Graphics and Visual Computing [1]

Introduction to Java Programming Language and Object-Oriented Programming[2]

Jarrian Vince G. Gojar[3]

August 28, 2024

[1]A course in the Bachelor of Science in Computer Science
[2]A topic in the course "Object-Oriented Programming"
[3]https://github.com/godkingjay

Sorsogon State University - Bulan Campus

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

Introduction to Java Programming Language and Object-Oriented Programming

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.

History of Java

Before Java, its 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.

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.

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/`

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/

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

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.");
  }
}
```

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.”

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

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.

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

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.

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

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.

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

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.

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.

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.

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

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/

Lexical Translations

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.

```java
// Carriage Return (CR)
System.out.print("Hello, World!\r");

// Line Feed (LF)
System.out.print("Hello, World!\n");

// Carriage Return Line Feed (CRLF)
System.out.print("Hello, World!\r\n");

// Output
// Java, World!
```

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.

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.

```java
// Using White Space to Improve Readability
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");
  }
```

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.

```
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");}
```

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.

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.

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.

```
  // Valid Identifiers
  int number;
  String firstName;
  double averageScore;
  boolean isComplete;
  void printMessage();

  // Invalid Identifiers
  int 1number; // Cannot start with a digit
  String first-name; // Cannot contain hyphens
  double average score; // Cannot contain spaces
```

```java
boolean isComplete?; // Cannot contain special characters
void print Message(); // Cannot contain spaces
```

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.

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.

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.

```
// Integer Literal
int integerLiteral = 10;
System.out.println("Integer Literal: " + integerLiteral);

// Floating-Point Literal
double floatingPointLiteral = 3.14;
System.out.println("Floating-Point Literal: " + floatingPointLiteral);

// Character Literal
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);
```

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.

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]);

      // Using Semicolon to Terminate Statements
      int x = 10;
      int y = 20;
      int z = x + y;
      System.out.println("Sum: " + z);

      // Using Comma to Separate Variables
      int a = 1, b = 2, c = 3;
      System.out.println("Values: " + a + ", " + b + ", " + c);

      // Using Period to Access Members
      String text = "Hello, World!";
      int length = text.length();
      System.out.println("Length: " + length);

      // Using Ellipsis to Indicate Variable Arguments
      int sum = add(1, 2, 3, 4, 5);
      System.out.println("Sum: " + sum);
    }

    /**
    * Method to Calculate the Sum of Variable Arguments
    *
    * @param numbers the variable arguments to be added
    * @return the sum of the variable arguments
    */
    public static int add(int... numbers) {
      int sum = 0;
      for (int number : numbers) {
        sum += number;
      }
      return sum;
    }
  }
```

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.

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.

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: Overall evaluation of the system

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.

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.

```java
byte byteVariable = 100;
```

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.

```java
short shortVariable = 1000;
```

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;
```

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;
```

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.

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;
```

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.

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;
```

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.

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';
```

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;
```

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';
```

You can also use Unicode escape sequences to represent characters. For example, ’0̆041’ represents the character ’A’ in the Latin alphabet.

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;
```

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.

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.

Reference Data Types

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

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!");
```

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");
```

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");
```

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");
```

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");
```

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");
```

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");
```

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");
```

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");
```

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
\\ String Array: Hello World
```

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.

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.

```java
\\ Animal.java

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

    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.");
    }
}
```

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
}
```

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

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

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...
}
```

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() {
        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.");
    }
}
```

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.

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.

```
\\ 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.");
        }
    }
}
```

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

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

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

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...
}
```

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);
}
```

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;
}
```

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.");
        }
    }
}
```

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.

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;

    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) {
        super(name, species, age, weight, color, habitat, x, y, fur, milk,
            isWarmBlooded, canGiveBirth);
        this.breed = breed;
        this.barkingSound = barkingSound;
        this.tailWagging = tailWagging;
        this.isTrained = isTrained;
    }

    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;
    }

    public void bark() {
        System.out.println("The dog barks: " + barkingSound);
    }

    public void fetch() {
        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.");
        }
    }
}
```

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...
}
```

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;
}
```

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;
}
```

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.

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.

```java
\\ Bird.java

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

    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) {
        super(name, species, age, weight, color, habitat, x, y);
        this.wings = wings;
        this.hasFeathers = hasFeathers;
        this.canFly = canFly;
        this.canLayEggs = canLayEggs;
    }

    public int getWings() {
        return wings;
    }

    public void setWings(int wings) {
        this.wings = wings;
    }

    public boolean isHasFeathers() {
        return hasFeathers;
    }

    public void setHasFeathers(boolean hasFeathers) {
        this.hasFeathers = hasFeathers;
    }

    public boolean canFly() {
        return canFly;
    }
    public void setCanFly(boolean canFly) {
        this.canFly = canFly;
    }

    public boolean canLayEggs() {
        return canLayEggs;
    }

    public void setCanLayEggs(boolean canLayEggs) {
        this.canLayEggs = canLayEggs;
    }

    public void fly() {
        if (canFly && hasFeathers) {
            System.out.println(name + " is flying.");
        } else {
            System.out.println(name + " cannot fly.");
        }
    }

    public void layEggs() {
        if (this.age >= 1 && canLayEggs) {
            System.out.println(name + " is laying eggs.");
        } else {
            System.out.println(name + " is too young to lay eggs.");
        }
    }

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

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.

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

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.

```
\\ Animal.java (makeSound() method)
void makeSound() {
    System.out.println(this.name + " makes a sound.");
}
```

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.

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;
    }

    public boolean canLayEggs() {
        return canLayEggs;
    }

    public void setCanLayEggs(boolean canLayEggs) {
        this.canLayEggs = canLayEggs;
    }

    public void shedSkin() {
        if (hasScales) {
            System.out.println(name + " is shedding its skin.");
        } else {
            System.out.println(name + " does not have scales and cannot shed its
                skin.");
        }
    }

    public void layEggs() {
        if (canLayEggs) {
            System.out.println(name + " is laying eggs.");
        } else {
            System.out.println(name + " cannot lay eggs.");
        }
    }

    public void regenerateTail() {
        if (canRegenerateTail && hasTail) {
            System.out.println(name + " is regenerating its tail.");
        } else {
            System.out.println(name + " cannot regenerate its tail.");
        }
    }
}
```

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.

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

    // Constructor...

    public abstract void biteHuman();

    // Getter and setter methods...

    // Methods...
}
```

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()”.

```java
\\ 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...
}
```

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.

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.

```java
\\ 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();
    }
}
```

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.

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);
```

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();
```

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);
```

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”.

Polymorphism

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

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.

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());
```

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.

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();
```

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.

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.

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/