Week 2

1. **INHERITANCE**

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a class to inherit properties and behaviors (methods) from another class, known as the superclass or base class. In Java, inheritance enables code reuse, promotes code organization, and facilitates the creation of hierarchical class structures.

Basic Syntax:

In Java, inheritance is achieved using the extends keyword to create a subclass that inherits from a superclass.

|  |
| --- |
| class Animal {  void eat() {  System.out.println("Animal is eating");  } }  class Dog extends Animal {  void bark() {  System.out.println("Dog is barking");  } } |

**Types of Inheritance:**

* Single Inheritance: A subclass inherits from only one superclass.
* Multilevel Inheritance: A subclass inherits from a superclass, and then another subclass inherits from that subclass, creating a chain of inheritance.
* Hierarchical Inheritance: Multiple subclasses inherit from a single superclass.
* Multiple Inheritance (Interface): A subclass can implement multiple interfaces, but Java does not support inheriting from multiple classes.

A diagram of a class

Description automatically generated

**Fig 1: Types of Inheritance**

1. **LEXICAL ISSUES IN JAVA**

Lexical issues in Java typically refer to problems related to the structure and syntax of the Java programming language. Here are some common lexical issues that Java developers may encounter:

* Reserved Keywords: Using reserved keywords as identifiers (such as variable names, class names, or method names) will result in a lexical error. For example, using "int" as a variable name is not allowed because "int" is a keyword in Java.
* Incorrect Identifier Naming: Identifiers in Java must follow certain rules, such as starting with a letter, dollar sign, or underscore, and can contain letters, digits, dollar signs, or underscores. Using invalid characters in identifiers will lead to lexical errors.
* Missing Semicolons: Java statements must end with semicolons. Omitting a semicolon at the end of a statement will result in a lexical error.
* Mismatched Braces: Java uses braces ({}) to define blocks of code, such as method bodies and class definitions. Forgetting to close a brace or using braces incorrectly can cause lexical errors.
* Unmatched Quotes: String literals in Java must be enclosed in double quotes (""). Forgetting to close a string literal with a double quote will result in a lexical error.
* Misplaced Operators: Operators in Java must be used in accordance with the syntax rules of the language. Misplacing operators or using them incorrectly can lead to lexical errors.
* Invalid Characters: Using characters that are not allowed in Java source code, such as certain special characters or control characters, will result in lexical errors.
* Incorrect Comments: Comments in Java must be properly formatted. Using incorrect comment syntax or failing to close a comment properly can cause lexical errors.
* Invalid Escape Sequences: Using invalid escape sequences in string literals or character literals will result in lexical errors. For example, "\z" is an invalid escape sequence in Java.
* Using Deprecated Features: Using deprecated features or APIs may not result in direct lexical errors, but it's considered bad practice and can lead to issues in the long term.

1. **WHITESPACES IN JAVA**

In Java, whitespace refers to characters that are used for spacing within the source code but do not affect the logic of the program. Java allows the use of whitespace characters such as spaces, tabs, and newline characters in various contexts. Here are some common uses of whitespace in Java:

Spaces and Tabs: Spaces and tabs are used for indentation to improve code readability. They are typically used to visually represent the structure of the code, such as indenting blocks of code within methods, loops, or conditional statements.

|  |
| --- |
| public class Example {  public static void main(String[] args) {  int x = 5;  if (x > 0) {  System.out.println("x is positive");  }  }  } |

Newline Characters: Newline characters (line breaks) are used to separate lines of code. They are used to mark the end of a line and indicate where one statement ends and the next begins.

|  |
| --- |
| public class Example {  public static void main(String[] args) {  int x = 5;  System.out.println("Hello, Java!");  }  } |

Whitespace within Expressions: Whitespace can be used to separate tokens within expressions for clarity. For example, spaces are used to separate operators and operands in arithmetic expressions.

|  |
| --- |
| int result = (10 + 5) \* 2; |

Whitespace in String Literals: Whitespace characters can also be included within string literals. For example, spaces, tabs, and newline characters can be part of the content of a string.

Whitespace in Method Signatures and Declarations: Whitespace is used to separate keywords, identifiers, parameters, and other elements in method signatures and declarations.

1. **IDENTIFIERS IN JAVA**

In Java, an identifier is a name given to a variable, method, class, package, or other program element. Identifiers are used to uniquely identify these program elements within the scope of the program. Here are the rules for naming identifiers in Java:

* Valid Characters: Identifiers can only contain letters (a-z, A-Z), digits (0-9), underscores (\_), and dollar signs ($). The first character of an identifier cannot be a digit.
* No Spaces: Identifiers cannot contain spaces.
* Reserved Keywords: Identifiers cannot be the same as Java keywords. Keywords are reserved for specific purposes in the language and cannot be used as identifiers. Examples of keywords include class, int, public, static, etc.
* Case Sensitivity: Java is case-sensitive, so uppercase and lowercase letters are considered different. Therefore, myVariable, MyVariable, and MYVARIABLE are considered distinct identifiers.
* Length: Identifiers can be of any length, but it's good practice to keep them concise and meaningful.
* Unicode Characters: Java allows the use of Unicode characters in identifiers. This means you can use characters from other languages, such as accented letters or symbols, though it's generally advisable to stick to ASCII characters for portability and read**ability.**

|  |  |
| --- | --- |
| **Valid identifiers** | **Invalid Identifiers** |
| int myVariable;  String userName;  void calculateArea() {}  class MyClass {}  final double PI = 3.14159; | int 2variable; // Starts with a digit  float my Variable; // Contains space  String class; // Reserved keyword  boolean my-variable; // Contains hyphen |

1. **JAVA LITERALS**

Variables are critical in programming because that’s how you store data in a particular memory location. For example, a Java program, while running, stores values in containers known as “variables,” which are defined as a basic storage unit. To enhance the program’s readability, the programmer must follow particular conventions while naming these variables and assigning values to them. For example, a source code representing a fixed value is "literal.”

Literals in Java are a synthetic representation of boolean, character, numeric, or string data. They are a means of expressing particular values within a program. They are constant values that directly appear in a program and can be assigned now to a variable.

**Types of Literals in Java**

Literals in Java are typically classified into six types and then into various sub-types. The primary literal types are:

**1. Integral Literals**

Integral literals consist of digit sequences and are broken down into these sub-types:

**Decimal Integer:** Decimal integers use a base ten and digits ranging from 0 to 9. They can have a negative (-) or a positive (+), but non-digit characters or commas aren’t allowed between characters. Example: 2022, +42, -68.

**Octal Integer**: Octal integers use a base eight and digits ranging from 0 to 7. Octal integers always begin with a “0.” Example: 007, 0295.

**Hexa-Decimal:** Hexa-decimal integers work with a base 16 and use digits from 0 to 9 and the characters of A through F. The characters are case-sensitive and represent a 10 to 15 numerical range. Example: 0xf, 0xe.

**Binary Integer:** Binary integers uses a base two, consisting of the digits “0” and “1.” The prefix “0b” represents the Binary system. Example: 0b11011.

**Floating-Point Literals**

Floating-point literals are expressed as exponential notations or as decimal fractions. They can represent either a positive or negative value, but if it’s not specified, the value defaults to positive. Floating-point literals come in these formats:

**Floating:** Floating format single precision (4 bytes) end with an “f” or “F.” Example: 4f. Floating format double precision (8 bytes) end with a “d” or “D.”

Example: 3.14d.

**Decimal:** This format uses 0 through 9 and can have either a suffix or an exponent. Example: 99638.440.

**Decimal in Exponent form:** The exponent form may use an optional sign, such as a "-," and an exponent indicator, such as "e" or "E." Example: 456.5f.

**Char Literals**

Character (Char) literals are expressed as an escape sequence or a character, enclosed in single quote marks, and always a type of character in Java. Char literals are sixteen-bit Unicode characters ranging from 0 to 65535. Example: char ch = 077.

**String Literals**

String literals are sequences of characters enclosed between double quote ("") marks. These characters can be alphanumeric, special characters, blank spaces, etc.

Examples: "John", "2468", "\n", etc.

**Boolean Literals**

Boolean literals have only two values and so are divided into two literals:

* True represents a real boolean value
* False represents a false boolean value

So, Boolean literals represent the logical value of either true or false. These values aren't case-sensitive and are equally valid if rendered in uppercase or lowercase mode. Boolean literals can also use the values of “0” and “1.”

**6. Null Literals**

Null literals represent a null value and refer to no object. Nulls are typically used as a marker to indicate that a reference type object isn’t available. They often describe an uninitialized state in the program. It is a mistake to try to dereference a null value. Example: Patient age = NULL;

1. **KEYWORDS IN JAVA**

Java has a few reserved words used to represent specific functionalities. They can’t be used for any other purposes and to name a variable. There are total of 53 keywords in Java.

A white rectangular table with black text

Description automatically generated

**Fig 2: Java keywords**

1. **JAVA DATATYPES**

A data type is a classification of data. It tells the compiler or interpreter how the programmer aims to use the variables or method. Data types are a crucial factor in all computer programming languages. The task of a programmer is to develop a workable program by assigning the right types of data to the right variables. Data types represent the type, nature, and set of operations for the value which they store. There are two data types of categories in Java: Primitive and Non-Primitive.

A diagram of different types of data

Description automatically generated

**Fig 3: Datatypes in java**

**Integer type:**

An integer type stores an integer number with no fractional or decimal places. Java has four integer types – byte, short, int, and long.

**Byte:**

The byte is the smallest data type among all the integer data types. It is an 8-bit signed two’s complement integer. It stores whole numbers ranging from -128 to 127.

**Syntax:** byte byteVariable;

**Short**

Short is a 16-bit signed two’s complement integer. It stores whole numbers with values ranging from -32768 to 32767. Its default value is 0.

**Syntax:** short shortVariable;

**Int**

Int is a 32-bit signed two’s complement integer that stores integral values ranging from 2147483648 (-2^31) to 2147483647 (2^31 -1). Its default value is 0.

**Syntax:** int intVariable;

**Long**

long is a 64-bit signed two’s complement integer that stores values ranging from -9223372036854775808(-2^63) to 9223372036854775807(2^63 -1). It is used when we need a range of values more than those provided by int. Its default value is 0L. This data type ends with ‘L’ or ‘l’.

**Syntax:** long longVariable;

|  |
| --- |
| class IntegerDataTypes  {  public static void main(String args[]) {  int a = 10;  short s = 2;  byte b = 6;  long l = 125362133223l;    System.out.println("The integer variable is " + a + '\n');  System.out.println("The short variable is " + s + '\n');  System.out.println("The byte variable is " + b + '\n');  System.out.println("The long variable is " + l);  }  } |

**Output:**

**A close-up of a white background

Description automatically generated**

**boolean**

A boolean is actually one of the most simple primitive data types in Java. As you may already know, a boolean can contain only 2 values: true or false. A boolean is stored in just one bit of data. But, for convenience, Java stores a boolean in a single byte instead of just a bit.

|  |
| --- |
| public class BoolTest {  public static void main(String[] args) {  boolean myBoolean = false;  int a = 5;  int b = 7;  System.out.println(a < b);  System.out.println(0 > 7);  System.out.println(myBoolean == false);  }  } |

Output:

true

false

true

**char**

It is also called a character and is stored in 16 bits of memory that represent a Unicode-encoded character. The range of a char type is from 0 to 65,535. This represents \u0000' to ‘\uffff' in Unicode.

1. **VARIABLES DECLARATION IN JAVA**

Java Variables or variables in any other programming language are containers, which hold some value. Because Java is a strongly typed language, so every variable must be declared and initialized or assigned before it is used. A variable, in the simplest way, is declared by placing a valid type followed by the variable name. And so, don't forget to place a semicolon at the end of every statement because it completes the statement. Throughout the tutorial we will use terms 'variable' and 'field' interchangeably as they refer to the same thing and there should be no confusion regarding that. Following statements demonstrate variable declaration in Java.

**Rules for naming variables:**

* Variable names are case-sensitive.
* Variable name can be an unlimited-length sequence of Unicode letters and digits.
* Variable name must begin with either a letter or the dollar sign "$", or the underscore character "\_".
* Characters except the dollar sign "$" and the underscore character "\_" for example, '+' or '?' or '@' are invalid for a variable name.
* No white space is permitted in variable names.
* Variable name we choose must not be a keyword or reserved word.

**Java Variable Types**

Java variables are categorized in different types depending upon where are they declared? For example, in a class or in a method's body. Java defines following four types of variables.

**Local Variables**

As name suggests, a local Java variable is scoped to the block, which is between the opening and closing braces. Local variables are declared within a block or constructor or method's body. Local variables are created when control enters into the block or constructor of method's body and are destroyed once the control exits from the body of method or constructor or block.

Local variables are only visible to the methods or blocks or constructors in which they are declared; they are not accessible from the rest of the class. There is no designated access modifier for local variables. Also, there is no default value for local variables, therefore, local variables must be initialized or assigned some value before they are used first time.

**Instance Variables or Non-Static Fields**

Data members or fields or variables of a class that are declared non-static, but outside a method, constructor or any block are known as instance variables. Instance variables are created when an object of a class is created by using new keyword. Objects store their states in non-static instance variables. Instance variables are declared private or public or protected or default (no keyword). Instance variables are initialized to some default values by the compiler, in case they are not initialized by the creator of class. Instance variables are destroyed along with the object they have been created for. The following program declares some non-static instance variables:

|  |
| --- |
| class Book  {  //instance variables  private String title;  private String publisher;  private int numOfPages;    //zero argument constructor needed when  //one or more argument constructors are defined  public Book() {}    public Book (String title, String publisher, int numOfPages)  {  this.title = title;  this.publisher = publisher;  this.numOfPages = numOfPages;  }    public void printBookDetails()  {  System.out.println("Title: " + title);  System.out.println("Publisher: " + publisher);  System.out.println("Num of Pages: " + numOfPages);  }  } |

**Class Variables or Static Fields**

Class variables in Java are fields declared with static keyword. Modifier static informs the compiler that there will be only one copy of such variables created regardless of how many objects of this class are created. Static variables are created when a class is loaded into the memory by the class loader and destroyed when a class is destroyed or unloaded. Visibility of static fields will depend upon access modifiers. Default vales given to the static members will follow the same rule as it is done in instance variables.

**Method Parameters**

While passing values to member methods we need parameter variables. Parameters are not fields; they are always called variables and used only to copy their values into the formal parameters. Variables or values passed to a method by caller are called actual parameters, and the callee receives these values in formal parameters. Formal parameters are local variables to that particular method.

**Java Variables - Dynamic Initialization**

Initialization is the process of providing value to a variable at declaration time. A variable is initialized once in its life time. Any attempt of setting a variable's value after its declaration is called assignment. To use a local variable you have to either initialize or assign it before the variable is first used. But for class members, the compulsion is not so strict. If you don't initialize them then compiler takes care of the initialization process and set class members to default values.

Java allows its programmers to initialize a variable at run time also. Initializing a variable at run time is called dynamic initialization. The following piece of code (DynamicInitializationDemo.java) demonstrates it.

|  |
| --- |
| public class DynamicInitializationDemo  {  public static void main(String[] args)  {  //dynSqrt will be initialized when Math.sqrt  //will be executed at run time  double dynSqrt = Math.sqrt (16);  System.out.println("sqrt of 16 is : " + dynSqrt);  }  }  OUTPUT  ======  sqrt of 16 is : 4.0 |