    }
}
```

Code 1.48: Mammal Class (Inheritance)

The “Mammal” class is a child class of the “Animal” class. It inherits the properties and methods of the “Animal” class.

```java
public class Mammal extends Animal {
    String fur;
    boolean milk;
    boolean isWarmBlooded;
    boolean canGiveBirth;

    // Rest of the code...
}
```

Code 1.49: Mammal Class (Member Variables)

The "Mammal" class has member variables or attributes that represent the state of a mammal, such as its fur, milk production, warm-blooded nature, and ability to give birth. These attributes represent the real-world properties of a mammal.

```
public class Mammal extends Animal {
    // Member variables or attributes...

    Mammal(String name, String species, int age, int weight, String color,
        String habitat, int x, int y) {
        super(name, species, age, weight, color, habitat, x, y);
    }

    Mammal(String name, String species, int age, int weight, String color,
        String habitat, int x, int y, String fur, boolean milk, boolean
        isWarmBlooded, boolean canGiveBirth) {
        super(name, species, age, weight, color, habitat, x, y);
        this.fur = fur;
        this.milk = milk;
        this.isWarmBlooded = isWarmBlooded;
        this.canGiveBirth = canGiveBirth;
    }

    // Rest of the code...
}
```

Code 1.50: Mammal Class (Constructors)

The "Mammal" class has two constructors that initialize the member variables of the "Mammal" class as well as the "Animal" class. Having multiple constructors in a class is called method overloading.

```
Mammal(String name, String species, int age, int weight, String color,
    String habitat, int x, int y) {
    super(name, species, age, weight, color, habitat, x, y);
}
```

Code 1.51: Mammal Class (First Constructor)

The first constructor initializes the member variables of the "Mammal" class. This constructor calls the constructor of the parent class ("Animal") using the "super" keyword.

```
Mammal(String name, String species, int age, int weight, String color,
    String habitat, int x, int y, String fur, boolean milk, boolean
    isWarmBlooded, boolean canGiveBirth) {
    super(name, species, age, weight, color, habitat, x, y);
    this.fur = fur;
    this.milk = milk;
    this.isWarmBlooded = isWarmBlooded;
    this.canGiveBirth = canGiveBirth;
```

```
}
```

Code 1.52: Mammal Class (Second Constructor)

The second constructor initializes the member variables of the "Mammal" class as well as the "Animal" class. This constructor allows additional properties of a mammal to be set, such as fur, milk production, warm-blooded nature, and ability to give birth.

```
public class Mammal extends Animal {
    // Member variables or attributes...

    // Constructors...

    void giveBirth() {
        if (this.age >= 2 && this.canGiveBirth) {
            System.out.println(this.name + " is giving birth.");
        } else {
            System.out.println(this.name + " is too young to give birth.");
        }
    }
}
```

Code 1.53: Mammal Class (Methods)

The "Mammal" class has a method called "giveBirth()" that makes the mammal give birth. This method is specific to the "Mammal" class and is not present in the "Animal" class.

1.5.3 Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.

Encapsulation is used to:

- Control the way data is accessed or modified
- Prevent data from being modified by unauthorized code
- Allow data to be accessed only through getter methods
- Protect the integrity of the data

Encapsulation is an important concept in OOP that helps in maintaining the integrity of the data and the code that manipulates it.

```
// Dog.java

public class Dog extends Mammal {
    private String breed;
    private String barkingSound;
    private boolean tailWagging;
    private boolean isTrained;
```

```
8
9       Dog(String name, String species, int age, int weight, String color,
            String habitat, int x, int y, String fur, boolean milk, boolean
            isWarmBlooded, boolean canGiveBirth, String breed, String
            barkingSound, boolean tailWagging, boolean isTrained) {
10          super(name, species, age, weight, color, habitat, x, y, fur, milk,
                isWarmBlooded, canGiveBirth);
11          this.breed = breed;
12          this.barkingSound = barkingSound;
13          this.tailWagging = tailWagging;
14          this.isTrained = isTrained;
15      }
16
17      public String getBreed() {
18          return breed;
19      }
20
21      public void setBreed(String breed) {
22          this.breed = breed;
23      }
24
25      public String getBarkingSound() {
26          return barkingSound;
27      }
28
29      public void setBarkingSound(String barkingSound) {
30          this.barkingSound = barkingSound;
31      }
32
33      public boolean isTailWagging() {
34          return tailWagging;
35      }
36
37      public void setTailWagging(boolean tailWagging) {
38          this.tailWagging = tailWagging;
39      }
40
41      public boolean isTrained() {
42          return isTrained;
43      }
44
45      public void setTrained(boolean isTrained) {
46          this.isTrained = isTrained;
47      }
48
49      public void bark() {
50          System.out.println("The dog barks: " + barkingSound);
51      }
52
53      public void fetch() {
54          if (this.isTrained) {
```

```
            System.out.println("The dog fetches the ball.");
        } else {
            System.out.println("The dog is not trained to fetch.");
        }
    }

    public void wagTail() {
        if (this.tailWagging) {
            System.out.println("The dog wags its tail.");
        } else {
            System.out.println("The dog is not wagging its tail.");
        }
    }

    public void sit() {
        if (this.isTrained) {
            System.out.println("The dog sits.");
        } else {
            System.out.println("The dog is not trained to sit.");
        }
    }
}
```

Code 1.54: Dog Class(Encapsulation)

The "Dog" class is a child class of the "Mammal" class. It inherits the properties and methods of the "Mammal" class. The "Dog" class also inherits the "Animal" class through the "Mammal" class. The "Dog" class demonstrates the concept of encapsulation by using private member variables and providing public getter and setter methods to access and modify the private member variables.

```
public class Dog extends Mammal {
    // Member variables or attributes...

    // Constructor...

    public String getBreed() {
        return breed;
    }

    public void setBreed(String breed) {
        this.breed = breed;
    }

    public String getBarkingSound() {
        return barkingSound;
    }

    public void setBarkingSound(String barkingSound) {
        this.barkingSound = barkingSound;
```

```
    }

    public boolean isTailWagging() {
        return tailWagging;
    }

    public void setTailWagging(boolean tailWagging) {
        this.tailWagging = tailWagging;
    }

    public boolean isTrained() {
        return isTrained;
    }

    public void setTrained(boolean isTrained) {
        this.isTrained = isTrained;
    }

    // Rest of the code...
}
```

Code 1.55: Dog Class (Getter and Setter Methods)

The "Dog" class has private member variables that represent the state of a dog, such as its breed, barking sound, tail-wagging behavior, and training status. These member variables are private to the class, which means they cannot be accessed or modified directly from outside the class. To access or modify these private member variables, the "Dog" class provides public getter and setter methods.

```
public String getBreed() {
    return breed;
}

public String getBarkingSound() {
    return barkingSound;
}

public boolean isTailWagging() {
    return tailWagging;
}

public boolean isTrained() {
    return isTrained;
}
```

Code 1.56: Dog Class (Getter Methods)

The getter methods are used to access the value of a private member variable. They return the value of the private member variable to the caller. The getter methods are public, which means they can be accessed from outside the class. They provide read-only access to the private

member variables. For convention, the getter methods are named with the prefix "get" followed by the name of the member variable.

Aside for its read-only access, the getter methods are also used to perform additional operations, such as validation or computation, before returning the value of the private member variable. This allows the class to maintain the integrity of the data and provide a consistent interface to the caller.

```
public void setBreed(String breed) {
    this.breed = breed;
}

public void setBarkingSound(String barkingSound) {
    this.barkingSound = barkingSound;
}

public void setTailWagging(boolean tailWagging) {
    this.tailWagging = tailWagging;
}

public void setTrained(boolean isTrained) {
    this.isTrained = isTrained;
}
```

Code 1.57: Dog Class (Setter Methods)

The setter methods are used to set the value of a private member variable. They assign a new value to the private member variable based on the input provided by the caller. The setter methods are public, which means they can be accessed from outside the class. They provide write-only access to the private member variables. For convention, the setter methods are named with the prefix "set" followed by the name of the member variable.

Aside for its write-only access, the setter methods are also used to perform additional operations, such as validation or computation, before setting the value of the private member variable. For example, in a website where users have different roles, the setter method can be used to check if the user has the necessary permissions to set the value of a private member variable.

1.5.4 Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Polymorphism allows methods to do different things based on the object that they are acting upon. It is a powerful feature of OOP that allows code reusability and flexibility.

```
// Bird.java

public class Bird extends Animal {
    private int wings;
    private boolean hasFeathers;
    private boolean canFly;
```

```
7       private boolean canLayEggs;
8
9       Bird(String name, String species, int age, int weight, String color,
            String habitat, int x, int y, int wings, boolean hasFeathers,
            boolean canFly, boolean canLayEggs) {
10          super(name, species, age, weight, color, habitat, x, y);
11          this.wings = wings;
12          this.hasFeathers = hasFeathers;
13          this.canFly = canFly;
14          this.canLayEggs = canLayEggs;
15      }
16
17      public int getWings() {
18          return wings;
19      }
20
21      public void setWings(int wings) {
22          this.wings = wings;
23      }
24
25      public boolean isHasFeathers() {
26          return hasFeathers;
27      }
28
29      public void setHasFeathers(boolean hasFeathers) {
30          this.hasFeathers = hasFeathers;
31      }
32
33      public boolean canFly() {
34          return canFly;
35      }
36
37      public void setCanFly(boolean canFly) {
38          this.canFly = canFly;
39      }
40
41      public boolean canLayEggs() {
42          return canLayEggs;
43      }
44
45      public void setCanLayEggs(boolean canLayEggs) {
46          this.canLayEggs = canLayEggs;
47      }
48
49      public void fly() {
50          if (canFly && hasFeathers) {
51              System.out.println(name + " is flying.");
52          } else {
53              System.out.println(name + " cannot fly.");
54          }
55      }
```

```
56
57      public void layEggs() {
58          if (this.age >= 1 && canLayEggs) {
59              System.out.println(name + " is laying eggs.");
60          } else {
61              System.out.println(name + " is too young to lay eggs.");
62          }
63      }
64
65      @Override
66      void makeSound() {
67          System.out.println(this.name + " chirps.");
68      }
69  }
```

Code 1.58: Bird Class (Polymorphism)

The “Bird” class is a child class of the “Animal” class. It inherits the properties and methods of the “Animal” class. The “Bird” class demonstrates the concept of polymorphism by overriding the “makeSound()” method of the “Animal” class.

```
1  @Override
2  void makeSound() {
3      System.out.println(this.name + " chirps.");
4  }
```

Code 1.59: Bird Class (Overriding Methods)

The “Bird” class overrides the “makeSound()” method of the “Animal” class. The “makeSound()” method makes the bird chirp. This method is specific to the “Bird” class and is not present in the “Animal” class.

```
1  // Animal.java (makeSound() method)
2  void makeSound() {
3      System.out.println(this.name + " makes a sound.");
4  }
```

Code 1.60: Animal Class (makeSound() Method)

For objects of the “Bird” class, the “makeSound()” method will output “chirps” instead of the default “makes a sound” output of the “Animal” class. This is an example of polymorphism in action, where the same method name is used in both the parent and child classes, but the behavior of the method is different based on the object that it is acting upon.

1.5.5 Abstraction

Abstraction is a concept of object-oriented programming that allows hiding the implementation details and showing only the functionality to the user. It helps to reduce programming complexity and effort.

An abstract class is a class that cannot be instantiated. It is used to provide a common interface for all the classes that inherit from it. Abstract classes can have abstract methods that are declared but not implemented. These methods must be implemented by the child classes that inherit from the abstract class.

```java
// Reptile.java

public abstract class Reptile extends Animal {
    private boolean hasScales;
    private boolean hasTail;
    private boolean isColdBlooded;
    private boolean canRegenerateTail;
    private boolean canLayEggs;

    Reptile(String name, String species, int age, int weight, String color,
        String habitat, int x, int y, boolean hasScales, boolean hasTail,
        boolean isColdBlooded, boolean canRegenerateTail, boolean
        canLayEggs) {
        super(name, species, age, weight, color, habitat, x, y);
        this.hasScales = hasScales;
        this.hasTail = hasTail;
        this.isColdBlooded = isColdBlooded;
        this.canRegenerateTail = canRegenerateTail;
        this.canLayEggs = canLayEggs;
    }

    public abstract void biteHuman();

    public boolean hasScales() {
        return hasScales;
    }

    public void setHasScales(boolean hasScales) {
        this.hasScales = hasScales;
    }

    public boolean isColdBlooded() {
        return isColdBlooded;
    }

    public void setColdBlooded(boolean isColdBlooded) {
        this.isColdBlooded = isColdBlooded;
    }

    public boolean canRegenerateTail() {
        return canRegenerateTail;
    }

    public void setCanRegenerateTail(boolean canRegenerateTail) {
        this.canRegenerateTail = canRegenerateTail;
    }
```

```
44
45      public boolean canLayEggs() {
46          return canLayEggs;
47      }
48
49      public void setCanLayEggs(boolean canLayEggs) {
50          this.canLayEggs = canLayEggs;
51      }
52
53      public void shedSkin() {
54          if (hasScales) {
55              System.out.println(name + " is shedding its skin.");
56          } else {
57              System.out.println(name + " does not have scales and cannot shed
                    its skin.");
58          }
59      }
60
61      public void layEggs() {
62          if (canLayEggs) {
63              System.out.println(name + " is laying eggs.");
64          } else {
65              System.out.println(name + " cannot lay eggs.");
66          }
67      }
68
69      public void regenerateTail() {
70          if (canRegenerateTail && hasTail) {
71              System.out.println(name + " is regenerating its tail.");
72          } else {
73              System.out.println(name + " cannot regenerate its tail.");
74          }
75      }
76  }
```

Code 1.61: Reptile Class (Abstraction)

The “Reptile” class is an abstract class that extends the “Animal” class. It inherits the properties and methods of the “Animal” class and adds additional properties and methods specific to reptiles. Since the “Reptile” class is an abstract class, it cannot be instantiated.

```
1   public abstract class Reptile extends Animal {
2       // Member variables or attributes...
3
4       // Constructor...
5
6       public abstract void biteHuman();
7
8       // Getter and setter methods...
9
```

```
    // Methods...
}
```

Code 1.62: Reptile Class (Abstract Method)

The “Reptile” class has an abstract method called “biteHuman()” that must be implemented by the child classes that inherit from the “Reptile” class. Abstract methods are declared without an implementation and must be implemented by the child classes.

The “Snake” class is a child class of the “Reptile” class which is an abstract class. The “Snake” class inherits the properties and methods of the “Reptile” class and implements the abstract method “biteHuman()”.

```
// Snake.java

public class Snake extends Reptile {
    // Member variables or attributes...

    // Constructor...

    public void biteHuman() {
        System.out.println("The snake bites the human...");
        if (isVenomous) {
            System.out.println("The human is poisoned!");
        } else {
            System.out.println("The human is not poisoned.");
        }
    }

    // Getter and setter methods...

    // Methods...
}
```

Code 1.63: Snake Class (Implementing Abstract Method)

The “Snake” class implements the abstract method “biteHuman()” declared in the “Reptile” class. The “biteHuman()” method simulates the snake biting a human and poisoning the human if the snake is venomous.

1.5.6 Using Classes and Objects

Using the classes and objects defined in the previous sections, we can create instances of the classes and interact with them.

```
// BasicOOP.java

public class BasicOOP {

    public static void main(String[] args) {
```

```
        Animal animal = new Animal("Dog", "Canis lupus familiaris", 5, 20,
            "Brown", "Domestic", 0, 0);

        System.out.println("Name: " + animal.name);
        System.out.println("Species: " + animal.species);
        System.out.println("Age: " + animal.age);

        animal.move(5, 10);

        Mammal mammal = new Mammal("Cat", "Felis catus", 3, 10, "White",
            "Domestic", 0, 0, "Short", true, true, true);
        Bird bird = new Bird("Eagle", "Aquila chrysaetos", 10, 15, "Brown",
            "Wild", 0, 0, 2, true, true, true);

        System.out.println("Mammal Name: " + mammal.name);
        System.out.println("Mammal Species: " + mammal.species);
        System.out.println("Mammal Age: " + mammal.age);

        System.out.println("Bird Name: " + bird.name);
        System.out.println("Bird Species: " + bird.species);
        System.out.println("Bird Age: " + bird.age);

        mammal.makeSound();
        bird.makeSound();

        Dog dog = new Dog("Buddy", "Golden Retriever", 2, 30, "Golden",
            "Domestic", 0, 0, "Golden", true, true, true, "Golden Retriever",
            "Woof!", true, true);

        System.out.println("Dog Name: " + dog.name);
        System.out.println("Dog Species: " + dog.species);
        System.out.println("Dog Age: " + dog.age);

        System.out.println("Bark sound when playing: " +
            dog.getBarkingSound());
        dog.setBarkingSound("Woof woof woof!");
        System.out.println("Bark sound when angry: " +
            dog.getBarkingSound());

        Snake snake = new Snake("Cobra", "Naja naja", 5, 10, "Black",
            "Wild", 0, 0, true, true, true, true, true, true, true);

        System.out.println("Snake Name: " + snake.name);
        System.out.println("Snake Species: " + snake.species);
        System.out.println("Snake Age: " + snake.age);

        snake.biteHuman();
    }
}
```

Code 1.64: Basic Usage of Classes and Objects

The "BasicOOP" class demonstrates the use of classes and objects in Java to implement OOP concepts such as inheritance, polymorphism, encapsulation, and abstraction. It creates instances of the "Animal", "Mammal", "Bird", "Dog", and "Snake" classes and interacts with them by calling their methods and accessing their properties.

1.5.6.1 Object Creation and Initialization

```
Animal animal = new Animal("Dog", "Canis lupus familiaris", 5, 20, "Brown",
    "Domestic", 0, 0);

System.out.println("Name: " + animal.name);
System.out.println("Species: " + animal.species);
System.out.println("Age: " + animal.age);

animal.move(5, 10);
```

Code 1.65: Object Declaration and Initialization

In this code, we create an instance of the "Animal" class and initialize it with some values. An "object" is an instance of a class. Declaring a variable of a class is called "declaration", and assigning a value to the variable is called "initialization". When a class is defined, no memory is allocated, but when it is instantiated (i.e., an object is created), memory is allocated. Instantiation is the process of creating an object from a class. To access the properties and methods of an object, we use the dot operator (".). For example, to access the "name" property of the "animal" object, we use "animal.name".

The syntax for creating an object in Java is:

```
ClassName objectName = new ClassName();
```

Code 1.66: Object Creation Syntax

1.5.6.2 Inheritance

```
Mammal mammal = new Mammal("Cat", "Felis catus", 3, 10, "White",
    "Domestic", 0, 0, "Short", true, true, true);
Bird bird = new Bird("Eagle", "Aquila chrysaetos", 10, 15, "Brown", "Wild",
    0, 0, 2, true, true, true);

System.out.println("Mammal Name: " + mammal.name);
System.out.println("Mammal Species: " + mammal.species);
System.out.println("Mammal Age: " + mammal.age);

System.out.println("Bird Name: " + bird.name);
System.out.println("Bird Species: " + bird.species);
System.out.println("Bird Age: " + bird.age);
```

Code 1.67: Basic Inheritance

In this code, we demonstrate inheritance by creating objects of the “Mammal” and “Bird” classes, which inherit from the “Animal” class. The “Mammal” and “Bird” classes inherit the properties and methods of the “Animal” class. We create instances of the “Mammal” and “Bird” classes and access their properties such as “name”, “species”, and “age”.

1.5.6.3 Polymorphism

```
mammal.makeSound();
bird.makeSound();
```

Code 1.68: Basic Polymorphism

In this code, we demonstrate polymorphism by calling the “makeSound()” method on the “mammal” and “bird” objects. The “makeSound()” method is defined in the “Animal” class and overridden in the “Mammal” and “Bird” classes. When the “makeSound()” method is called on the “mammal” and “bird” objects, the implementation of the method in the respective classes is executed. This allows the same method name to do different things based on the object that it is acting upon.

1.5.6.4 Encapsulation

```
Dog dog = new Dog("Buddy", "Golden Retriever", 2, 30, "Golden", "Domestic",
    0, 0, "Golden", true, true, true, "Golden Retriever", "Woof!", true,
    true);

System.out.println("Dog Name: " + dog.name);
System.out.println("Dog Species: " + dog.species);
System.out.println("Dog Age: " + dog.age);

System.out.println("Bark sound when playing: " + dog.getBarkingSound());
dog.setBarkingSound("Woof woof woof!");
System.out.println("Bark sound when angry: " + dog.getBarkingSound());
```

Code 1.69: Basic Encapsulation

In this code, we demonstrate encapsulation by creating an instance of the “Dog” class and accessing the private member variables of the class using public getter and setter methods. The “Dog” class has private member variables such as “breed”, “barkingSound”, “tailWagging”, and “isTrained”. We use the public getter and setter methods to access and modify these private member variables. This protects the data from being arbitrarily accessed or modified by other code defined outside the class.

1.5.6.5 Abstraction

```
/**
 * Trying to create an instance of the abstract class ‘Reptile‘ will result
    in
 * a compilation error.
 */
```

```
// Reptile reptile = new Reptile("Lizard", "Lacertilia", 1, 1, "Green",
 // "Domestic", 0, 0, true, true, true, true,
 // true);

Snake snake = new Snake("Cobra", "Naja naja", 5, 10, "Black", "Wild", 0, 0,
    true, true, true, true, true, true, true);

System.out.println("Snake Name: " + snake.name);
System.out.println("Snake Species: " + snake.species);
System.out.println("Snake Age: " + snake.age);

snake.biteHuman();
```

Code 1.70: Basic Abstraction

In this code, we demonstrate abstraction by creating an abstract class “Reptile” with abstract methods and creating a concrete class “Snake” that extends the “Reptile” class and implements the abstract methods. An abstract class cannot be instantiated, but it can be used as a blueprint for other classes that inherit from it. The “Snake” class extends the “Reptile” class and implements the “biteHuman()” method, which is declared as an abstract method in the “Reptile” class. The “Snake” class provides an implementation for the “biteHuman()” method, which simulates the snake biting a human.

1.5.7 Summary

Object-oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.

The main principles of OOP are:

- Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object. It allows a class to inherit the properties and methods of another class.
- Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
- Abstraction is a concept of object-oriented programming that allows hiding the implementation details and showing only the functionality to the user. It helps to reduce programming complexity and effort.

In Java, everything is an object aside from the primitive data types (e.g., int, float, char, etc.). Java is an object-oriented programming language that is simple, easy to learn, secure, robust, high performance, multithreaded, interpreted, distributed, and dynamic.

2

Control Statements

2.1 Introduction

Control statements are used to control the flow of execution in a program. They allow you to make decisions, repeat code, and jump to different parts of the program. Control statements are essential for writing programs that perform different actions based on different conditions.

In Java, control statements can be categorized into the following types:

- Conditional statements: These statements allow you to make decisions based on conditions. The most common conditional statements are the "if", "if-else", "if-else-if ladder", and "switch" statements.
- Iteration statements: These statements allow you to repeat a block of code multiple times. The most common iteration statements are the "while", "do-while", "for", and "for-each" loops.
- Jump statements: These statements allow you to jump to different parts of the program. The most common jump statements are the "break", "continue", and "return" statements.

2.2 Conditional Statements and Decision Making

Conditional statements are used to make decisions in a program. They allow you to execute different blocks of code based on different conditions. Conditional statements are essential for writing programs that perform different actions based on different conditions.

2.2.1 If Statement

The "if" statement is used to execute a block of code if a specified condition is true. If the condition is false, the code block is not executed.

```
1  int x = 10;
2
3  if (x > 0) {
4      System.out.println("The number is positive.");
5  }
```

Code 2.1: If Statement

In this code, the "if" statement checks if the value of the variable "x" is greater than 0. If the condition is true, the message "The number is positive." is printed to the console.

2.2.2 If-Else Statement

The "if-else" statement is used to execute a block of code if a specified condition is true and another block of code if the condition is false.

```
int x = -5;

if (x > 0) {
    System.out.println("The number is positive.");
} else {
    System.out.println("The number is not positive.");
}
```

Code 2.2: If-Else Statement

In this code, the "if-else" statement checks if the value of the variable "x" is greater than 0. If the condition is true, the message "The number is positive." is printed to the console. If the condition is false, the message "The number is not positive." is printed to the console.

2.2.3 If-Else-If Ladder

The "if-else-if" ladder is used to execute one block of code out of several blocks based on multiple conditions. If the first condition is true, the first block of code is executed. If the first condition is false and the second condition is true, the second block of code is executed, and so on.

```
int x = 0;

if (x > 0) {
    System.out.println("The number is positive.");
} else if (x < 0) {
    System.out.println("The number is negative.");
} else {
    System.out.println("The number is zero.");
}
```

Code 2.3: If-Else-If Ladder

In this code, the "if-else-if" ladder checks if the value of the variable "x" is greater than 0, less than 0, or equal to 0. Depending on the value of "x", the corresponding message is printed to the console.

2.2.4 Nested If-Else Statements

Nested "if-else" statements are used to check multiple conditions within another condition. The inner "if-else" statement is executed only if the outer condition is true.

```
int x = 10;
int y = 20;

if (x > 0) {
    if (y > 0) {
        System.out.println("Both numbers are positive.");
    } else {
        System.out.println("The first number is positive, but the second
            number is not positive.");
    }
} else {
    System.out.println("The first number is not positive.");
}
```

Code 2.4: Nested If-Else Statements

In this code, the outer "if" statement checks if the value of the variable "x" is greater than 0. If the condition is true, the inner "if" statement checks if the value of the variable "y" is greater than 0. Depending on the values of "x" and "y", the corresponding message is printed to the console. If the condition of the outer "if" statement is false, the message "The first number is not positive." is printed to the console.

2.2.5 Switch Statement

The "switch" statement is used to execute one block of code out of multiple blocks based on the value of an expression. It is an alternative to the "if-else-if" ladder when you have to check multiple conditions based on the value of a single expression. The "switch" statement is more efficient than the "if-else-if" ladder when you have a large number of conditions to check.

```
int day = 3;
String dayName = "";

switch (day) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
        dayName = "Tuesday";
        break;
    case 3:
        dayName = "Wednesday";
        break;
    case 4:
        dayName = "Thursday";
        break;
    case 5:
        dayName = "Friday";
        break;
    case 6:
        dayName = "Saturday";
        break;
```

```
        case 7:
            dayName = "Sunday";
            break;
        default:
            dayName = "Invalid day";
            break;
    }

    System.out.println("The day is " + dayName);
```

Code 2.5: Switch Statement

In this code, the “switch” statement checks the value of the variable “day”. Depending on the value of “day”, the corresponding day name is assigned to the variable “dayName”. If the value of “day” does not match any of the “case” values, the “default” block of code is executed.

2.3 Iteration Statements

Iteration statements are used to repeat a block of code multiple times. They allow you to execute the same code multiple times without writing the code repeatedly. Iteration statements are essential for writing programs that perform repetitive tasks.

- While Loop: The “while” loop is used to execute a block of code as long as a specified condition is true.
- Do-While Loop: The “do-while” loop is similar to the “while” loop, but it always executes the block of code at least once before checking the condition.
- For Loop: The “for” loop is used to execute a block of code a specified number of times.
- For Each Loop: The “for-each” loop is used to iterate over an array or collection of elements.
- Nested Loops: Nested loops are loops within loops. They are used to perform repetitive tasks that require multiple levels of iteration.

2.3.1 While Loop

The “while” loop is used to execute a block of code as long as a specified condition is true. It is used when the number of iterations is not known in advance. The condition is checked before the block of code is executed, this means that the block of code may not be executed at all if the condition is false from the beginning.

```
int i = 1;

while (i <= 5) {
    System.out.println(i);
    i++;
}
```

Code 2.6: While Loop

In this code, the “while” loop prints the value of the variable “i” to the console as long as the value of “i” is less than or equal to 5. The value of “i” is incremented by 1 in each iteration.

2.3.2 Do-While Loop

The "do-while" loop is similar to the "while" loop, but it always executes the block of code at least once before checking the condition. This means that the block of code is executed once even if the condition is false from the beginning.

```
int i = 1;

do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

Code 2.7: Do-While Loop

In this code, the "do-while" loop prints the value of the variable "i" to the console as long as the value of "i" is less than or equal to 5. The value of "i" is incremented by 1 in each iteration. The block of code is executed at least once before checking the condition. If the condition is false from the beginning, the block of code is executed once. If after the first execution the condition is false, the block of code is not executed.

2.3.3 For Loop

The "for" loop is used to execute a block of code a specified number of times. It is used when the number of iterations is known in advance. The "for" loop consists of three parts: initialization, condition, and increment/decrement.

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

Code 2.8: For Loop

In this code, the "for" loop prints the value of the variable "i" to the console from 1 to 5. The loop initializes the value of "i" to 1, checks if the value of "i" is less than or equal to 5, and increments the value of "i" by 1 in each iteration.

Initialization (int i = 1): The loop initializes the value of the variable "i" to 1 at the beginning of the loop. This part is usually done to set the initial value of the loop variable.

Condition (i <= 5): The loop checks if the value of the variable "i" is less than or equal to 5 before executing the block of code. If the condition is true, the block of code is executed. If the condition is false, the loop terminates.

Increment/Decrement (i++): The loop increments the value of the variable "i" by 1 after executing the block of code. This part is usually done to update the loop variable after each iteration.

2.3.4 For Each Loop

The "for-each" loop is used to iterate over an array or collection of elements. It simplifies the process of iterating over elements in an array or collection without using an index variable. The loop automatically iterates over each element in the array or collection.

```
int[] numbers = {1, 2, 3, 4, 5};

for (int number : numbers) {
    System.out.println(number);
}
```

Code 2.9: For Each Loop

In this code, the "for-each" loop iterates over the elements in the "numbers" array and prints each element to the console. The loop automatically assigns each element in the array to the variable "number" in each iteration. This particular variant of the "for" loop is useful when you want to iterate over all the elements in an array or collection without using an index variable.

2.3.5 Nested Loops

Nested loops are loops within loops. They are used to perform repetitive tasks that require multiple levels of iteration. Nested loops are useful when you need to iterate over a two-dimensional array or perform a task that requires multiple levels of iteration.

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.println("i: " + i + ", j: " + j);
    }
}
```

Code 2.10: Nested Loops

In this code, the outer "for" loop iterates over the values of the variable "i" from 1 to 3. For each value of "i", the inner "for" loop iterates over the values of the variable "j" from 1 to 3. The loop prints the values of "i" and "j" to the console in each iteration. This results in a total of 9 iterations (3 iterations of the outer loop * 3 iterations of the inner loop).

2.4 Jump Statements

Jump statements re used to skip, move to the next iteration, or exit a code block. They allow you to transfer execution control from one point to another point in the program. These statements provide flexibility and control over program logic to the programmer.

- Break Statement: The "break" statement is used to terminate the execution of the nearest looping statement or switch statement.
- Continue Statement: The "continue" statement is used to go to the next iteration of the loop.
- Return Statement: The "return" statement is used to transfer control from one method to the method that called it.

2.4.1 Break Statement

The “break” statement has three uses. First, it terminates the sequence in a “switch” statement. Second, it can be used to exit a loop. And lastly, it can be used as a form of goto.

2.4.2 Break Statement to Exit a Loop

In loops, you can forcefully terminate a loop, bypassing the conditional expression and execution of the remaining code in the body of the loop. When a “break” statement is encountered inside a loop, the loop is terminated.

```
// BreakLoop.java
public class BreakLoop {
  public static void main(String[] args) {
    int breakIndicator = 0;

    for (int i = 1; i <= 5; i++) {
      breakIndicator = i;
      System.out.println("Iteration No.: " + i);

      if (i >= 3) {
        break;
      }

      System.out.println("Continue Iteration...");
    }

    System.out.println("Loop Exited at Iteration: " + breakIndicator);
  }
}

// Output:
// Iteration No.: 1
// Continue Iteration...
// Iteration No.: 2
// Continue Iteration...
// Iteration No.: 3
```

Code 2.11: Break Statement to Exit a Loop

In this code, the “break” statement is used to terminate the loop when the value of “i” is greater than or equal to 3. The loop is exited when the value of “i” is 3. The “break” statement is placed inside the loop and is executed when the condition “i >= 3” is met.

2.4.3 Break Statement as a form of Goto

Other than its usage for terminating a loop and in “switch” statements, the “break” statement can also be used as a form of goto. Java does not have a goto statement but Jav does have an expanded form of the “break” statement which can be used to break out of one or more blocks of code.

```java
// BreakGoto.java
package com.oop.Jump;

public class BreakGoto {
  public static void main(String[] args) {
    boolean breakIndicator = true;

    first: {
      second: {
        third: {
          System.out.println("Before the break statement");
          if (breakIndicator)
            break second;
          System.out.println("This won't execute.");
        }
        System.out.println("This won't execute.");
      }

      System.out.println("This is after second block.");
    }
  }
}

// Output:
// Before the break statement
// This is after second block.
```

Code 2.12: Break Statement as a Form of Goto

In this code, the "break" statement is used to break out of the "second" block of code. The "break" statement is placed inside the "second" block of code and is executed when the condition "breakIndicator" is true. The "break" statement is used to break out of the "second" block of code and move to the next statement after the "second" block of code.

2.4.4 Continue Statement

The "continue" statement is used to skip the remaining code in the current iteration of a loop and move to the next iteration. It is often used to skip certain iterations of a loop based on a condition.

```java
// Continue.java
package com.oop.Jump;

public class Continue {
  public static void main(String[] args) {
    for (int i = 1; i <= 10; i++) {
      if (i % 2 != 0) {
        continue;
      }
      System.out.println(i + " is even.");
    }
```

```
    }
  }

  // Output:
  // 2 is even.
  // 4 is even.
  // 6 is even.
  // 8 is even.
  // 10 is even.
```

Code 2.13: Continue Statement

In this code, the “continue” statement is used to skip the remaining code in the current iteration of the loop when the value of “i” is odd. The “continue” statement is placed inside the loop and is executed when the condition “i % 2 != 0” is met.

2.4.5 Return Statement

The “return” statement is used to transfer control from one method to the method that called it. It is often used to return a value from a method.

```
// Return.java
package com.oop.Jump;

public class Return {

  public static void main(String[] args) {
    int squareOfTwo = square(2);
    System.out.println("Square of 2 is: " + squareOfTwo);

    String s1 = "Hello";
    String s2 = "World";
    String s3 = concat(s1, s2);
    System.out.println("Concatenated String: " + s3);
  }

  public static int square(int n) {
    return n * n;
  }

  public static String concat(String s1, String s2) {
    return s1 + s2;
  }
}
```

Code 2.14: Return Statement

In this code, there are two methods: “square” and “concat”. The “square” method takes an integer parameter “n” and returns the square of “n”. The “concat” method takes two string parameters “s1” and “s2” and returns the concatenation of “s1” and “s2”.

2.5 Summary

In this chapter, we have covered the following topics:

- The three types of Control Statements which are:
  - Conditional Statements
  - Iteration Statements or Looping Statements
  - Jump Statements
- Conditional Statements: are conditions that are used to make decisions. They are used to determine which code to execute based on a condition. most common conditional statements are “if”, “if-else”, and “switch”.
- Iteration Statements or Looping Statements: are used to repeat a block of code multiple times. The most common iteration statements are “for”, “while”, and “do-while”.
- Jump Statements: are used to transfer control from one part of the program to another. The most common jump statements are “break” for exiting loops or implementing a form of goto, “continue” for skipping the remaining code in the current iteration of a loop, and “return” for transferring control from one method to the method that called it.

2.6 Coding Exercises

The “Scanner” class can be used to read input from the user. The following exercises are designed to help you practice using control statements in Java. You can use the “Scanner” class to read input from the user and test your solutions.

```
1  // ScannerExample.java
2  import java.util.Scanner;
3
4  public class ScannerExample {
5    public static void main(String[] args) {
6      Scanner scanner = new Scanner(System.in);
7
8      System.out.print("Enter a number: ");
9      int number = scanner.nextInt();
10     System.out.println("You entered: " + number);
11
12     scanner.close();
13   }
14 }
```

Code 2.15: Simple Implementation of the Scanner Class

In the above code, we import the “Scanner” class by using the statement “import java.util.Scanner;”. We create an instance of the “Scanner” class by using the statement “Scanner scanner = new Scanner(System.in);”. We read an integer input from the user by using the statement “int number = scanner.nextInt();”. We print the input number by using the statement “System.out.println("You entered: " + number);”. Finally, we close the scanner by using the statement “scanner.close();”.

The following are used to read different types of input from the user:

- “scanner.nextBoolean()” to read a boolean value.
- “scanner.nextByte()” to read a byte value.

- “scanner.nextShort()” to read a short value.
- “scanner.nextInt()” to read an integer value.
- “scanner.nextLong()” to read a long value.
- “scanner.nextFloat()” to read a float value.
- “scanner.nextDouble()” to read a double value.
- “scanner.nextLine()” to read a string value.
- “scanner.next()” to read a single word.

Note In the following exercises, you will use the “Scanner” class to read input from the user.

2.6.1 Exercise 1: Find the Largest Number (Conditional Statements)

In this exercise, you will write a Java program to find the largest of three numbers entered by the user. You should use the “Scanner” class to read input from the user and conditional statements to compare the three numbers and determine the largest number. Finally, you should print the largest number to the console. If the numbers are equal, you should print a message saying "There is no largest number".

1. With the use of the “Scanner” class, read three numbers from the user. and store them in variables “num1”, “num2”, and “num3”.

2. Use conditional statements to compare the three numbers and determine the largest number.

   (a) If “num1” is greater than “num2” and “num1” is greater than “num3”, then “num1” is the largest number and print "The largest number is: num1".

   Example: If “num1” is 5, “num2” is 3, and “num3” is 4, then the output should be "The largest number is: 5".

   (b) If none of the above conditions are true, then the numbers are equal. Thus, print a message saying "There is no largest number".

   Example: If “num1” is 5, “num2” is 5, and “num3” is 5, then the output should be "There is no largest number".

2.6.2 Exercise 2: Print Multiplication Table (Iteration Statements)

In this exercise, you will write a Java program to print the multiplication table of a number entered by the user. You should use the “Scanner” class to read input from the user and iteration statements to print the multiplication table of the entered number. Finally, you should print the multiplication table to the console.

1. With the use of the “Scanner” class, read a number from the user and store it in a variable “number”.

2. Use a “for” loop to iterate from 1 to 10.

   (a) Inside the loop, calculate the product of “number” and the loop variable and store it in a variable “result”.

   (b) Print the multiplication table in the format "number * loop variable = result".

   Example:
   If the user enters 5, then the output should be:

5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50

2.6.3 Exercise 3: Factorial of a Number (Iteration Statements)

In this exercise, you will write a Java program to calculate the factorial of a number entered by the user. You should use the "Scanner" class to read input from the user and iteration statements to calculate the factorial of the entered number. Finally, you should print the factorial to the console.

A factorial of a non-negative integer "n" is the product of all positive integers less than or equal to "n". It is denoted by "n!".

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 \tag{2.1}$$

The formula above shows how to calculate the factorial of a number.

Example: The factorial of 5 is calculated as follows:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \tag{2.2}$$

In this exercise, you will calculate the factorial of a number entered by the user.

1. With the use of the "Scanner" class, read a number from the user and store it in a variable "number".
2. Initialize a variable "factorial" to 1.
3. Use a "for" loop to iterate from 1 to "number".
   (a) Inside the loop, multiply the loop variable by "factorial" and store the result in "factorial".
   (b) Print the factorial of the entered number.

   Example:
   If the user enters 5, then the output should be:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \tag{2.3}$$

   Thus, the factorial of 5 is 120.

2.6.4 Exercise 4: Fibonacci Series (Iteration Statements)

In this exercise, you will write a Java program to print the Fibonacci series up to a specified number of terms entered by the user. You should use the "Scanner" class to read input from the user and iteration statements to calculate and print the Fibonacci series. Finally, you should print the Fibonacci series to the console.

The Fibonacci series is a series of numbers in which each number is the sum of the two preceding ones, usually starting with 0 and 1. The sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on.

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \text{ for } n > 1 \tag{2.4}$$

The formula above shows how to calculate the Fibonacci series.

Example: The Fibonacci series up to 10 terms is calculated as follows:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34 \tag{2.5}$$

In this exercise, you will calculate and print the Fibonacci series up to a specified number of terms entered by the user.

1. With the use of the "Scanner" class, read the number of terms from the user and store it in a variable "terms".
2. Initialize two variables "firstTerm" and "secondTerm" to 0 and 1 respectively.
3. Print the first two terms of the Fibonacci series.
4. Use a "for" loop to iterate from 1 to "terms".
   (a) Inside the loop, calculate the next term of the Fibonacci series by adding "firstTerm" and "secondTerm" and store it in a variable "nextTerm".
   (b) Print the next term of the Fibonacci series.
   (c) Update the values of "firstTerm" and "secondTerm" to the previous two terms of the Fibonacci series.

       Example:
       If the user enters 10, then the output should be:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34 \tag{2.6}$$

       Thus, the Fibonacci series up to 10 terms is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

2.6.5 Exercise 5: Sum of Numbers (Jump Statements)

In this exercise, you will write a Java program to calculate the sum of numbers entered by the user. You should use the "Scanner" class to read input from the user and jump statements to

control the flow of the program. Finally, you should print the sum of the numbers to the console.

1. With the use of the “Scanner” class, read numbers from the user until the user enters a negative number. Store the numbers in a variable “number” and the sum of the numbers in a variable “sum”.

2. Use a “while” loop to read numbers from the user until the user enters a negative number.

   (a) Inside the loop, check if the value of “number” is negative. If it is, break out of the loop.

   (b) Add the value of “number” to the sum of the numbers.

   (c) Continue reading numbers from the user.

   Example:
   If the user enters 5, 10, 15, -1, then the output should be:
   The sum of the numbers is: 30

3

Methods

3.1 Introduction to Methods

A method is a block of code that performs a specific task. It is a set of statements that are grouped together to perform an operation. Methods are used to organize code, make it reusable, and reduce redundancy. In Java, methods are defined within a class and can be called from other parts of the program.

3.2 Declaring Methods

A method in Java is declared using the following syntax:

```
accessModifier returnType methodName(parameterList) {
    // method body
}
```

Code 3.1: Method Declaration

Code 3.1 shows the components of a method declaration:

- **Access Modifier**: The access modifier specifies the visibility of the method. It can be "public", "private", "protected", or "default".
- **Return Type**: The return type specifies the type of value that the method returns. It can be a primitive type, an object type, or "void" if the method does not return a value.
- **Method Name**: The method name is a unique identifier for the method. It is used to call the method from other parts of the program.
- **Parameter List**: The parameter list specifies the parameters that the method accepts. Parameters are variables that are used to pass values to the method. They are optional and can be of any data type.
- **Method Body**: The method body contains the statements that define the behavior of the method. It is enclosed in curly braces.

3.3 Calling Methods

A method is called by using its name followed by parentheses. If the method accepts parameters, the values of the parameters are passed inside the parentheses.

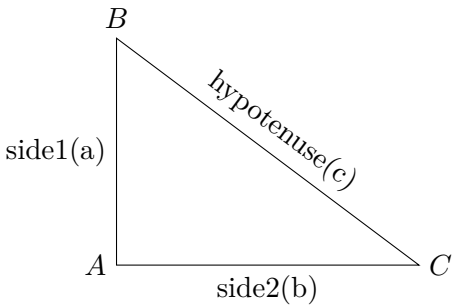
```
// Calling a method without parameters
methodName();

// Calling a method with parameters
methodName(parameter1, parameter2);
```

Code 3.2: Calling a Method

Code 3.2 shows how to call a method with and without parameters. To call a method, you use the method name followed by parentheses. If the method accepts parameters, you pass the values of the parameters inside the parentheses.

3.4 Parameters and Arguments

Parameters are variables that are used to pass values to a method. They are specified in the method declaration and act as placeholders for the values that are passed to the method. Arguments are the actual values that are passed to the method when it is called.



$$c^2 = a^2 + b^2$$

or

$$c = \sqrt{a^2 + b^2}$$

Figure 1: Pythagorean Theorem

```
// Hypotenuse.java
public class Hypotenuse {
  public static void main(String[] args) {
    double side1 = 3.0;
    double side2 = 4.0;

    double hypotenuse = calculateHypotenuse(side1, side2);
    System.out.println("The hypotenuse is: " + hypotenuse);
  }

  public static double calculateHypotenuse(double side1, double side2) {
    double hypotenuse = Math.sqrt(side1 * side1 + side2 * side2);
    return hypotenuse;
  }
```

```
15  }
16
17  // Output:
18  // The hypotenuse is: 5.0
```

Code 3.3: Sample Method for Calculating the Hypotenuse

Code 3.3 shows a sample method for calculating the hypotenuse of a right-angled triangle using the Pythagorean theorem as shown in Figure 1. The method "calculateHypotenuse" accepts two parameters "side1" and "side2" which represent the two sides of the triangle. The method calculates the hypotenuse using the formula $c = \sqrt{a^2 + b^2}$ and returns the result. As per the example, the arguments 3.0 and 4.0 are passed to the method to calculate the hypotenuse of the triangle.

3.5 Types of Methods

There are two types of methods in Java: predefined methods and user-defined methods.

3.5.1 Predefined Methods

Predefined methods are built-in methods that are provided by Java. They are available in the Java API and can be used directly in your programs. Predefined methods are used to perform common tasks such as input/output operations, mathematical calculations, string manipulation, and more.

```
1   // Math class methods
2   double squareRoot = Math.sqrt(16);
3   double power = Math.pow(2, 3);
4   double absoluteValue = Math.abs(-5);
5
6   // String class methods
7   String str = "Hello, World!";
8   int length = str.length();
9   String upperCase = str.toUpperCase();
10  String lowerCase = str.toLowerCase();
```

Code 3.4: Examples of Predefined Methods

Code 3.4 shows examples of predefined methods in Java. The "Math" class provides methods for mathematical calculations such as calculating the square root, power, and absolute value of a number. The "String" class provides methods for string manipulation such as getting the length of a string, converting a string to uppercase, and converting a string to lowercase.

3.5.2 User-Defined Methods

User-defined methods are methods that are created by the programmer to perform specific tasks. They are defined within a class and can be called from other parts of the program. User-defined methods are used to organize code, make it reusable, and reduce redundancy.

```
1   // UserDefinedMethods.java
```

```java
public class UserDefinedMethods {
  public static void main(String[] args) {
    float areaEllipse = calculateAreaEllipse(3, 4);
    System.out.println("Area of Ellipse: " + areaEllipse);

    String reversedString = reverseString("Hello, World!");
    System.out.println("Reversed String: " + reversedString);
  }

  public static float calculateAreaEllipse(float a, float b) {
    return (float) (Math.PI * a * b);
  }

  public static String reverseString(String str) {
    String reversed = "";
    for (int i = str.length() - 1; i >= 0; i--) {
      reversed += str.charAt(i);
    }
    return reversed;
  }
}

// Output:
// Area of Ellipse: 37.699112
// Reversed String: !dlroW ,olleH
```

Code 3.5: Sample User-Defined Methods

Code 3.5 shows examples of user-defined methods in Java. The “calculateFactorial” method calculates the factorial of a number using a “for” loop. The “reverseString” method reverses a string by iterating over the characters of the string in reverse order.

3.6 Static Methods

A static method is a method that belongs to the class rather than an instance of the class. It can be called directly using the class name without creating an object of the class. Static methods are used for utility functions that do not require access to instance variables.

```java
// StaticMethod.java
public class StaticMethod {
  public static void main(String[] args) {
    Person person1 = new Person("Alice", 25);
    Person person2 = new Person("Bob", 30);

    int countPersons = Person.getCountPersons();
    System.out.println("Number of Persons: " + countPersons);
  }
}

// Person.java
```

```
13  public class Person {
14    private String name;
15    private int age;
16    private static int count = 0;
17
18    public Person(String name, int age) {
19      this.name = name;
20      this.age = age;
21      count++;
22    }
23
24    public String getName() {
25      return name;
26    }
27
28    public int getAge() {
29      return age;
30    }
31
32    public static int getCountPersons() {
33      return count;
34    }
35  }
36
37  // Output:
38  // Number of Persons: 2
```

Code 3.6: Sample Static Method

Code 3.6 shows an example of a static method in Java. The "Person" class has a static method "getCountPersons" that returns the number of instances of the "Person" class. The static method can be called using the class name "Person" without creating an object of the class.

3.7 Method Overloading

Method overloading is a feature that allows a class to have multiple methods with the same name but different parameters. It is used to provide different implementations of a method based on the number or type of parameters.

```
1   // MethodOverloading.java
2   public class MethodOverloading {
3     public static void main(String[] args) {
4       int sum1 = add(1, 2);
5       int sum2 = add(1, 2, 3);
6       double sum3 = add(1.5, 2.5);
7
8       System.out.println("Sum of 1 and 2: " + sum1);
9       System.out.println("Sum of 1, 2, and 3: " + sum2);
10      System.out.println("Sum of 1.5 and 2.5: " + sum3);
11    }
```

```
12
13    public static int add(int a, int b) {
14      return a + b;
15    }
16
17    public static int add(int a, int b, int c) {
18      return a + b + c;
19    }
20
21    public static double add(double a, double b) {
22      return a + b;
23    }
24  }
25
26  // Output:
27  // Sum of 1 and 2: 3
28  // Sum of 1, 2, and 3: 6
29  // Sum of 1.5 and 2.5: 4.0
```

Code 3.7: Sample Method Overloading

Code 3.7 shows an example of method overloading in Java. The "add" method is overloaded to accept different numbers of parameters and different types of parameters. The method can be called with two integers, three integers, or two doubles.

3.8 Recursion

Recursion is a programming technique in which a method calls itself to solve a problem. It is used to break down a complex problem into smaller subproblems that are easier to solve. Recursion consists of two parts: the base case and the recursive case. The base case is the condition that stops the recursion, while the recursive case is the condition that calls the method recursively.

```
1   // RecursiveMethod.java
2   public class RecursiveMethod {
3     public static void main(String[] args) {
4       int factorial = calculateFactorial(5);
5       System.out.println("Factorial of 5: " + factorial);
6     }
7
8     public static int calculateFactorial(int n) {
9       if (n == 0) {
10        return 1;
11      } else {
12        return n * calculateFactorial(n - 1);
13      }
14    }
15  }
16
17  // Output:
```

```
18  // Factorial of 5: 120
```

Code 3.8: Sample Recursive Method

Code 3.8 shows an example of a recursive method in Java. The "calculateFactorial" method calculates the factorial of a number using recursion. The base case is when the number is 0, in which case the method returns 1. The recursive case is when the number is greater than 0, in which case the method calls itself with the number decremented by 1.

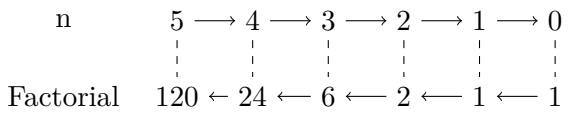


Figure 2: Recursive Calls for Calculating the Factorial of 5

Figure 2 shows the recursive calls for calculating the factorial of 5 using the "calculateFactorial" method. The method is called repeatedly with decreasing values of "n" until the base case is reached.

3.9 Summary

A **Method** is a block of code that performs a specific task. It is used to organize code, make it reusable, and reduce redundancy. Methods are declared using an access modifier, return type, method name, parameter list, and method body. Methods can be called using the method name followed by parentheses.

Parameters are variables that are used to pass values to a method. They are specified in the method declaration and act as placeholders for the values that are passed to the method. **Arguments** are the actual values that are passed to the method when it is called.

There are two types of methods in Java: predefined methods and user-defined methods. **Predefined methods** are built-in methods provided by Java for common tasks such as input/output operations, mathematical calculations, and string manipulation. **User-defined methods** are methods created by the programmer to perform specific tasks.

A **static method** is a method that belongs to the class rather than an instance of the class. It can be called directly using the class name without creating an object of the class. Static methods are used for utility functions that do not require access to instance variables.

Method overloading is a feature that allows a class to have multiple methods with the same name but different parameters. It is used to provide different implementations of a method based on the number or type of parameters.

Recursion is a programming technique in which a method calls itself to solve a problem. It is used to break down a complex problem into smaller subproblems that are easier to solve. Recursion consists of two parts: the base case and the recursive case.

3.10 Coding Exercises

The following exercises are designed to help you practice using methods in Java. You can create separate methods for each exercise and call them from the "main" method to test your solutions.

3.10.1 Exercise 1: Calculate Speed

In this exercise, you will write a Java program to calculate the speed of a vehicle. The speed of a vehicle is calculated using the formula:

$$\text{Speed} = \frac{\text{Distance}}{\text{Time}} \tag{3.1}$$

$$\text{Distance} = \text{Final Distance} - \text{Initial Distance} \tag{3.2}$$

$$\text{Time} = \text{Final Time} - \text{Initial Time} \tag{3.3}$$

$$\text{Speed} = \frac{\text{Final Distance} - \text{Initial Distance}}{\text{Final Time} - \text{Initial Time}} \tag{3.4}$$

1. Create a method called "calculateSpeed" that accepts the initial distance, final distance, initial time, and final time as parameters and returns the speed of the vehicle.

2. Calculate the speed of a vehicle that travels from an initial distance of 0 meters to a final distance of 100 meters in 10 seconds.

3. Print the speed of the vehicle to the console.

   Example:
   A vehicle left from 0 meters and traveled to 100 meters in 10 seconds. The speed of the vehicle is 10 meters per second.

   $InitialDistance$: 0
   $FinalDistance$: 100
   $InitialTime$: 0
   $FinalTime$: 10

   $Speed = \frac{100-0}{10-0} = \frac{100}{10} = 10$
   $Speed = 10$ meters per second

   Output:
   The vehicle traveled from 0 meters to 100 meters in 0 seconds to 10 seconds at a speed of 10 meters per second.

```java
// CalculateSpeed.java
package com.oop.Exercises;

import java.util.Scanner;

public class CalculateSpeed {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        double finalDistance = 0, initialDistance = 0, finalTime = 0,
            initialTime = 0;

        do {
            if (finalDistance <= initialDistance) {
                System.out.println("\nFinal distance should be greater than
                    initial distance.");
```

```
15              }
16
17              System.out.print("Enter the initial distance: ");
18              initialDistance = scanner.nextDouble();
19              System.out.print("Enter the final distance: ");
20              finalDistance = scanner.nextDouble();
21          } while (finalDistance <= initialDistance);
22
23          do {
24              if (finalTime <= initialTime) {
25                  System.out.println("\ndFinal time should be greater than
                        initial time.");
26              }
27
28              System.out.print("Enter the initial time: ");
29              initialTime = scanner.nextDouble();
30              System.out.print("Enter the final time: ");
31              finalTime = scanner.nextDouble();
32          } while (finalTime <= initialTime);
33
34          double speed = calculateSpeed(initialDistance, finalDistance,
                initialTime, finalTime);
35
36          String output = String.format(
37                  "The vehicle traveled from %.2f meters to %.2f meters in %.2f
                        seconds to %.2f seconds at a speed of %.2f meters per
                        second.",
38                  initialDistance, finalDistance, initialTime, finalTime,
                        speed);
39
40          System.out.println("\n" + output);
41
42          scanner.close();
43      }
44
45      public static double calculateSpeed(double initialDistance, double
            finalDistance, double initialTime,
46              double finalTime) {
47          // Calculate the speed of the vehicle
48      }
49  }
```

Code 3.9: Initial Code for Exercise 1

Complete the code in Code 3.9 by implementing the “calculateSpeed” method to calculate the speed of the vehicle. Test the program by entering the initial distance, final distance, initial time, and final time from the user and printing the speed of the vehicle to the console.

3.10.2 Exercise 2: Arithmetic Sequence

In this exercise, you will write a Java program to calculate the sum of an arithmetic sequence using recursion. An arithmetic sequence is a sequence of numbers in which the difference between consecutive terms is constant. The sum of an arithmetic sequence is calculated using the formula:

$$\text{Sum} = \frac{n}{2} \times (2a + (n-1)d) \tag{3.5}$$

Where:

- n is the number of terms in the sequence.
- a is the first term in the sequence.
- d is the common difference between consecutive terms.
- $2a + (n-1)d$ is the sum of the first and last terms in the sequence.
- $\frac{n}{2} \times (2a + (n-1)d)$ is the sum of the arithmetic sequence.

1. Create a method called "calculateArithmeticSequence" that accepts the first term, common difference, number of terms, and current term as parameters and returns the sum of the arithmetic sequence.

2. Calculate the sum of an arithmetic sequence with the first term 1, common difference 2, and number of terms 5.

   Base Case: If the current term is equal to the number of terms, return the current term.
   Recursive Case: Calculate the sum of the arithmetic sequence using recursion.

3. Print the sum of the arithmetic sequence to the console.

   Example:
   If the first term is 1, the common difference is 2, and the number of terms is 5, then the sum of the arithmetic sequence is 25.

   $Sequence : 1, 3, 5, 7, 9$
   $Sum : 1 + 3 + 5 + 7 + 9 = 25$

```java
// ArithmeticSequence.java
package com.oop.Exercises;

import java.util.Scanner;

public class ArithmeticSequence {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the first term of the arithmetic sequence: ");
        double firstTerm = scanner.nextDouble();
        System.out.print("Enter the common difference of the arithmetic sequence: ");
        double commonDifference = scanner.nextDouble();
```

```
14          System.out.print("Enter the number of terms in the arithmetic
                sequence: ");
15          int numberOfTerms = scanner.nextInt();
16
17          double sum = calculateArithmeticSequence(firstTerm,
                commonDifference, numberOfTerms, 1);
18
19          String output = String.format(
20                  "The sum of the arithmetic sequence with the first term %.2f,
                        common difference %.2f, and %d terms is %.2f.",
21                  firstTerm, commonDifference, numberOfTerms, sum);
22
23          System.out.println("\n" + output);
24
25          scanner.close();
26      }
27
28      public static double calculateArithmeticSequence(double firstTerm,
            double commonDifference, int numberOfTerms,
29              int currentTerm) {
30          // Calculate the sum of the arithmetic sequence using recursion
31      }
32  }
```

Code 3.10: Initial Code for Exercise 2

Complete the code in Code 3.10 by implementing the “calculateArithmeticSequence” method to calculate the sum of the arithmetic sequence using recursion. Test the program by entering the first term, common difference, and number of terms from the user and printing the sum of the arithmetic sequence to the console.

4

Scope and Access Modifiers

4.1 Introduction to Scope

In Java, the **scope** of a variable refers to the region of the program where the variable is accessible. Variables in Java have different scopes depending on where they are declared. The scope of a variable determines where it can be used in the program.

4.2 Types of Scope

Scopes in Java can be classified into different types.

4.2.1 Block Scope

A variable with block scope is accessible only within the block of code where it is declared. A block is a set of statements enclosed in curly braces "{ }". Variables declared inside a block are local to that block and cannot be accessed outside the block.

```
1  // BlockScope.java
2  public class BlockScope {
3      public static void main(String[] args) {
4          int x = 10;
5
6          if (x > 5) {
7              int y = 20;
8              System.out.println("x: " + x);
9              System.out.println("y: " + y);
10         }
11
12         // Variable y is not accessible here
13         System.out.println("x: " + x);
14     }
15 }
```

Code 4.1: Block Scope

Code 4.1 shows an example of block scope in Java. The variable "x" is declared in the "main"

method and is accessible throughout the method. The variable “y” is declared inside the “if” block and is accessible only within the block.

4.2.2 Class Scope

A variable with class scope is accessible throughout the class in which it is declared. Class-level variables are declared outside any method and are accessible by all methods in the class.

```
// ClassScope.java
public class ClassScope {
    static int x = 10;
    int y = 20;

    public static void main(String[] args) {
        System.out.println("x: " + x);

        ClassScope obj = new ClassScope();
        System.out.println("y: " + obj.y);

        obj.display();
    }

    public void display() {
        System.out.println("x: " + x);
        System.out.println("y: " + y);
    }
}
```

Code 4.2: Class Scope

Code 4.2 shows an example of class scope in Java. The variable “x” is declared as a static variable and is accessible throughout the class. The variable “y” is an instance variable and is accessible by all methods in the class.

4.2.3 Method Scope

A variable with method scope is accessible only within the method where it is declared. Method-level variables are local to the method and cannot be accessed outside the method.

```
// MethodScope.java
public class MethodScope {
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        System.out.println("x: " + x);

        display(y);
    }

    public static void display(int y) {
        // Variable x is not accessible here
```

```
        System.out.println("x: " + x);

        // Variable y is accessible here
        System.out.println("y: " + y);

        // Variable z is accessible here
        int z = 20;
        System.out.println("z: " + z);
    }
}
```

Code 4.3: Method Scope

Code 4.3 shows an example of method scope in Java. The variable "x" is declared in the "main" method and is accessible only within the method. The variable "y" is passed as a parameter to the "display" method and is accessible within the method. The variable "z" is declared inside the "display" method and is accessible only within the method.

4.3 Introduction to Access Modifiers

In Java, **access modifiers** are keywords that are used to control the visibility of classes, methods, and variables. Access modifiers determine whether a class, method, or variable can be accessed by other classes or methods in the program.

4.4 Types of Access Modifiers

4.4.1 Public

The **public** access modifier allows a class, method, or variable to be accessed by any other class or method in the program. Public members are visible to all classes and methods.

```
// PublicAccessModifier.java
public class PublicAccessModifier {
    public static void display() {
        System.out.println("This is a public method.");
    }
}

// PublicAccessModifierDemo.java
public class PublicAccessModifierDemo {
    public static void main(String[] args) {
        PublicAccessModifier.display();
    }
}
```

Code 4.4: Public Access Modifier

Code 4.4 shows an example of the public access modifier in Java. The "display" method in the "PublicAccessModifier" class is declared as public and can be accessed by the "PublicAccess-

ModifierDemo" class.

4.4.2 Private

The **private** access modifier restricts the visibility of a class, method, or variable to the class in which it is declared. Private members are accessible only within the class in which they are declared.

```
// PrivateAccessModifier.java
public class PrivateAccessModifier {
    private static void display() {
        System.out.println("This is a private method.");
    }

    public static void main(String[] args) {
        display();
    }
}

// PrivateAccessModifierDemo.java
public class PrivateAccessModifierDemo {
    public static void main(String[] args) {
        // The display method is not accessible here
        PrivateAccessModifier.display();
    }
}
```

Code 4.5: Private Access Modifier

Code 4.5 shows an example of the private access modifier in Java. The "display" method in the "PrivateAccessModifier" class is declared as private and can be accessed only within the class.

4.4.3 Protected

The **protected** access modifier allows a class, method, or variable to be accessed by classes in the same package or subclasses of the class. Protected members are visible to classes in the same package and subclasses.

```
// ProtectedAccessModifier.java
public class ProtectedAccessModifier {
    protected static void display() {
        System.out.println("This is a protected method.");
    }
}

// ProtectedAccessModifierDemo.java
public class ProtectedAccessModifierDemo extends ProtectedAccessModifier {
    public static void main(String[] args) {
        display();
    }
```

```
}
```

Code 4.6: Protected Access Modifier

Code 4.6 shows an example of the protected access modifier in Java. The "display" method in the "ProtectedAccessModifier" class is can be accessed by the "ProtectedAccessModifierDemo" class because it is a subclass of the "ProtectedAccessModifier" class.

4.4.4 Default

The **default** access modifier is also known as package-private. It does not use any keyword and is the default access modifier in Java. Default members are accessible only within the same package.

```
// DefaultAccessModifier.java
package com.oop.AccessModifier.Default;

class DefaultAccessModifier {
    static void display() {
        System.out.println("This is a default method.");
    }
}

// DefaultAccessModifierDemo.java
package com.oop.AccessModifier.Default;

public class DefaultAccessModifierDemo {
    public static void main(String[] args) {
        DefaultAccessModifier.display();
    }
}
```

Code 4.7: Default Access Modifier

Code 4.7 shows an example of the default access modifier in Java. The "display" method in the "DefaultAccessModifier" class is declared without any access modifier and can be accessed only within the same package.

4.5 Summary

Scope in Java refers to the region of the program where a variable is accessible. Variables in Java have different scopes depending on where they are declared. The scope of a variable determines where it can be used in the program.

There are different types of scope in Java:

- **Block Scope**: Variables with block scope are accessible only within the block of code where they are declared.
- **Class Scope**: Variables with class scope are accessible throughout the class in which they are declared.

- **Method Scope**: Variables with method scope are accessible only within the method where they are declared.

Access modifiers in Java are keywords that are used to control the visibility of classes, methods, and variables. Access modifiers determine whether a class, method, or variable can be accessed by other classes or methods in the program.

There are different types of access modifiers in Java:

- **Public**: Allows a class, method, or variable to be accessed by any other class or method in the program.
- **Private**: Restricts the visibility of a class, method, or variable to the class in which it is declared.
- **Protected**: Allows a class, method, or variable to be accessed by classes in the same package or subclasses of the class.
- **Default**: Allows a class, method, or variable to be accessed only within the same package.

5

Object-Oriented Programming

5.1 UML Class Diagram

A **Unified Modeling Language (UML)** class diagram is a graphical representation of a class in object-oriented programming. It shows the attributes and methods of a class, as well as the relationships between classes.

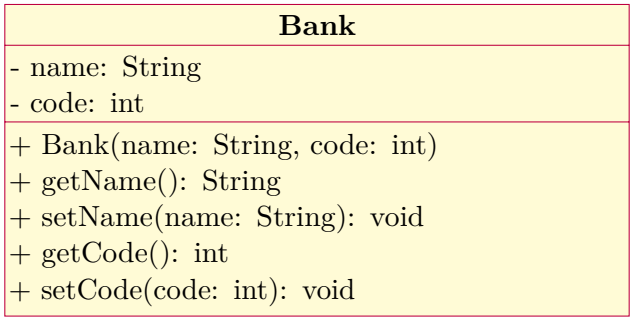


Figure 3: Class Diagram for a Bank

Figure 3 shows a class diagram for a "Bank" class in Java. A class diagram consists of the following elements:

- **Class Name**: The name of the class is displayed at the top of the class box.
- **Attributes**: The attributes of the class are listed below the class name. Each attribute includes the access modifier, name, and data type.
- **Methods**: The methods of the class are listed below the attributes. Each method includes the access modifier, name, parameters, and return type.
- **Access Modifiers**: The access modifiers indicate the visibility of the attributes and methods.

In the class diagram for the "Bank" class, the class has two attributes: "name" and "code", one constructor, and four methods. The attributes and methods are indicated with the access modifiers "+" (public) and "-" (private).

The "+" and "-" symbols in front of the attributes and methods indicate the access modifiers. In a class diagram, there are four types of access modifiers:

- "+" (Public): The attribute or method is accessible by all classes.
- "-" (Private): The attribute or method is accessible only within the class.

- "#" (Protected): The attribute or method is accessible by subclasses.
- "~" (Default): The attribute or method is accessible within the package.

```java
// Bank.java
package com.oop.BankSystem;

public class Bank {
    private String name;
    private int code;

    public Bank(String name, int code) {
        this.name = name;
        this.code = code;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getCode() {
        return code;
    }

    public void setCode(int code) {
        this.code = code;
    }
}
```

Code 5.1: Bank Class in Java

Code 5.1 shows the implementation of the "Bank" class in Java. The class has two private attributes: "name" and "code". It has two constructors to initialize the attributes, as well as getter and setter methods for the attributes.

5.1.1 Bank Class Diagram

A Bank class diagram is a graphical representation of the Bank class in object-oriented programming. It shows the attributes and methods of the Bank class, as well as the relationships between classes.

Figure 4 shows a class diagram for a Bank system in Java. The Bank system consists of six classes: the "Bank" class, the "Account" class, the "SavingsAccount" class, the "CheckingAccount" class, the "Customer" class, and the "Loan" class. The classes have attributes and methods that represent the functionality of a bank system. The class diagram also shows the relationships between the classes, such as inheritance, composition, and aggregation. It also shows the multiplicity of the relationships between classes.

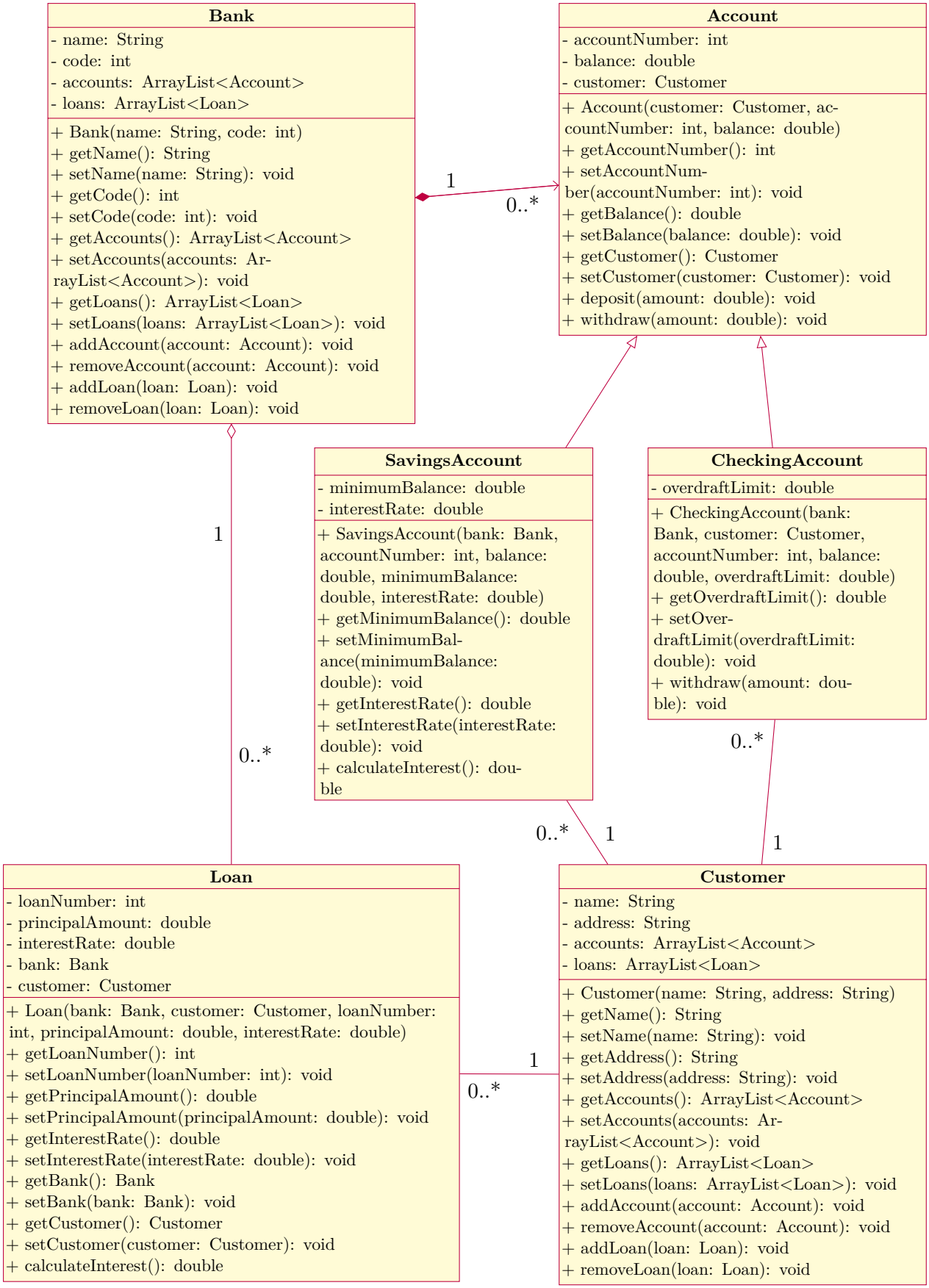


Figure 4: Bank Class Diagram

5.1.2 Relationships Between Classes

In object-oriented programming, classes can have different types of relationships with other classes. Relationships between classes are represented in a class diagram using lines and symbols. Say for example, the Bank class diagram has the following relationships between classes:

- The "Bank" class has a composition relationship with the "Account" class.
- The "SavingAccount" and "CheckingAccount" classes inherit from the "Account" class.
- The "Customer" class has association relationships with the "SavingsAccount", "CheckingAccount", and "Loan" classes.
- The "Loan" class has an aggregation relationship with the "Bank" class.
- The "Account" class has an association relationship with the "Bank" class.
- The "Customer" class has association relationships with the "SavingsAccount", "CheckingAccount", and "Loan" classes.

There are different types of relationships between classes in a class diagram. Below are some of the common types of relationships:

- **Association**: An association is a relationship between two classes that indicates how the classes are related to each other. It can be one-to-one, one-to-many, or many-to-many.
- **Aggregation**: Aggregation is a type of association where one class is composed of one or more other classes. It represents a whole-part relationship. If the main class is destroyed, the composed classes can still exist.
- **Composition**: Composition is a stronger form of aggregation where one class is composed of one or more other classes. The composed classes cannot exist without the main class. If the main class is destroyed, the composed classes are also destroyed.
- **Inheritance**: Inheritance is a relationship between a superclass and a subclass. The subclass inherits the attributes and methods of the superclass.

In the Bank class diagram, the classes have the following relationships:

- The "Bank" class has a composition relationship with the "Account" class.
- The "SavingAccount" and "CheckingAccount" classes inherit from the "Account" class.
- The "Customer" class has association relationships with the "SavingsAccount", "CheckingAccount", and "Loan" classes.
- The "Loan" class has an aggregation relationship with the "Bank" class.

5.1.3 Multiplicity

Multiplicity in a class diagram indicates the number of instances of one class that can be associated with the instances of another class. It is represented using numbers or symbols near the association lines between classes.

There are different types of multiplicity in a class diagram:

- **One-to-One (1:1)**: Indicates that one instance of a class is associated with one instance of another class.
- **One-to-Many (1:*)**: Indicates that one instance of a class is associated with multiple instances of another class.
- **Many-to-One (*:1)**: Indicates that multiple instances of a class are associated with one instance of another class.
- **Many-to-Many (*:*)**: Indicates that multiple instances of a class are associated with multiple instances of another class.

In the Bank class diagram, the relationships between classes have the following multiplicities:

- The “Bank” class has a one-to-many relationship with the “Account” class. This means that one bank can have multiple accounts.
- The “Customer” class has a one-to-many relationship with the “SavingsAccount”, “CheckingAccount”, and “Loan” classes. This means that one customer can own multiple accounts and loans.
- The “Loan” class has a many-to-one relationship with the “Bank” class. This means that multiple loans can be associated with one bank.

5.2 Inheritance

Inheritance in object-oriented programming refers to the concept where a class inherits attributes and methods from another class. The class that inherits the attributes and methods is called a **subclass** or **child class**, and the class that provides the attributes and methods is called a **superclass** or **parent class**. In the real-world, one example of inheritance is the relationship between animals and mammals. Mammals are a subclass of animals and inherit attributes and methods from the animal superclass.

5.2.1 Superclass and Subclass

Inheritance involves two types of classes: a **superclass** and a **subclass**. The superclass is the class that provides the attributes and methods to be inherited, it is also known as the parent class. The subclass is the class that inherits the attributes and methods from the superclass, it is also known as the child class.

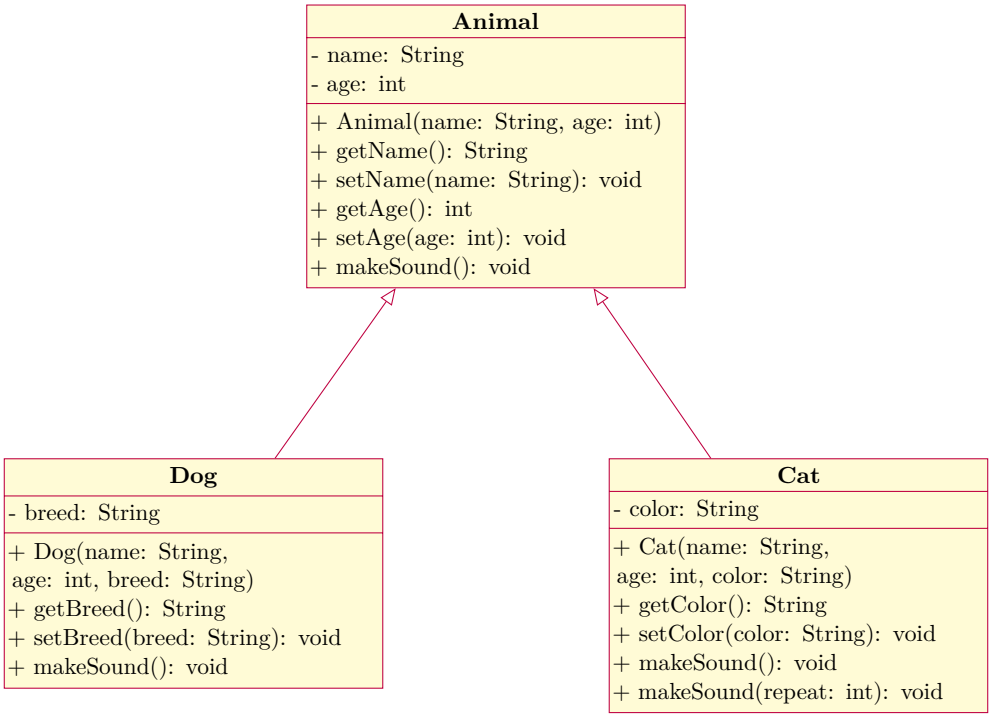


Figure 5: Inheritance in Animal, Dog, and Cat Classes

Figure 5 shows a class diagram of the “Animal”, “Dog”, and “Cat” classes in Java. The “Animal” class is the superclass that provides the attributes and methods common to all animals. The “Dog” and “Cat” classes are subclasses that inherit the attributes and methods from the

"Animal" class. The subclasses can also have additional attributes and methods specific to dogs and cats.

```java
// Animal.java
package com.oop.AnimalKingdom;

public class Animal {
    private String name;
    private int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
```

Code 5.2: Animal Class in Java

```java
// Dog.java
package com.oop.AnimalKingdom;

public class Dog extends Animal {
    private String breed;

    public Dog(String name, int age, String breed) {
        super(name, age);
        this.breed = breed;
    }

    public String getBreed() {
```

```
        return breed;
    }

    public void setBreed(String breed) {
        this.breed = breed;
    }

    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}
```

Code 5.3: Dog Class in Java

```
// Cat.java
package com.oop.AnimalKingdom;

public class Cat extends Animal {
    private String color;

    public Cat(String name, int age, String color) {
        super(name, age);
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }

    public void makeSound(int repeat) {
        for (int i = 0; i < repeat; i++) {
            System.out.print("Meow ");
        }
    }
}
```

Code 5.4: Cat Class in Java

Code 5.5 shows the implementation of the “Animal” class in Java. The class has two private

attributes: “name” and “age”. It has a constructor to initialize the attributes, as well as getter and setter methods for the attributes. The class also has a method to make a sound.

Code 5.6 and Code 5.8 show the implementation of inheritance in the “Dog” and “Cat” classes in Java. The subclasses inherit the attributes and methods from the “Animal” class using the “extends” keyword. The subclasses can override the methods of the superclass to provide specific implementations. The subclasses can also have additional attributes and methods specific to dogs and cats. They also show the use of the “super” keyword to call the superclass constructor which initializes the attributes of the superclass.

5.2.2 Multiple Levels of Inheritance

Inheritance can have multiple levels, where a subclass can further inherit from another subclass. This creates a hierarchy of classes with increasing levels of specialization.

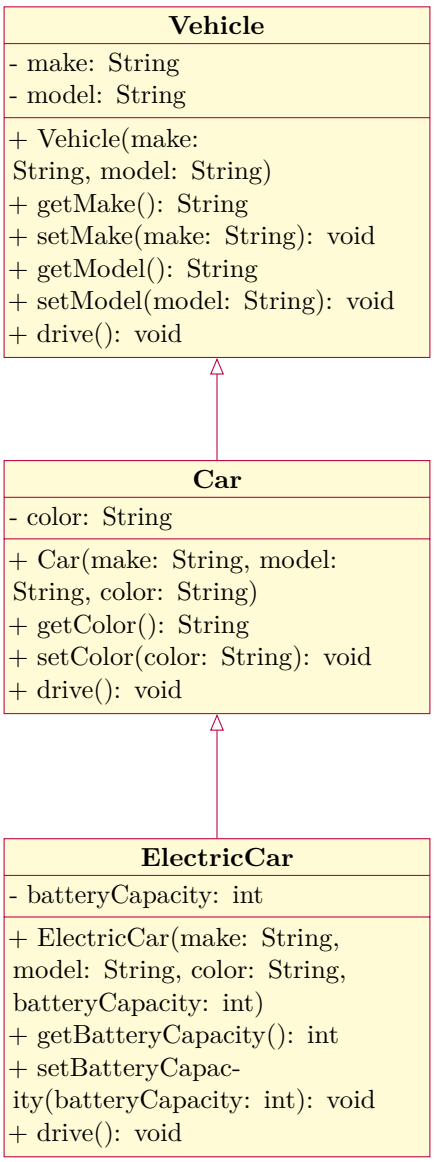


Figure 6: Multiple Levels of Inheritance in Vehicle, Car, and ElectricCar Classes

Figure 6 shows a class diagram of the “Vehicle”, “Car”, and “ElectricCar” classes in Java. The “Vehicle” class is the superclass that provides the attributes and methods common to all vehicles. The “Car” class is a subclass that inherits the attributes and methods from the “Vehicle”

class and adds additional attributes and methods specific to cars. The "ElectricCar" class is a subclass of the "Car" class and inherits the attributes and methods from the "Car" class and the "Vehicle" class. The subclasses can override the methods of the superclass to provide specific implementations.

5.3 Polymorphism

Polymorphism in object-oriented programming refers to the ability of a method to behave differently based on the object that calls it. Polymorphism allows a method to take different forms based on the object that invokes it. It enables flexibility in the design of software systems by providing multiple ways to call a method. Say for example, the "makeSound" method in the "Animal" class can behave differently based on the object that calls it.

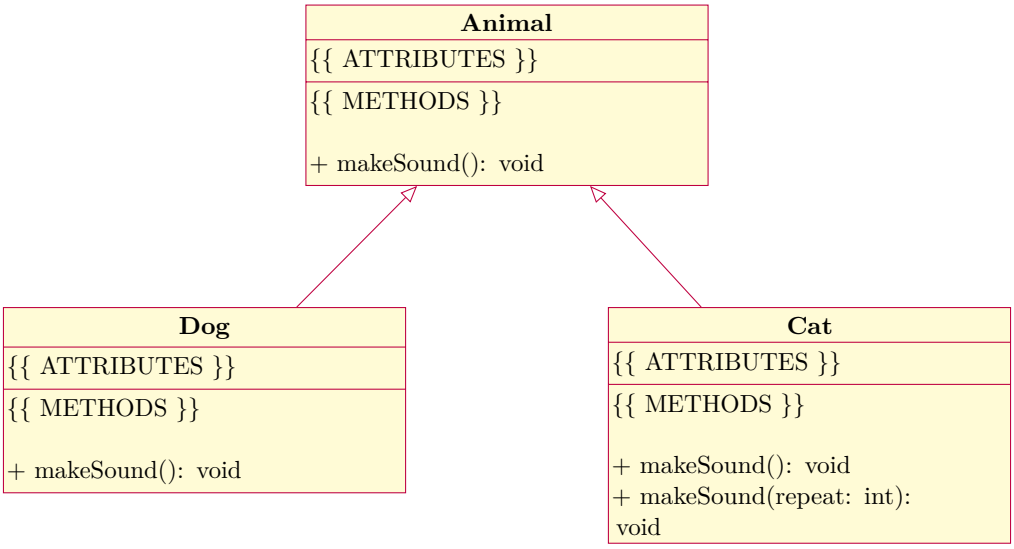


Figure 7: Polymorphism in Animal, Dog, and Cat Classes from Figure 5

5.3.1 Method Overriding

Method overriding is a form of polymorphism that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. The subclass can override the method to provide its own implementation. In figure 7, the "Dog" and "Cat" classes override the "makeSound" method of the "Animal" class to provide specific implementations for dogs and cats. In Java, method overriding is achieved by using the "@Override" annotation.

```java
// Animal.java
package com.oop.AnimalKingdom;

public class Animal {
    /**
     * Rest of the code
     */

    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}
```

Code 5.5: Animal Class in Java

```java
// Dog.java
package com.oop.AnimalKingdom;

public class Dog extends Animal {
    /**
      * Rest of the code
      */

    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}
```

Code 5.6: Dog Class in Java

```java
// Cat.java
package com.oop.AnimalKingdom;

public class Cat extends Animal {
    /**
      * Rest of the code
      */

    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }

    public void makeSound(int repeat) {
        for (int i = 0; i < repeat; i++) {
            System.out.print("Meow ");
        }
    }
}
```

Code 5.7: Cat Class in Java

Code 5.5 shows the implementation of the "Animal" class in Java. The class has a "makeSound" method that prints a generic message. Code 5.6 and Code 5.8 show the implementation of the "Dog" and "Cat" classes in Java. The subclasses override the "makeSound" method of the superclass to provide specific implementations for dogs and cats. If an object of the "Animal" class is created and the "makeSound" method is called, it will print "Animal makes a sound". If an object of the "Dog" class is created and the "makeSound" method is called, it will print "Dog barks". If an object of the "Cat" class is created and the "makeSound" method is called, it

will print “Cat meows”.

5.3.2 Method Overloading

Method overloading is a form of polymorphism that allows a class to have multiple methods with the same name but different parameters. The methods can have different numbers or types of parameters. Method overloading enables flexibility in the design of software systems by providing multiple ways to call a method.

```java
// Cat.java
package com.oop.AnimalKingdom;

public class Cat extends Animal {
    /**
      * Rest of the code
      */

    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }

    public void makeSound(int repeat) {
        for (int i = 0; i < repeat; i++) {
            System.out.print("Meow ");
        }
    }
}
```

Code 5.8: Cat Class in Java

Code 5.8 shows the implementation of the “Cat” class in Java. The class has two “makeSound” methods: one with no parameters and one with an integer parameter. The method with no parameters prints “Cat meows”, and the method with an integer parameter prints “Meow” multiple times based on the value of the parameter. If an object of the “Cat” class is created and the “makeSound” method is called with an integer parameter, it will print “Meow” multiple times based on the value of the parameter.

5.4 Encapsulation

Encapsulation in object-oriented programming is the process of hiding the internal implementation details of a class and providing a public interface to interact with the class. Encapsulation protects the data of a class from being accessed directly and ensures that the data is accessed and modified through methods. In Java, encapsulation is achieved by using access modifiers to control the visibility of attributes and methods. The access modifiers define the level of access to the attributes and methods of a class. Then, getter and setter methods are used to read and modify the attributes of a class. For convention, the names of getter and setter methods are prefixed with “get” and “set”, respectively, followed by the attribute name in camel case.

Figure 8 illustrates the ‘Person‘ class in Java. This class contains three private attributes: ‘name‘, ‘age‘, and ‘address‘. These attributes are encapsulated within the class, meaning they

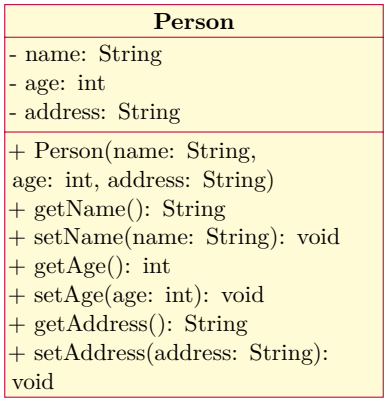


Figure 8: Encapsulation in Person Class

cannot be accessed directly from outside the class. Instead, the class provides getter and setter methods to interact with these attributes. The getter methods allow you to read the values of the attributes, while the setter methods allow you to modify them.

```java
// Person.java
package com.oop.PersonInfo;

public class Person {
    private String name;
    private int age;
    private String address;

    public Person(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    // Getter method for name
    public String getName() {
        return name;
    }

    // Setter method for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for age
    public int getAge() {
        return age;
    }

    // Setter method for age
    public void setAge(int age) {
        this.age = age;
    }
```

```
34
35      // Getter method for address
36      public String getAddress() {
37          return address;
38      }
39
40      // Setter method for address
41      public void setAddress(String address) {
42          this.address = address;
43      }
44  }
```

Code 5.9: Person Class in Java

Code 5.9 demonstrates the implementation of the ‘Person‘ class in Java. This class contains three private attributes: ‘name‘, ‘age‘, and ‘address‘. To manage access to these attributes, the class includes getter and setter methods. The getter methods retrieve the attribute values, while the setter methods update them. As a convention, the names of getter and setter methods are prefixed with ‘get‘ and ‘set‘, respectively, followed by the attribute name in camel case.

5.5 Abstraction

Abstraction in object-oriented programming is the process of hiding the implementation details of a class and showing only the essential features to the outside world. Abstraction allows you to focus on what an object does rather than how it does it. In Java, abstraction is achieved by using abstract classes and interfaces. Abstract classes are classes that cannot be instantiated and can contain abstract methods. Abstract methods are methods that do not have a body and must be implemented by subclasses. Interfaces are reference types that contain only abstract methods and constants. Classes can implement interfaces to provide specific implementations for the abstract methods.

There are two ways to achieve abstraction in Java:

- **Abstract Classes**: An abstract class is a class that cannot be instantiated and can contain abstract methods. Abstract methods are methods that do not have a body and must be implemented by subclasses.
- **Interfaces**: An interface is a reference type in Java that contains only abstract methods and constants. Classes can implement interfaces to provide specific implementations for the abstract methods.

5.5.1 Abstract Classes

Abstract classes in Java are classes that cannot be instantiated and can contain abstract methods. Abstract methods are methods that do not have a body and must be implemented by subclasses. Abstract classes can also contain concrete methods with a body. In Java, you can create an abstract class using the ‘abstract’ keyword.

Figure 9 illustrates the ‘Shape‘ abstract class in Java. This class contains a private attribute ‘color‘ and an abstract method ‘calculateArea()‘. The ‘calculateArea()‘ method does not have a body and must be implemented by subclasses. The ‘Circle‘ and ‘Rectangle‘ classes are concrete subclasses of the ‘Shape‘ class that implement the ‘calculateArea()‘ method to calculate the area of a circle and rectangle, respectively.

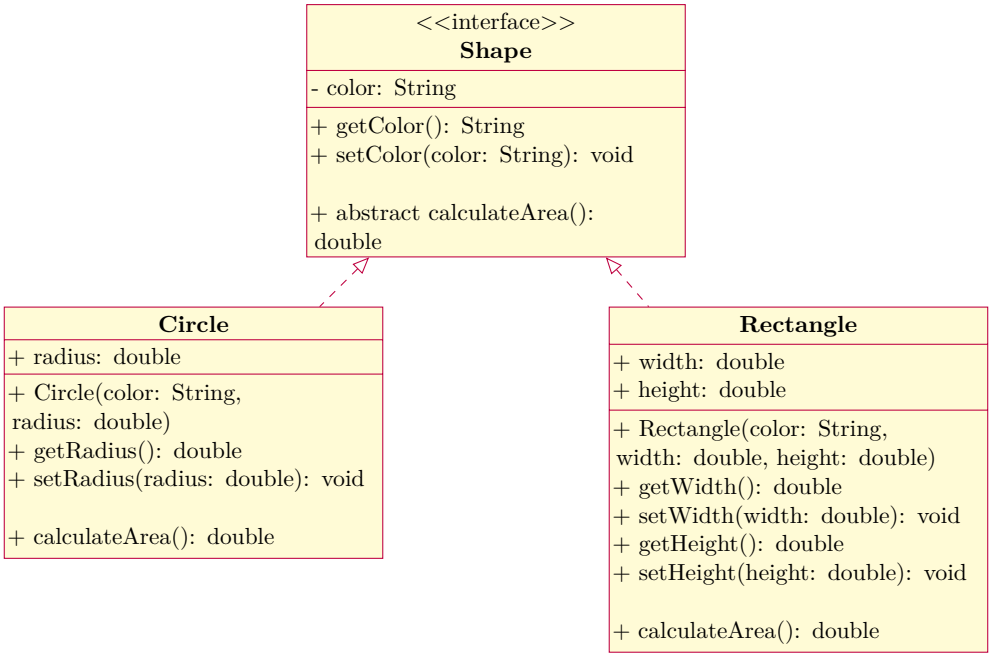


Figure 9: Abstract Class Shape with Concrete Subclasses Circle and Rectangle

```java
// Shape.java
package com.oop.Shapes;

public abstract class Shape {
    private String color;

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public abstract double calculateArea();
}
```

Code 5.10: Shape Class in Java

```java
// Circle.java
package com.oop.Shapes;

public class Circle extends Shape {
    private double radius;

    public Circle(String color, double radius) {
        setColor(color);
        this.radius = radius;
```

```
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
```

Code 5.11: Circle Class in Java

```
// Rectangle.java
package com.oop.Shapes;

public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(String color, double width, double height) {
        setColor(color);
        this.width = width;
        this.height = height;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return width * height;
```

```
33      }
34  }
```

Code 5.12: Rectangle Class in Java

Code 5.10 shows the implementation of the ‘Shape‘ abstract class in Java. The class contains a private attribute ‘color‘ and an abstract method ‘calculateArea()‘. Code 5.11 and Code 5.12 show the implementation of the ‘Circle‘ and ‘Rectangle‘ classes in Java. These classes are concrete subclasses of the ‘Shape‘ class and implement the ‘calculateArea()‘ method to calculate the area of a circle and rectangle, respectively.

5.5.2 Interfaces

Interfaces in Java are reference types that contain only abstract methods and constants. An interface provides a contract for classes to implement, specifying the methods that must be implemented by the classes that implement the interface. Classes can implement multiple interfaces, allowing them to provide specific implementations for the abstract methods defined in the interfaces.

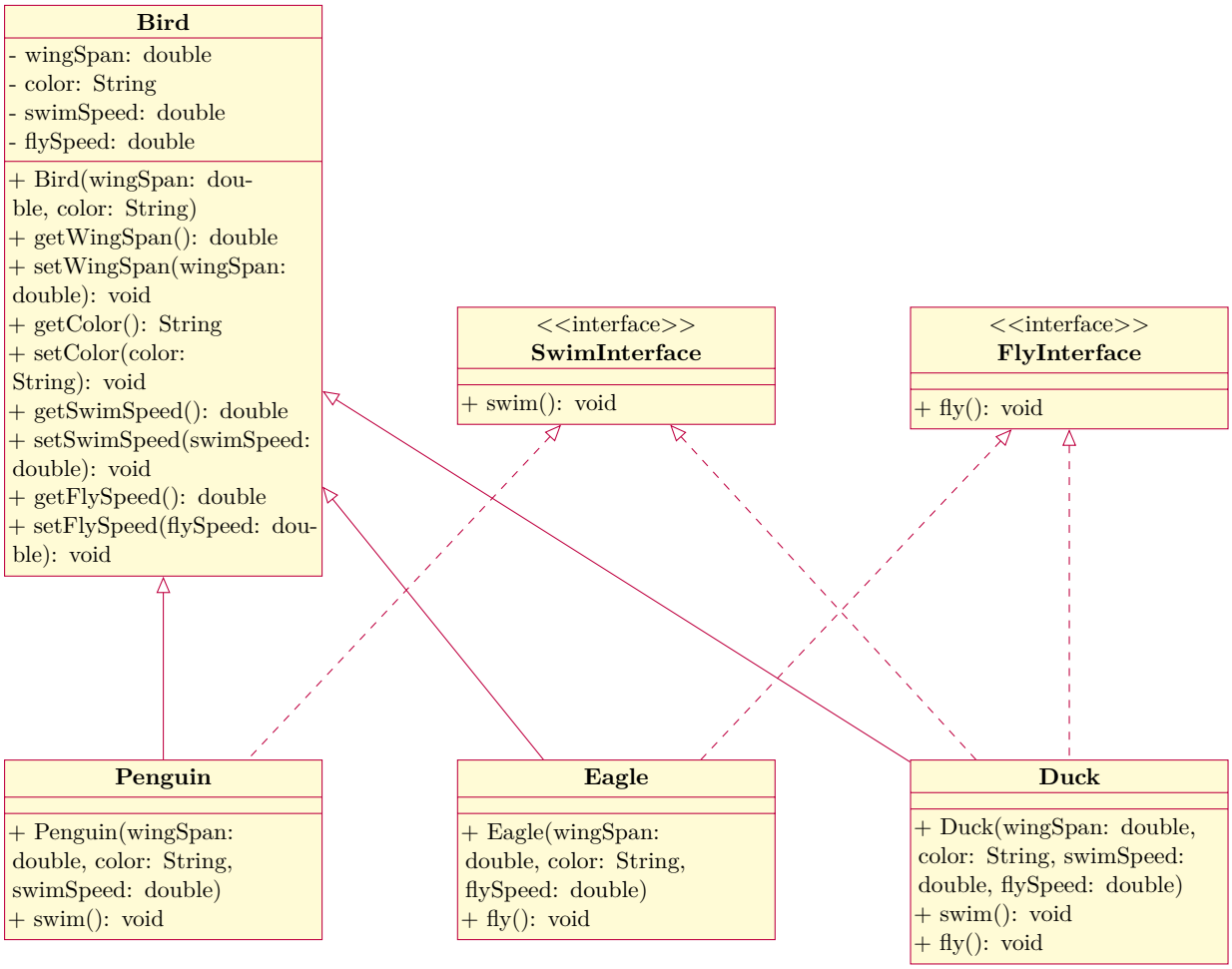


Figure 10: Interfaces Swim and Fly Implemented by Penguin, Eagle, and Duck Classes

Figure 10 illustrates the ‘Bird‘ class and its subclasses ‘Penguin‘, ‘Eagle‘, and ‘Duck‘ in Java. It also shows the ‘SwimInterface‘ and ‘FlyInterface‘ interfaces. The ‘Bird‘ class contains private attributes ‘wingSpan‘ and ‘color‘, and the ‘Penguin‘, ‘Eagle‘, and ‘Duck‘ classes inherit from the

‘Bird‘ class and implement the ‘SwimInterface‘ and ‘FlyInterface‘ interfaces. The ‘SwimInterface‘ and ‘FlyInterface‘ interfaces contain abstract methods ‘swim()‘ and ‘fly()‘, respectively, that must be implemented by the classes that implement the interfaces. In this example, the ‘Penguin‘ class implements the ‘swim()‘ method, the ‘Eagle‘ class implements the ‘fly()‘ method, and the ‘Duck‘ class implements both the ‘swim()‘ and ‘fly()‘ methods.

```java
// Bird.java
package com.oop.Birds;

public class Bird {
    private double wingSpan;
    private String color;
    private double swimSpeed;
    private double flySpeed;

    public Bird(double wingSpan, String color) {
        this.wingSpan = wingSpan;
        this.color = color;
    }

    // Getter and setter methods
    // Rest of the code
}
```

Code 5.13: Bird Class in Java

```java
// SwimInterface.java
package com.oop.Birds;

public interface SwimInterface {
    void swim();
}
```

Code 5.14: SwimInterface Interface in Java

```java
// FlyInterface.java
package com.oop.Birds;

public interface FlyInterface {
    void fly();
}
```

Code 5.15: FlyInterface Interface in Java

```java
// Penguin.java
package com.oop.Birds;

```

```
public class Penguin extends Bird implements SwimInterface {
    public Penguin(double wingSpan, String color, double swimSpeed) {
        super(wingSpan, color);
        setSwimSpeed(swimSpeed);
    }

    @Override
    public void swim() {
        System.out.println("Penguin swims at speed: " + getSwimSpeed());
    }
}
```

Code 5.16: Penguin Class in Java

```
// Eagle.java
package com.oop.Birds;

public class Eagle extends Bird implements FlyInterface {
    public Eagle(double wingSpan, String color, double flySpeed) {
        super(wingSpan, color);
        setFlySpeed(flySpeed);
    }

    @Override
    public void fly() {
        System.out.println("Eagle flies at speed: " + getFlySpeed());
    }
}
```

Code 5.17: Eagle Class in Java

```
// Duck.java
package com.oop.Birds;

public class Duck extends Bird implements SwimInterface, FlyInterface {
    public Duck(double wingSpan, String color, double swimSpeed, double
        flySpeed) {
        super(wingSpan, color);
        setSwimSpeed(swimSpeed);
        setFlySpeed(flySpeed);
    }

    @Override
    public void swim() {
        System.out.println("Duck swims at speed: " + getSwimSpeed());
    }

    @Override
    public void fly() {
```

```
18          System.out.println("Duck flies at speed: " + getFlySpeed());
19      }
20  }
```

Code 5.18: Duck Class in Java

Code 5.13 shows the implementation of the ‘Bird‘ class in Java. The class contains private attributes ‘wingSpan‘, ‘color‘, ‘swimSpeed‘, and ‘flySpeed‘. Code 5.14 and Code 5.15 show the implementation of the ‘SwimInterface‘ and ‘FlyInterface‘ interfaces in Java. Code 5.16, Code 5.17, and Code 5.18 show the implementation of the ‘Penguin‘, ‘Eagle‘, and ‘Duck‘ classes in Java. These classes inherit from the ‘Bird‘ class and implement the ‘SwimInterface‘ and ‘FlyInterface‘ interfaces. The ‘Penguin‘ class implements the ‘swim()‘ method, the ‘Eagle‘ class implements the ‘fly()‘ method, and the ‘Duck‘ class implements both the ‘swim()‘ and ‘fly()‘ methods.

5.6 Typecasting in Objects

Typecasting in Java is the process of converting an object of one type to another type. There are two types of typecasting in Java: **upcasting** and **downcasting**.

5.6.1 Upcasting (Implicit)

Upcasting is the process of converting an object of a subclass to an object of a superclass. Upcasting is implicit and does not require the use of the cast operator. Upcasting can be done automatically when an object of a subclass is assigned to a variable of the superclass.

```
1   // Animal.java
2   package com.oop.AnimalKingdom;
3
4   public class Animal {
5       // Rest of the code
6   }
7
8   // Dog.java
9   package com.oop.AnimalKingdom;
10
11  public class Dog extends Animal {
12      // Rest of the code
13  }
14
15  // Main.java
16  package com.oop.AnimalKingdom;
17
18  public class Main {
19      public static void main(String[] args) {
20          Animal animal = new Dog(); // Upcasting
21      }
22  }
```

Code 5.19: Upcasting in Java

Code 5.20 demonstrates upcasting in Java. The ‘Animal‘ class is the superclass, and the ‘Dog‘ class is the subclass. In the ‘Main‘ class, an object of the ‘Dog‘ class is created and assigned to a variable of the ‘Animal‘ type. Upcasting is implicit in this case because the object is of the ‘Dog‘ type.

5.6.2 Upcasting (Explicit)

Upcasting can also be done explicitly using the cast operator. Explicit upcasting is required when the object is originally created as an instance of the subclass.

```java
// Animal.java
package com.oop.AnimalKingdom;

public class Animal {
    // Rest of the code
}

// Dog.java
package com.oop.AnimalKingdom;

public class Dog extends Animal {
    // Rest of the code
}

// Main.java
package com.oop.AnimalKingdom;

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal animal = (Animal) dog; // Upcasting
    }
}
```

Code 5.20: Upcasting in Java

Code 5.20 demonstrates explicit upcasting in Java. The ‘Animal‘ class is the superclass, and the ‘Dog‘ class is the subclass. In the ‘Main‘ class, an object of the ‘Dog‘ class is created and assigned to a variable of the ‘Dog‘ type. The object is then upcasted to the ‘Animal‘ type using the ‘(Animal)‘ operator. Explicit upcasting is required in this case because the object is of the ‘Dog‘ type.

5.6.3 Downcasting (Explicit)

Downcasting is the process of converting an object of a superclass to an object of a subclass. Downcasting is explicit and requires the use of the cast operator. Downcasting can only be done if the object is originally created as an instance of the subclass.

```java
// Animal.java
package com.oop.AnimalKingdom;

```

```java
public class Animal {
    // Rest of the code
}

// Dog.java
package com.oop.AnimalKingdom;

public class Dog extends Animal {
    // Rest of the code
}

// Main.java
package com.oop.AnimalKingdom;

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        Dog dog = (Dog) animal; // Downcasting
    }
}
```

Code 5.21: Downcasting in Java

Code 5.21 demonstrates downcasting in Java. The ‘Animal‘ class is the superclass, and the ‘Dog‘ class is the subclass. In the ‘Main‘ class, an object of the ‘Dog‘ class is created and assigned to a variable of the ‘Animal‘ type. The object is then downcasted to the ‘Dog‘ type using the ‘(Dog)‘ operator. Downcasting is implicit in this case because the object is of the ‘Dog‘ type.

5.7 Summary

In this chapter, we discussed the concepts of inheritance, polymorphism, encapsulation, and abstraction in object-oriented programming. We explored how these concepts are implemented in Java using classes and interfaces. We also discussed typecasting in objects and demonstrated upcasting and downcasting in Java.

6

Enumeration

6.1 Summary

7

Coding Guidelines and Best Practices

8

References

A. **Books**

- Loy, M., Niemeyer, P., & Leuck, D. (2023). *Learning Java: An Introduction to Real-World Programming with Java*. O'Reilly Media. ISBN: 9781098145538
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals*. Pearson Education. ISBN: 978-0137673629
- Schildt, H. (2022). *Java: The Complete Reference, Twelfth Edition*. McGraw Hill Professional. ISBN: 978-1-260-46341-5
- Schildt, H. (2022). *Java: A Beginner's Guide, Ninth Edition*. McGraw Hill Professional. ISBN: 978-1-260-46355-2
- Paul Deitel & Harvey Deitel (July 14th 2021). *Java: How To Program, Early Objects, 11th edition*. Deitel & Associates, Inc. ISBN: 13:9780137505166
- Cay S. Horstmann (2019). *Brief Java: early objects 9th edition*. Wiley. ISBN: 978-1-119-49913-8
- Chemuturi, M. (2019). *Computer Programming for Beginners: A Step-By-Step Guide*. Chapman and Hall/CRC. ISBN: 978-1138320482
- Chua (2019). *Intermediate Programming Using C*. ISBN: 13 987-1498711630

B. **Other Sources**

- GeeksforGeeks (2022). *Object Oriented Programming (OOPs) Concept in Java*. GeeksforGeeks. https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/
- JavaTpoint (2022). *Java OOPs Concepts*. JavaTpoint. https://www.javatpoint.com/java-oops-concepts
- Tutorialspoint (2019). *Java Tutorial*. Tutorialspoint. https://www.tutorialspoint.com/java/index.htm
- W3Schools (2019). *Java Tutorial*. W3Schools. https://www.w3schools.com/java/