**JAVA**

* **Overview**
* Java is a high-level, object-oriented programming language that was developed by Sun Microsystems in the mid-1990s.
* It was designed to be platform-independent, meaning that Java programs can run on any computer or device that has a Java Virtual Machine (JVM) installed, regardless of the underlying hardware or operating system. One of the key features of Java is its "write once, run anywhere" (WORA) capability.
* Java is also known for its robustness, reliability, and security. It has a strong type of system and automatic memory management, which helps to prevent common programming errors such as memory leaks and null pointer exceptions. Additionally, Java has built-in support for encryption, authentication, and other security features, making it a trusted choice for developing secure software.
* Java is widely used in the enterprise and web development communities and is also popular in the academic world for teaching programming concepts. The Java platform includes many standard libraries and tools, making it easy to develop complex applications with minimal coding effort. Additionally, the Java community is large and active, with many open-source libraries, frameworks, and development tools available for free.
* **JVM setup**
* Install the Java Development Kit (JDK): The JDK is a software development kit that includes the Java Runtime Environment (JRE), the Java compiler, and other development tools. You can download the latest version of the JDK from the official Java website (<https://www.oracle.com/java/technologies/javase-downloads.html>).
* Once you download the JDK installer, run it and follow the installation instructions. Set up the Java environment variables: After installing the JDK, you need to set up the Java environment variables. These variables tell your operating system where to find the Java executables and libraries.
* To set up the Java environment variables, follow these steps: a. On Windows: Open the Control Panel, go to System and Security, and then click on System. Click on the "Advanced system settings" link, and then click on the "Environment Variables" button. Under "System variables", click on the "New" button and add a new variable named "JAVA\_HOME" with the path to the JDK installation directory. Then, add the "bin" directory of the JDK installation to the "Path" variable.
* **Basic syntax**
* Java is case-sensitive.
* Java class names start with a capital letter.
* Java method names start with a lowercase letter.
* Java statements end with a semicolon: In Java, statements must end with a semicolon.
* Java blocks are enclosed in curly braces: In Java, blocks of code are enclosed in curly braces.
* Java comments start with "//" or "/" and end with "/".
* **Objects and classes**
* In Java, objects are instances of classes. A class is a blueprint or template that defines the properties and behavior of a certain type of object. When you create an object from a class, you are creating an instance of that class. For example, suppose we have a class called "Person" that defines the properties and behavior of a person. The class might have properties such as name, age, and address, as well as methods for setting and getting those properties. To create an object of type "Person", we would write code like this:
* Person john = new Person();
* This code creates a new object of type "Person" and assigns it to a variable called "john". The "new" keyword is used to create a new object, and the parentheses after the class name are used to specify any arguments that are passed to the class constructor. Once we have created an object, we can access its properties and methods using dot notation.
* john.name = "John Doe";
* This code sets the "name" property of the "john" object to "John Doe". We can also call methods on the object, like this
* john.sayHello();
* This code calls the "sayHello" method of the "john" object. Overall, classes and objects are a fundamental concept in Java programming. Classes define the structure and behavior of objects, while objects are instances of those classes that can be created and manipulated in a Java program.
* **Constructors**
* In Java, a constructor is a special method that is used to create objects of a class. Constructors have the same name as the class, and they are called automatically when an object is created using the new keyword.
* Constructors can be used to initialize the instance variables of a class, and they can take input parameters to customize the initialization process. Constructors can also be overloaded, which means that you can define multiple constructors with different parameter lists. Here is an example of a simple constructor for a class called Person:
* public class Person { private String name; private int age; public Person(String name, int age) { this.name = name; this.age = age; } // other methods and variables }
* In this example, the Person class has a constructor that takes two parameters: name and age. Inside the constructor, the values of name and age are assigned to the instance variables with the same names using this keyword. Constructors are an important part of object-oriented programming in Java, as they provide a way to initialize objects with default or customized values.
* **Datatypes**
* In Java, there are eight primitive data types:
* byte: A byte is a 8-bit signed integer that can store values from -128 to 127.
* short: A short is a 16-bit signed integer that can store values from -32,768 to 32,767.
* int: An int is a 32-bit signed integer that can store values from -2,147,483,648 to 2,147,483,647.
* long: A long is a 64-bit signed integer that can store values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
* float: A float is a 32-bit floating point number that can store values with a range of approximately 1.4E-45 to 3.4E+38.
* double: A double is a 64-bit floating point number that can store values with a range of approximately 4.9E-324 to 1.8E+308.
* boolean: A boolean is a data type that can store either true or false.
* char: A char is a 16-bit Unicode character that can store a single character.
* **Variable types**
* Local Variables: A local variable is a variable that is declared inside a method or block of code.
* Global/Instance Variables: An instance variable is a variable that is declared inside a class, but outside of any method. Instance variables are also called non-static variables because they belong to a specific instance of the class, rather than to the class itself. Instance variables are initialized to default values if they are not explicitly initialized.
* Class/Static Variables: A class/static variable is a variable that is declared with the "static" keyword inside a class, but outside of any method. Class variables are also called static variables because they belong to the class itself, rather than to a specific instance of the class. Class variables are initialized to default values if they are not explicitly initialized.
* **Modifier types**
* Modifiers are keywords that are used to modify the declaration of classes, methods, variables, and other program elements.
* Access Modifiers: Access modifiers control the accessibility of classes, methods, and variables. There are four access modifiers in Java:
  + public: Public elements are accessible from anywhere in the program.
  + private: Private elements are only accessible within the same class.
  + protected: Protected elements are accessible within the same package and from subclasses outside the package.
  + default (no modifier): Default elements are only accessible within the same package
* Non-access modifiers are used to modify the behavior of classes, methods, and variables. There are several non-access modifiers in Java:
  + static: Static elements belong to the class itself, rather than to a specific instance of the class.
  + final: Final elements cannot be modified after they are initialized.
  + abstract: Abstract classes and methods cannot be instantiated and must be subclassed and implemented by concrete classes.
  + synchronized: Synchronized methods can only be accessed by one thread at a time.
  + transient: Transient variables are not serialized when an object is written to a stream.
  + volatile: Volatile variables are not cached, and their value is always read from main memory.
* **Basic Operators, loops, and decision.**
* Operators: Operators are symbols that are used to perform operations on variables and values.
  + Arithmetic operators: +, -, \*, /, %
  + Assignment operators: =, +=, -=, \*=, /=, %=
  + Comparison operators: ==, !=, >, <, >=, <=
  + Logical operators: &&, ||, !
  + Bitwise operators: &, |, ^, ~, <<, >>
* Loop Control: Loop control is used to execute a block of code repeatedly.
  + for loop: A for loop is used to execute a block of code a fixed number of times.
  + while loop: A while loop is used to execute a block of code as long as a specified condition is true.
  + do-while loop: A do-while loop is similar to a while loop, but the code is executed at least once before the condition is checked.
* Decision Making: Decision making is used to execute a block of code based on a specified condition.
  + if statement: An if statement is used to execute a block of code if a specified condition is true.
  + if-else statement: An if-else statement is used to execute one block of code if a condition is true, and a different block of code if the condition is false.
  + switch statement: A switch statement is used to execute one of several blocks of code based on the value of a variable.
* **Strings**
* The String class is part of the java.lang package and provides a wide range of methods for manipulating strings. Strings are widely used in Java programs for storing and processing textual data. String objects are immutable, which means that once a string object is created, its value cannot be changed. Instead, when you modify a string, a new string object is created with the modified value. This can be inefficient if you need to perform a lot of string manipulation, as it can lead to a lot of object creation and memory allocation.
* Here are some common methods of the String class:
  + length(): returns the length of the string.
  + charAt(index): returns the character at the specified index.
  + substring(startIndex, endIndex): returns a substring of the original string from startIndex to endIndex (exclusive).
  + indexOf(str): returns the index of the first occurrence of the specified substring, or -1 if the substring is not found. equals(str): returns true if the string is equal to the specified string, false otherwise.
  + compareTo(str): compares the string to the specified string and returns a negative value if the string is less than the specified string, zero if the strings are equal, or a positive value if the string is greater than the specified string.
  + toUpperCase(): returns a new string with all characters converted to uppercase.
  + toLowerCase(): returns a new string with all characters converted to lowercase.
  + trim(): returns a new string with leading and trailing whitespace removed.
  + Strings can also be concatenated using the + operator, or using the concat() method.
* In addition to the String class, Java also provides other classes for working with textual data, such as the StringBuilder and StringBuffer classes, which are used for efficient string manipulation, and the Pattern and Matcher classes, which are used for regular expression matching.
* **Arrays**
* In Java, an array is a data structure that stores a fixed-size sequential collection of elements of the same type. Arrays are widely used in Java programs for storing and processing data. An array is declared by specifying its type and size. For example,
* int[] myArray = new int[5];
* Arrays can be initialized with values using an initializer list. For example, to create an array of integers with the values 1, 2, 3, 4, and 5, you would use the following syntax:
* int[] myArray = {1, 2, 3, 4, 5};
* Arrays can also be sorted using the Arrays.sort() method, and can be searched using the Arrays.binarySearch() method.
* Here are some common methods of the Arrays class:
  + sort(arr): sorts the elements of the array in ascending order.
  + binarySearch(arr, key): searches for the specified key in the array using a binary search algorithm and returns the index of the key, or a negative value if the key is not found.
  + toString(arr): returns a string representation of the array.
  + fill(arr, val): sets all elements of the array to the specified value.
* Arrays can also be multidimensional. For example, to create a 2D array of integers with a size of 3x3.
* int[][] myArray = new int[3][3]; This creates an array with 3 rows and 3 columns. You can access the elements of a multidimensional array using two indexes, one for the row and one for the column.
* **Date&Time, Regular Expressions.**
* Date and Time: Java provides the java.time package, which contains classes for working with date and time values. Some of the important classes in this package include:
  + LocalDate: represents a date (year, month, and day)
  + LocalTime: represents a time (hours, minutes, and seconds)
  + LocalDateTime: represents both a date and time
  + ZonedDateTime: represents a date and time with a time zone You can use these classes to perform various operations, such as parsing and formatting dates and times, calculating durations, adding or subtracting time, and more.
* Regular Expressions: Java supports regular expressions (regex), which are patterns used to match text. The java.util.regex package contains classes for working with regex, including:
  + Pattern: represents a compiled regex pattern
  + Matcher: used to match a pattern against a text string
  + PatternSyntaxException: thrown when an invalid regex pattern is used.
* **I/O files and package**
* The Java I/O (Input/Output) package provides classes and interfaces for performing input/output operations in Java. It provides a way for Java programs to read from and write to files, streams, and other sources and sinks of data.
* The Java I/O package consists of two main types of classes: streams and readers/writers. Streams are used to handle binary data, while readers and writers are used to handle text data.
* The java.io package provides various classes for file I/O operations such as FileInputStream, FileOutputStream, FileReader, FileWriter, BufferedReader, BufferedWriter, etc.
* The File class is used to represent a file or directory in the file system. You can use the File class to create, delete, and rename files and directories. The File class also provides methods for checking whether a file or directory exists, and whether it is a file or directory. Here are some common classes and methods used for file I/O operations in Java:
  + FileInputStream/FileOutputStream: used for reading/writing binary data from/to a file.
  + FileReader/FileWriter: used for reading/writing text data from/to a file.
  + BufferedReader/BufferedWriter: used for reading/writing text data from/to a character stream (such as a file).
  + Scanner: used for reading formatted input from a variety of sources, including files and streams.
  + PrintWriter: used for writing formatted output to a variety of destinations, including files and streams.
  + File: used for creating, deleting, and renaming files and directories, and for checking whether a file or directory exists and is a file or directory.
* **Exceptions and innerclasses**
* Exceptions: Java provides a mechanism for handling errors and unexpected events called exceptions. An exception is an object that represents an error condition or unexpected situation that occurred during program execution. Exceptions can be caught and handled using try-catch blocks, which allow you to gracefully recover from errors and take appropriate actions.
* Java provides a hierarchy of exception classes that you can use to represent different types of errors. These classes are organized in a tree-like structure, with the base class being the Throwable class. Some of the important exception classes in Java include:
  + RuntimeException: represents exceptions that can occur during runtime, such as null pointer exceptions and arithmetic exceptions.
  + IOException: represents exceptions that can occur during I/O operations, such as file not found or network errors. Exception: the base class for all checked exceptions.
* Inner Classes: In Java, you can define a class inside another class, which is known as an inner class or nested class. Inner classes have access to the members of the enclosing class, including private members. Inner classes are used to encapsulate related functionality and improve code organization and modularity.
* There are four types of inner classes in Java:
  + Static nested class: a static nested class is a static class that is defined inside another class.
  + Inner class: an inner class is a non-static class that is defined inside another class.
  + Local class: a local class is a non-static class that is defined inside a method or a block of code.
  + Anonymous class: an anonymous class is a local class that is defined without a name.
* Inner classes can be used to implement callbacks, define event listeners, and improve code organization and encapsulation.
* **Encapsulation**
* Encapsulation is one of the fundamental concepts of object-oriented programming in Java. It refers to the process of hiding the implementation details of a class from other classes, and providing a well-defined interface for accessing and modifying the properties of the class. In Java, encapsulation is typically achieved by making the properties of a class private, and providing public methods (also called accessors) for accessing and modifying those properties. By doing this, we can ensure that the internal state of the class is protected from unwanted modification, and that the class always remains in a valid state.
* **Inheritance**
* Inheritance is one of the key concepts of object-oriented programming in Java. It allows one class (the subclass) to inherit properties and methods from another class (the superclass). In Java, we use the extends keyword to define a subclass that is inherited from a superclass.
* Main benefits of inheritance is code reusability. By defining a superclass with common properties and methods, we can reuse that code in multiple subclasses. In addition, inheritance allows us to create a hierarchy of related classes, with each subclass adding more specialized behavior.
* Java only supports single inheritance, which means that a class can only inherit from one superclass at a time. However, Java also supports interface inheritance, which allows a class to implement multiple interfaces, each of which defines a set of methods that the class must implement.
* **Polymorphism**
* Polymorphism is another key concept in object-oriented programming in Java. It allows us to write code that can work with objects of different types, if those objects share a common interface or superclass.
* There are two main types of polymorphism in Java: method overloading and method overriding.
  + Method overloading is when we define multiple methods with the same name in a class, but with different parameter lists. Java uses the number and types of the parameters to determine which method to call at runtime.
  + Method overriding is when we define a method in a subclass that has the same name and signature (i.e., the same name and parameter list) as a method in the superclass. The subclass method can provide custom behavior for that method, which will be called instead of the superclass method when the method is called on an object of the subclass.
* Polymorphism allows us to write code that is more flexible and adaptable, since it can work with objects of different types as long as they share a common interface or superclass.
* **Abstraction**
* Abstraction is the process of hiding implementation details while showing only the necessary information to the user. In Java, abstraction can be achieved through abstract classes and interfaces.
* An abstract class is a class that cannot be instantiated, which means you cannot create objects of that class. Abstract classes are meant to be extended by other classes, which can then provide their own implementation for the abstract methods in the abstract class. An abstract method is a method that has no implementation in the abstract class, but is meant to be implemented by its concrete subclasses.
* Abstraction is a powerful tool in Java that allows you to write code that is easier to understand, modify, and maintain. By hiding implementation details behind abstract classes and interfaces, you can create more flexible and modular software systems that are easier to extend and adapt to changing requirements.
* **Interfaces**
* In Java, an interface is a collection of abstract methods and constants that are meant to be implemented by any class that wants to be considered a type of that interface. An interface defines a set of methods that a class must implement if it wants to be considered a type of that interface. Any class that implements an interface must provide an implementation for all the methods defined in the interface.
* Interfaces are a powerful tool in Java that allows you to write code that is more flexible and modular. By using interfaces, you can define a common set of methods that different classes can implement, allowing you to write more flexible and reusable code.
* **Packages**
* In Java, a package is a namespace that contains a group of related classes, interfaces, and sub-packages. Packages help to organize code into logical units, which makes it easier to manage and maintain code. By using packages, you can avoid naming conflicts between classes with the same name that may exist in different parts of your program. The naming convention for packages in Java is to use a reverse domain name (e.g., com.example.mypackage). To create a package in Java, you simply add the package declaration at the top of your Java source file, like this:
* package com.example.mypackage;
* This package declaration must be the first line in your Java source file, before any imports or other code. You can also organize your packages into sub-packages by creating sub-directories in your file system. To use a class from another package, you must import that class using the import statement.
* There are several built-in packages in Java, such as java.lang, which contains fundamental classes and interfaces that are automatically imported into every Java program. Other common packages include java.util (for utility classes), java.io (for input/output operations), and java.awt (for GUI components).
* **Datastructures**
* Arrays - A fixed-size collection of elements of the same type that can be accessed using an index. Arrays are used to store and manipulate data in a linear manner.
* ArrayList - A resizable array that stores elements of the same type. ArrayLists are similar to arrays, but they can dynamically increase or decrease in size as needed.
* LinkedList - A linked list that stores elements of the same type. Each element in a linked list is connected to the next element through a pointer, which makes it easy to add or remove elements in the list.
* Stack - A last-in, first-out (LIFO) collection of elements of the same type. Stacks are used to store and manipulate data in a specific order, such as undo/redo operations.
* Queue - A first-in, first-out (FIFO) collection of elements of the same type. Queues are used to store and manipulate data in a specific order, such as processing requests in a web server.
* HashMap - A collection of key-value pairs that allows fast access to the values based on the keys. Hashmaps are used to store and retrieve data quickly based on a specific key.
* TreeMap - A sorted map that stores key-value pairs in a specific order based on the keys. TreeMaps are used to store and retrieve data in a specific order.
* HashSet - A collection of unique elements of the same type. HashSets are used to store and manipulate data that should not have any duplicates.
* TreeSet - A sorted set that stores elements in a specific order. TreeSets are used to store and retrieve data in a specific order.
* **Collections**
* In Java, the Collections framework provides a set of classes and interfaces that are used to store and manipulate collections of objects. Collections are used to group related objects and perform operations on them. The Collections framework provides several data structures that implement various interfaces to support different types of collections. Some of the commonly used interfaces in the Collections framework include.
  + List - A collection that maintains elements in a specific order, allowing duplicates.
  + Set - A collection that contains unique elements, with no specific order.
  + Map - A collection that maps keys to values, where each key is unique.
* The Collections framework also provides a number of classes that implement these interfaces. Some of the commonly used classes include:
  + ArrayList - A resizable array that implements the List interface.
  + LinkedList - A linked list that implements the List interface.
  + HashSet - A set that uses a hash table to store elements.
  + TreeSet - A set that maintains elements in sorted order.
  + HashMap - A map that uses a hash table to store key-value pairs.
  + TreeMap - A map that maintains key-value pairs in sorted order.
* In addition to these basic data structures, the Collections framework also provides a number of utility classes and methods that can be used to perform common operations on collections. Some of these classes and methods include:
  + Collections - A utility class that provides methods for sorting, shuffling, and searching collections.
  + Iterator - An interface that provides methods for iterating over collections.
  + Comparator - An interface that provides methods for comparing elements in a collection.
  + Stream - A new feature introduced in Java 8 that provides a way to perform aggregate operations on collections.
* **Generics**
* Generics is a feature of Java that allows types to be parameterized. It allows developers to define classes, interfaces, and methods that can work with different types of objects, without the need for explicit casting. The main benefit of generics is that it provides type safety at compile time. By using generics, developers can specify the type of data that a collection or class will hold, ensuring that only the correct type of data is used throughout the code. This helps to prevent errors that might occur if the wrong type of data is used at runtime.
* In Java, generics are implemented using type parameters, which are specified using angle brackets (< >). For example, a generic ArrayList class can be defined as follows:
* ArrayList<String> list = new ArrayList<String>();
* Generics can also be used with interfaces and methods. For example, the Collections.sort() method can be used to sort a collection of any type, if the type implements the Comparable interface. Here's an example:
* public static <T extends Comparable<T>> void sort(List<T> list) { // implementation } In this example, the type parameter <T extends Comparable<T>> specifies that the method can be called with any type that implements the Comparable interface.
* Generics provides a way to write reusable and type-safe code. By using generics, Java developers can create classes, interfaces, and methods that can be used with different types of objects, without sacrificing type safety or requiring explicit casting.
* **Serialization/Deserialization**
* Serialization is the process of converting an object's state to a byte stream, which can then be stored in a file or sent over a network. Deserialization is the reverse process, where the byte stream is converted back into an object's state.
* Serialization is used for a variety of purposes in Java, such as persisting an object's state between sessions, passing objects between applications over a network, or creating deep copies of objects. Java provides a built-in mechanism for serialization and deserialization through the Serializable interface. Any class that implements Serializable can be serialized and deserialized using the ObjectOutputStream and ObjectInputStream classes, respectively.
* Serialization and deserialization can also be customized by providing custom readObject() and writeObject() methods in the Serializable class. This allows developers to control the serialization and deserialization process, such as handling transient fields, enforcing invariants, or ensuring security.
* **Networking**
* Java provides a comprehensive set of APIs for network programming, enabling developers to build robust networked applications. The Java networking APIs provide a high-level abstraction for networking, while also offering low-level control over network sockets and protocols. Here are some key concepts and APIs related to networking in Java:
  + IP Addresses and Ports: In networking, devices are identified by their IP addresses, which are unique identifiers assigned to each device on a network. Ports are used to identify specific applications running on a device. In Java, the InetAddress class provides methods for working with IP addresses, while the Socket class provides methods for working with ports.
  + Sockets: A socket is an endpoint for communication between two devices over a network. In Java, the Socket class represents a client-side socket, while the ServerSocket class represents a server-side socket. The Socket class provides methods for connecting to a server, sending and receiving data, and closing the connection.
  + URLs and URIs: URLs (Uniform Resource Locators) and URIs (Uniform Resource Identifiers) are used to identify resources on the web. In Java, the URL and URI classes provide methods for working with URLs and URIs, such as parsing and constructing them.
  + HTTP and HTTPS: HTTP (Hypertext Transfer Protocol) is the protocol used for communication between web browsers and servers. HTTPS (HTTP Secure) is a secure version of HTTP that uses SSL/TLS encryption to protect data transmitted between the client and server. In Java, the HttpURLConnection and HttpsURLConnection classes provide methods for working with HTTP and HTTPS connections.
  + Datagram Sockets: Datagram sockets provide a connectionless, unreliable communication protocol, like UDP (User Datagram Protocol). In Java, the DatagramSocket and DatagramPacket classes provide methods for working with datagram sockets.
  + Multicast Sockets: Multicast sockets enable communication between multiple devices over a network, using a single IP address. In Java, the MulticastSocket class provides methods for working with multicast sockets.
  + Non-Blocking I/O: Non-blocking I/O is a technique for handling I/O operations without blocking the program's execution. In Java, the NIO (New I/O) package provides classes for non-blocking I/O, such as the Selector, SocketChannel, and ServerSocketChannel classes.
  + RMI (Remote Method Invocation): RMI is a Java-specific technology that enables communication between Java objects running on different devices over a network. In Java, the java.rmi package provides classes for working with RMI.
* Overall, the Java networking APIs provide a powerful and flexible set of tools for building networked applications, from simple client-server connections to more complex distributed systems.
* **Multithreading**
* Multithreading is programming technique that enables the execution of multiple threads simultaneously within a single program. Each thread runs independently and concurrently, allowing for better utilization of CPU resources and faster program execution. Java provides built-in support for multithreading through its Thread class and related interfaces.
* Creating a new thread in Java involves creating an instance of the Thread class and overriding its run() method. The run() method contains the code that will be executed by the thread when it is started. Here are some key concepts related to multithreading in Java:
  + Thread Synchronization: This is a mechanism that ensures that two or more threads do not execute a particular block of code simultaneously. Java provides synchronized keywords and related constructs to implement thread synchronization.
  + Thread Priorities: Threads can be assigned different priorities based on their importance. Java provides constants to specify thread priorities.
  + Thread Lifecycle: A thread in Java goes through various stages in its lifetime, such as the new, runnable, blocked, waiting, timed waiting, and terminated stages.
  + Thread Pooling: This is a technique that involves creating a group of threads that can be reused to perform multiple tasks. Java provides the Executor framework for thread pooling.
* Multithreading is commonly used in Java applications for tasks such as network communication, GUI development, and background processing. It is also widely used in server-side applications to handle multiple requests simultaneously. However, multithreading can introduce its own set of issues, such as race conditions and deadlocks.