Introduction to Java

History:

1991: Java was developed by James Gosling, Mike Sheridan, and Patrick Naughton at Sun Microsystems.

Initially called "Oak," it was intended for interactive television but was too advanced.

1995: Oak was renamed Java, and the language was restructured for the burgeoning World Wide Web.

1996: The first official version, JDK 1.0, was released.

2009: Oracle Corporation acquired Sun Microsystems, taking ownership of Java.

Goals:

Platform Independence: Write Once, Run Anywhere (WORA). Java applications can run on any device with a JVM.

Simplicity and Familiarity: Java's syntax is similar to C++, making it easier for developers to learn.

Security: Java has built-in security features to create tamper-free and virus-free systems.

High Performance: With Just-In-Time compilers, Java applications can achieve high performance.

Multithreading: Java supports multithreading, enabling developers to write programs that can perform multiple tasks simultaneously.

Robustness: Java emphasizes early checking for possible errors, runtime checking, and eliminating error-prone situations.

Fundamentals of OOPs

Object-Oriented Programming (OOP):

A programming paradigm based on the concept of objects which contain data in the form of fields (attributes) and code in the form of procedures (methods).

Core Concepts of OOP:

1. **Class:** A blueprint for objects. Defines a datatype by bundling data and methods that work on the data.

2. **Object:** An instance of a class. It is a basic unit of Object-Oriented Programming.

3. **Inheritance:** A mechanism where a new class inherits properties and behaviour from an existing class.

4. **Polymorphism:** The ability of different objects to respond to the same method call in different ways.

5. **Encapsulation:** The bundling of data with the methods that operate on the data, restricting direct access to some of the object's components.

6. **Abstraction:** The concept of hiding the complex implementation details and showing only the necessary features of an object.

Overview of JDK, JVM, and Garbage Collection

JDK (Java Development Kit):

A software development kit required to develop Java applications.

It includes:

JRE (Java Runtime Environment):

Provides libraries, Java Virtual Machine (JVM), and other components to run applications.

Development Tools:

Tools like compilers (`javac`), debuggers, and other utilities.

JVM (Java Virtual Machine):

An abstract machine that enables your computer to run a Java program.

It converts Java bytecode into machine code.

Key features include:

Platform Independence:

JVM allows the same Java program to run on different operating systems.

Memory Management: Manages memory areas like Heap and Stack.

Security: Verifies the bytecode before execution.

Garbage Collection:

An automatic memory management feature in Java.

It identifies and discards objects that are no longer needed to free up memory resources.

Automatic Process: Developers do not need to manually deallocate memory.

Mark and Sweep Algorithm:

Commonly used garbage collection technique where the "mark" phase identifies unused objects, and the "sweep" phase removes them.

Java Basics

Identifiers:

Names given to elements in a program such as variables, classes, and methods.

Rules for identifiers:

- Must start with a letter, dollar sign (`$`), or underscore (`_`).
- Subsequent characters can be letters, digits, dollar signs, or underscores.
- Case-sensitive and cannot be a keyword (`class`, `int`, etc.).

Keywords:

Reserved words with predefined meanings.

Examples: include `class`, `public`, `static`, `void`, `if`, `else`, `while`, `for`, `switch`, `break`, etc.

Data Types:

- **Primitive Data Types:**
  - **Boolean:** (true or false)
  - **Character: `char`**

- **Numeric:**
  - Integer: `byte`, `short`, `int`, `long`
  - Floating-point: `float`, `double`

- **Non-Primitive Data Types:** Classes, Interfaces, Arrays

Operators:

Symbols that perform operations on variables and values.

Types include:

Arithmetic Operators: `+`, `-`, `*`, `/`, `%`

Relational Operators: `==`, `!=`, `>`, `<`, `>=`, `<=`

Logical Operators: `&&`, `||`, `!`

Assignment Operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`

Control Statements

They control the flow of execution in a program, enabling the program to make decisions, perform repeated actions, and execute code conditionally.

Decision Constructs

1. **if Statement:**
   - Used to execute a block of code only if a specified condition is true.
2. **if-else Statement:**
   - Provides an alternative block of code to execute if the condition is false.
3. **if-else-if Ladder:**
   - Used to test multiple conditions sequentially.
4. **switch Statement:**
   - Allows the selection of one of many code blocks to execute.

Using Loop Constructs

1. **for Loop:** Repeatedly executes a block of code a specified number of times.

2. **while Loop:**

Repeatedly executes a block of code as long as a specified condition is true.

3. **do-while Loop:**

Executes a block of code once, and then repeatedly executes it as long as a specified condition is true.

4. **Enhanced for Loop (for-each Loop):**

Iterates through elements in a collection or array.

```
  for (type element : array) {
      // code to be executed
  }
```

Command Line Arguments

They are arguments passed to a Java program during its execution from the command line interface.

They are stored in the `String[] args` array in the `main` method.

Useful for configuring the behavior of a program without changing the code.

Summary

Decision Constructs: Allow execution of code based on conditions (`if`, `if-else`, `if-else-if`, `switch`).

Loop Constructs:

Allow repeated execution of code (`for`, `while`, `do-while`, enhanced for loop).

Command Line Arguments:

Enable the passing of arguments to a Java program during execution, accessible via the `String[] args` array in the `main` method.

Arrays in Java

An array is a data structure that stores a fixed-size sequential collection of elements of the same type.

In Java, arrays are objects, and they can be one-dimensional, two-dimensional, or multidimensional.

Creating and Using Arrays

1D Arrays

Declaration: Declaring an array specifies its type.

```
int[] array; // preferred          int array[]; // also valid
```

Instantiation: Allocating memory for the array.

```
array = new int[5]; // creates an array of 5 integers
```

Initialization: Assigning values to the array elements.

```
array[0] = 1;   array[1] = 2;   array[2] = 3;   array[3] = 4;   array[4] = 5;
// or directly during declaration
int[] array = {1, 2, 3, 4, 5};
```

Accessing Elements: Using the index to access elements.

```
int value = array[2]; // value is 3
```

Iterating Over an Array:

```
for (int i = 0; i < array.length; i++) {
    Sout(array[i]);
}
// Using enhanced for loop
for (int element : array) {
    Sout(element);
}
```

2D Arrays

Declaration: int[][] matrix;

Instantiation: matrix = new int[3][3]; // creates a 3x3 matrix

Initialization:

```
matrix[0][0] = 1;    matrix[0][1] = 2;      matrix[0][2] = 3;
// or directly during declaration
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

- **Accessing Elements:** int value = matrix[1][2]; // value is 6
- **Iterating Over a 2D Array:**

```
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        Sout(matrix[i][j] + " ");
    }
    Sout();
}
// Using enhanced for loop
for (int[] row : matrix) {
    for (int element : row) {
        Sout(element + " ");
    }
    Sout();
}
```

Multidimensional Arrays

Java supports arrays with more than two dimensions, although they are less commonly used.

Declaration: int[][][] multiArray;

Instantiation: multiArray = new int[3][3][3]; // creates a 3x3x3 array

Initialization and Access:

```
Multi_Array[0][0][0] = 1;
int value = multi_Array[0][0][0]; // value is 1
```

- Iterating Over Multidimensional Arrays:

```
for (int i = 0; i < multiArray.length; i++) {
    for (int j = 0; j < multiArray[i].length; j++) {
        for (int k = 0; k < multiArray[i][j].length; k++) {
            Sout(multiArray[i][j][k] + " ");
        }
        Sout( );
    }
    Sout( );
}
```

Jagged Arrays

It is also known as an array of arrays, is an array whose elements are arrays of potentially different sizes.

Declaration: int[][] jaggedArray;

Instantiation:

```
jaggedArray = new int[3][]; // create an array with 3 rows
jaggedArray[0] = new int[2]; // row 0 has 2 columns
jaggedArray[1] = new int[3]; // row 1 has 3 columns
jaggedArray[2] = new int[1]; // row 2 has 1 column
```

Initialization:

```
jaggedArray[0][0] = 1;     jaggedArray[0][1] = 2;      jaggedArray[1][0] = 3;
```

```
 jaggedArray[1][1] = 4;     jaggedArray[1][2] = 5;       jaggedArray[2][0] = 6;
```

Accessing Elements: int value = jaggedArray[1][2]; // value is 5

Iterating Over a Jagged Array:

```
 for (int i = 0; i < jaggedArray.length; i++) {
     for (int j = 0; j < jaggedArray[i].length; j++) {
         Sout(jaggedArray[i][j] + " ");
     }
     Sout();
 }
 // Using enhanced for loop
 for (int[] row : jaggedArray) {
     for (int element : row) {
         Sout(element + " ");
     }
     Sout();
 }
```

Summary

1D Arrays: Simple arrays storing elements in a linear fashion.

2D Arrays: Arrays storing elements in a tabular format (rows and columns).

Multidimensional Arrays: Arrays with more than two dimensions.

Jagged Arrays: Arrays of arrays where sub-arrays can have different sizes.

Classes & Objects

Java is an object-oriented programming (OOP) language, where the basic building blocks are classes and objects.

Defining a Class:

A class is a blueprint for creating objects.

It encapsulates data for the object and methods to manipulate that data.

Syntax:

```
class ClassName {
   // fields (variables)
   // methods (functions)
}
```

Example:

```
public class Person {
   // Fields (attributes)
   private String name;
   private int age;
   // Constructor
   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public void displayDetails() {        // Methods
      Sout("Name: " + name + ", Age: " + age);
   }
   public String getName() {    // Getter for name
      return name;
   }
   public void setName(String name) {    // Setter for name
      this.name = name;
   }
   public int getAge() {    // Getter for age
      return age;
```

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

Creating Objects

An object is an instance of a class. It is created using the `new` keyword.

Example:

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Person class
        Person person1 = new Person("Alice", 30);
        person1.displayDetails(); // Output: Name: Alice, Age: 30
        // Accessing fields and methods
        person1.setAge(31);
        Sout("Updated Age: " + person1.getAge()); // Output: Updated Age: 31
    }
}
```

Access Control

Access control specifies the visibility of fields, methods, and classes. Java provides four access modifiers:

1. **private:** Accessible only within the same class.
2. **default (no modifier):** Accessible only within the same package.
3. **protected:** Accessible within the same package and by subclasses.
4. **public:** Accessible from any other class.

Example:

```
public class AccessControl{
    private int privateVar; // Accessible only within this class
    int defaultVar;         // Accessible within the same package
    protected int protectedVar; // Accessible within the same package and
subclasses
    public int publicVar;   // Accessible from anywhere
}
```

Method Overloading

It allows a class to have more than one method with the same name, as long as their parameter lists are different (different type, number, or both).

Example:

```
public class Overloading {
    public void display(int a) {    // Method with one parameter
        Sout("Argument: " + a);
    }
    public void display(int a, int b) {    // Overloaded method with two params
        Sout("Arguments: " + a + " and " + b);
    }
    // Overloaded method with different parameter type
    public void display(String message) {
        Sout("Message: " + message);
    }
    public static void main(String[] args) {
        OverloadingExample obj = new OverloadingExample();
        obj.display(5);              // Output: Argument: 5
        obj.display(5, 10);          // Output: Arguments: 5 and 10
        obj.display("Hello, World!"); // Output: Message: Hello, World!
    }
}
```

Summary

Class: A blueprint for creating objects. It contains fields and methods.

Object: An instance of a class.

Access Control: Mechanism to restrict access to the members of a class. Includes `private`, `default`, `protected`, and `public`.

Method Overloading: The ability to define multiple methods with the same name but different parameter lists within the same class.

Constructors

A constructor is a special method used to initialize objects.

It is called when an instance of a class is created.

Constructors have the same name as the class and do not have a return type.

Types:

1. **Default Constructor:**

A no-argument constructor provided by Java if no other constructors are defined.

```
public class Example {
    public Example() {
        // Default constructor
    }
}
```

2. **Parameterized Constructor:**

- A constructor that accepts parameters to initialize object attributes.

```
public class Example {
    private int value;
    public Example(int value) {
        this.value = value;
```

```
      }
   }
```

Constructor Overloading

It allows a class to have more than one constructor with different parameter lists.

This provides flexibility in object creation.

Example:

```
public class Person {
    private String name;
    private int age;
    // Default constructor
    public Person ( ) {
        this.name = "Unknown";
        this.age = 0;
    }
    // Parameterized constructor
    public Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Another parameterized constructor
    public Person(String name) {
        this.name = name;
        this.age = 0;
    }
}
```

Use of `this` Keyword

- Referring to Instance Variables:

`this` is used to differentiate between instance variables and parameters with the same name.

```
 public class Example {
     private int value;
     public Example(int value) {
         this.value = value; // 'this.value' refers to the instance variable
     }
 }
```

- Calling Another Constructor:

`this` can be used to call another constructor in the same class.

```
 public class Example {
     private int value;
     public Example( ) {
         this(10); // Calls the parameterized constructor
     }
     public Example(int value) {
         this.value = value;
     }
 }
```

Static Keyword

Static Variables:

Belong to the class rather than any instance. Shared among all instances of the class.

```
 public class {
     public static int count = 0;
     public Example( ) {
         count++;
```

```
    }
}
```

Static Methods:

Can be called without creating an instance of the class. They can only access static data.

```
public class Example {
    public static void displayCount() {
        System.out.println("Count: " + count);
    }
}
```

Static Blocks:

Used for static initialization of a class. Runs when the class is loaded.

```
public class Example {
    static {
        // Static initialization block
        System.out.println("Static block executed.");
    }
}
```

Inheritance

It is a mechanism where one class (subclass/derived class) inherits the fields and methods of another class (superclass/base class).

Basics:

Superclass: The class being inherited from.

Subclass: The class that inherits from the superclass.

Example:

```
public class Animal {
    public void eat() {
        Sout("This animal eats food.");
    }
}
public class Dog extends Animal {
    public void bark() {
        Sout("The dog barks.");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat(); // Inherited method
        dog.bark(); // Dog's own method
    }
}
```

Types of Inheritance

1.**Single Inheritance**: A class inherits from one superclass.

2. **Multilevel Inheritance**: A class inherits from a subclass, creating a chain.

3. **Hierarchical Inheritance:** Multiple subclasses inherit from one superclass.

4. **Multiple Inheritance** (Not supported directly in Java): A class inherits from multiple superclasses (Java supports this through interfaces).

Using `super` Keyword

Accessing Superclass Methods: `super` can be used to call methods of the superclass.

```
  public class Animal {
    public void eat() {
        Sout("This animal eats food.");
    }
}
public class Dog extends Animal {
    public void eat() {
        super.eat(); // Calls superclass method
        Sout("The dog eats bones.");
    }
}
```

Calling Superclass Constructor: `super` can be used to call the superclass constructor.

```
public class Animal {
    public Animal(String name) {
        Sout("Animal: " + name);
    }
}
public class Dog extends Animal {
    public Dog(String name) {
        super(name); // Calls superclass constructor
        Sout("Dog: " + name);
    }
}
```

Method Overriding

Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass.

The method in the subclass must have the same signature as in the superclass.

Example:

```
public class Animal {
    public void makeSound() {
        Sout("Animal makes a sound.");
    }
}
public class Dog extends Animal {
    @Override
    public void makeSound() {
        Sout("Dog barks.");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Dog();
        myAnimal.makeSound(); // Output: Dog barks.
    }
}
```

Summary

Constructors: Special methods for initializing objects.

Can be overloaded to provide multiple ways to initialize objects.

this Keyword: Refers to the current instance of the class.

Used to access instance variables and methods, and to call other constructors.

static Keyword: Defines class-level fields and methods.

Static members are shared among all instances of the class.

Inheritance:

A mechanism to create a new class using properties and behaviours of an existing class.

Supports code reuse and hierarchical classification.

super Keyword:

Refers to the superclass and is used to call superclass methods and constructors.

Method Overriding:

Allows a subclass to provide a specific implementation of a method already defined in its superclass.

Dynamic Method Dispatch

DMD also known as runtime polymorphism or method overriding, is a mechanism by which a call to an overridden method is resolved at runtime rather than compile-time.

It is a fundamental aspect of Java's runtime behaviour.

Example:

```
class Animal {
   void sound() {
      Sout("Animal makes a sound");
   }
}
class Dog extends Animal {
   @Override
   void sound() {
      Sout("Dog barks");
   }
}
class Cat extends Animal {
    @Override
    void sound() {
        Sout("Cat meows");
    }
}
public class Test {
    public static void main(String[] args) {
        Animal a;
        a = new Dog();
        a.sound( );  // Outputs: Dog barks
        a = new Cat();
        a.sound( );  // Outputs: Cat meows
    }
}
```

Final Keyword

This keyword is used to restrict the usage of variables, methods, and classes.

1. **Final Variables:** The value of a final variable cannot be changed once it is initialized.

```
   final int MAX_VALUE = 100;
```

2. **Final Methods:** A final method cannot be overridden by subclasses.

```
   class Base {
       final void display() {
           Sout("This is a final method.");
       }
   }
```

3. **Final Classes:** A final class cannot be subclassed.

```
final class FinalClass {
    // class body
}
```

Abstract Methods & Classes

Abstract methods and classes provide a way to achieve abstraction in Java.

An abstract class cannot be instantiated and can contain abstract methods, which are methods without a body.

Abstract Class

Declaration:

```
abstract class Animal {
    abstract void sound(); // Abstract method
    void sleep() {
        Sout("This animal sleeps.");
    }
}
```

Implementation in Subclass:

```
class Dog extends Animal {
    @Override
    void sound() {
        Sout("Dog barks");
    }
}
public class Test {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.sound(); // Outputs: Dog barks
        a.sleep(); // Outputs: This animal sleeps.
    }
  }
```

Packages

Packages are used to group related classes, interfaces, and sub-packages.

They help in avoiding name conflicts and managing code in large applications.

Built-In Packages

Java provides many built-in packages such as `java.lang`, `java.util`, `java.io`, etc.

Example:

```
import java.util.ArrayList;
public class Test {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Hello");
        list.add("World");
        Sout(list);
    }
}
```

User-Defined Packages

To create a package, use the `package` keyword.

Example:

1. **Create a package:**

```
   package mypackage;
   public class MyClass {
       public void display() {
           Sout("This is my package!");
```

```
    }
  }
```

2. **Use the package:**

```
  import mypackage.MyClass;
  public class Test {
      public static void main(String[] args) {
          MyClass obj = new MyClass();
          obj.display(); // Outputs: This is my package!
      }
  }
```

Interfaces

Interfaces in Java are abstract types that allow the specification of methods that must be implemented by a class.

Interfaces support multiple inheritance since a class can implement multiple interfaces.

Declaration:

```
 interface Animal {
     void sound( );
     void sleep( );
 }
```

Implementation:

```
 class Dog implements Animal {
     @Override
     public void sound( ) {
         Sout("Dog barks");
     }
```

```
    @Override
    public void sleep() {
        Sout("Dog sleeps");
    }
 }
 public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound(); // Outputs: Dog barks
        dog.sleep(); // Outputs: Dog sleeps
    }
 }
```

Extending Interfaces

An interface can extend another interface, inheriting its methods.

Example:

```
interface Animal {
   void sound( );
}
interface Pet extends Animal {
   void play( );
}
class Dog implements Pet {
   @Override
   public void sound( ) {
      Sout("Dog barks");
   }
   @Override
```

```
    public void play() {
        Sout("Dog plays");
    }
}
public class Test {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound(); // Outputs: Dog barks
        dog.play(); // Outputs: Dog plays
    }
}
```

Summary

Dynamic Method Dispatch: Resolves calls to overridden methods at runtime.

Final Keyword: Restricts usage for variables (can't be changed), methods (can't be overridden), and classes (can't be inherited).

Abstract Methods & Classes: Provide a way to achieve abstraction; abstract classes cannot be instantiated and may contain abstract methods.

Packages: Used to group related classes and interfaces, avoid name conflicts, and manage code efficiently.

Interfaces: Abstract types that define methods to be implemented by classes, supporting multiple inheritance and enabling polymorphism.

Strings in Java

Java provides several classes to work with strings:

`String`, `StringBuffer`, `StringBuilder`, and `StringTokenizer`.

Each serves different purposes and has its own characteristics.

Intro to Strings (str)

Immutable String

In Java, `String` objects are immutable, meaning that once a `String` object is created, it cannot be changed.

Example:

```
String s = "Hello";
s = s.concat(" World"); // The original str is not changed; a new str is created.
Sout(s); // Outputs: Hello World
```

Methods of String Class

The `String` class provides many methods to perform operations on strings.

1. **Length:** returns length of the string.

```
String str = "Hello";
int len = str.length(); // Returns 5
```

2. **CharAt:** returns character present at that position/index.

```
char ch = str.charAt(1); // Returns 'e'
```

3. **Substring:** in order to get the ele's in given range

```
String substr = str.substring(1, 4); // Returns "ell"
```

4. Equals: returns true or false if input matches or not.

```
boolean isEqual = str.equals("Hello"); // Returns true
```

5. **CompareTo:** compare both string character by character

```
int result = str.compareTo("Hello"); // Returns 0 (since both strings are equal)
```

6. **IndexOf:** in order to get the index of input ele.

```
int index = str.indexOf('e'); // Returns 1
```

7. **Replace:** to update the existing character.

```
String newStr = str.replace('e', 'a'); // Returns "Hallo"
```

8. **ToUpperCase:** convert character into Upper Case alphabets.

```
String upperStr = str.toUpperCase(); // Returns "HELLO"
```

9. **ToLowerCase:** convert character into Lower Case alphabets.

```
String lowerStr = str.toLowerCase(); // Returns "hello"
```

10. **Trim:** remove extra spaces from start and end.

```
String trimmedStr = str.trim(); // Removes leading and trailing whitespaces
```

String Buffer and String Builder

`StringBuffer` and `StringBuilder` are mutable sequences of characters.
Both classes are used to create strings that can be modified after creation.
The main difference is that `StringBuffer` is synchronized (thread-safe) whereas `StringBuilder` is not, making `StringBuilder` faster but not thread-safe.

String Buffer Class

Creation: `StringBuffer sb = new StringBuffer("Hello");`

Methods:

1. **Append:** adds at the end by default.

```
sb.append(" World"); // Appends " World" to the existing string
```

2. **Insert:** Add element at given / particular index.

```
sb.insert(5, " Java"); // Inserts " Java" at index 5
```

3. **Replace:** modification of existing ele`s in given

   sb.replace(6, 10, "C++"); // Replaces "Java" with "C++"

4. **Delete:** remove ele in given range

   sb.delete(5, 9); // Deletes the substring from index 5 to 9

5. **Reverse:** changes the order

   sb.reverse( ); // Reverses the string

6. **Capacity:** elements it can hold.

   int capacity = sb.capacity(); // Returns the current capacity

7. **Ensure Capacity:** minimum ele's it can hold.

   sb.ensureCapacity(50); // Ensures that the capacity is at least 50

StringBuilder Class

Creation: StringBuilder sb = new StringBuilder("Hello");

Methods: Similar to `StringBuffer`.

```
sb.append(" World");          sb.insert(5, " Java");
sb.replace(6, 10, "C++");     sb.delete(5, 9);          sb.reverse();
```

toString Method

It is used to return the string representation of an object.

In `StringBuffer` and `StringBuilder`, it converts the object to a `String`.

Example:

StringBuffer sb = new StringBuffer("Hello");

String str = sb.toString(); // Converts StringBuffer to String

StringTokenizer Class

It is a legacy class used to split strings into tokens.

It is found in the `java.util` package.

Creation:

```
StringTokenizer st = new StringTokenizer("Hello World Java", " ");
```

Methods:

1. **HasMoreTokens:**

   ```
   while (st.hasMoreTokens()) {
       System.out.println(st.nextToken());
   }
   ```

2. **NextToken:**

   ```
   String token = st.nextToken(); // Returns the next token
   ```

3. **CountTokens:**

   ```
   int count = st.countTokens(); // Returns the number of remaining tokens
   ```

Summary

Strings: Immutable sequences of characters with many built-in methods for manipulation.

StringBuffer: Mutable, synchronized (thread-safe) sequences of characters.

StringBuilder: Mutable, non-synchronized (not thread-safe) sequences of characters.

toString Method: Converts objects to their string representation.

A legacy class for tokenizing strings into smaller parts.

Exception Handling (EH)

Exception handling is a mechanism in Java to handle runtime errors, ensuring the normal flow of the application is maintained.

It helps in managing errors by using a combination of `try`, `catch`, `throw`, `throws`, and `finally` blocks.

EH Fundamentals:

Exceptions are unexpected events that disrupt the normal flow of a program.

They can be categorized into checked exceptions, unchecked exceptions, and errors.

Exception Types

1. **Checked Exceptions:**

   - Checked at compile-time.
   - Must be declared using `throws` or handled using `try-catch`.
   - Examples: `IOException`, `SQLException`.

2. Unchecked Exceptions:

   - Checked at runtime.
   - Also known as runtime exceptions.
   - Examples: `NullPointerException`, `ArrayIndexOutOfBoundsException`.

3. Errors:

   - Not typically handled in Java.
   - Represent serious problems.
   - Examples: `OutOfMemoryError`, `StackOverflowError`.

Try and Catch

The `try` block contains code that might throw an exception.

The `catch` block catches and handles the exception.

Syntax:

```
try {
   // Code that may throw an exception
} catch (ExceptionType e) {
   // Code to handle the exception
}
```

Example:

```
public class Exception {
   public static void main(String[] args) {
      try {
         int data = 100 / 0; // This will throw an ArithmeticException
      } catch (ArithmeticException e) {
         System.out.println("Exception caught: " + e);
      }
   }
}
```

Multiple Catch Clauses

A `try` block can be followed by multiple `catch` blocks to handle different types of exceptions.

Syntax:

```
try {
   // Code that may throw exceptions
} catch (ExceptionType1 e1) {
   // Handle ExceptionType1
} catch (ExceptionType2 e2) {
   // Handle ExceptionType2
}
```

Example:

}

```
public class MultipleCatch {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];
            arr[5] = 30 / 0;
        } catch (ArithmeticException e) {
            Sout("ArithmeticException caught: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            Sout("ArrayIndexOutOfBoundsException caught: " + e);
        }
    }
}
```

Nested Try

A `try` block can be nested within another `try` block to handle multiple exceptions at different levels.

Syntax:

```
try {
    // Outer try block
    try {
        // Inner try block
    } catch (ExceptionType e) {
        // Handle exception for inner try block
    }
} catch (ExceptionType e) {
    // Handle exception for outer try block
}
```

Example:

```
public class NestedTry {
    public static void main(String[] args) {
        try {
            try {
                int data = 100 / 0; // Inner try block
            } catch (ArithmeticException e) {
                Sout("Inner try block: " + e);
            }
        } catch (Exception e) {
            Sout("Outer try block: " + e);
        }
    }
}
```

Throw, Throws, and Finally

Throw

The `throw` keyword is used to explicitly throw an exception.

Syntax:

```
throw new ExceptionType("Error message");
```

Example:

```
public class Throw {
    static void checkAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Access denied - You must be at least
18 years old.");
        } else {
            Sout("Access granted - You are old enough!");
        }
    }
    public static void main(String[] args) {
        checkAge(15);
    }
}
```

Throws

The `throws` keyword is used in method signatures to declare that a method might throw an exception.

Syntax:

```
returnType methodName() throws ExceptionType {
    // Method body
}
```

Example:

```
public class Throws {
    static void checkAge(int age) throws ArithmeticException {
        if (age < 18) {
            throw new ArithmeticException("Access denied - You must be at least 
18 years old.");
        } else {
            Sout("Access granted - You are old enough!");
        }
    }
    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (ArithmeticException e) {
            Sout("Exception caught: " + e);
        }
    }
}
```

Finally

The `finally` block contains code that will be executed regardless of whether an exception is thrown or not. It is typically used for cleanup activities.

Syntax:

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code to be executed after try-catch, regardless of the outcome
}
```

Example:

```
public class Finally {
    public static void main(String[] args) {
        try {
            int data = 100 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

Creating Custom Exceptions

You can create your own exceptions by extending the `Exception` class.

Example:

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}
public class CustomException {
    static void validate(int age) throws MyException {
        if (age < 18) {
            throw new MyException("Access denied - You must be at least 18 years
old.");
        } else {
            Sout("Access granted - You are old enough!");
        }
    }
    public static void main(String[] args) {
        try {
            validate(15);
        } catch (MyException e) {
            Sout("Exception caught: " + e);
        }
    }
}
```

Summary

Exception Types: Checked exceptions, unchecked exceptions, and errors.

Try and Catch: Handle exceptions using try-catch blocks.

Multiple Catch Clauses: Handle different exceptions with multiple catch blocks.

Nested Try: Use nested try blocks for different levels of exception handling.

Throw: Explicitly throw an exception.

Throws: Declare that a method might throw an exception.

Finally: Execute code regardless of whether an exception occurs.

Custom Exceptions: Create user-defined exceptions by extending the `Exception` class.

Multithreading in Java

Multithreading in Java allows concurrent execution of two or more threads, enabling maximum utilization of CPU.

Threads are lightweight processes that share the same memory space.

Java Thread Model (JTM):

JTM is based on shared memory, meaning multiple threads operate in the same memory space and share resources.

This can lead to issues if not managed properly, requiring synchronization.

Main Thread

The main thread is the initial thread that is created when a Java program starts.

It is the thread from which other threads are spawned.

Example:

```
public class MainThread {
    public static void main(String[] args) {
        Sout("This is the main thread.");
    }
}
```

Creating a Thread

Threads can be created in two ways: by implementing the `Runnable` interface or by extending the `Thread` class.

Implementing Runnable

1. **Implementing Runnable Interface:**

```
   class MyRunnable implements Runnable {
       public void run() {
           for (int i = 0; i < 5; i++) {
               Sout(Thread.currentThread().getName() + " - " + i);
           }
       }
   }
   public class Runnable {
       public static void main(String[] args) {
           Thread t1 = new Thread(new MyRunnable(), "Thread-1");
           t1.start();
           Thread t2 = new Thread(new MyRunnable(), "Thread-2");
           t2.start();
       }
   }
```

Extending Thread Class

2. Extending Thread Class:

```
class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            Sout(Thread.currentThread().getName() + " - " + i);
        }
    }
}
public class Thread {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.setName("Thread-1");
        t1.start();
```

```
          MyThread t2 = new MyThread();
          t2.setName("Thread-2");
          t2.start();
      }
  }
```

Creating Multiple Threads

Creating multiple threads involves starting several threads to run concurrently.

Example:

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            Sout(Thread.currentThread().getName() + " - " + i);
        }
    }
}
public class MultipleThreads {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            Thread t = new Thread(new MyRunnable(), "Thread-" + i);
            t.start();
        }
    }
}
```

Using isAlive( ) and join( )

- **isAlive( ):** Checks if a thread is still running.

```
  Thread t = new Thread(new MyRunnable());
  t.start();
  if (t.isAlive()) {
      Sout("Thread is alive.");
  }
```

- **join( ):** Waits for a thread to finish its execution.

```
  Thread t = new Thread(new MyRunnable());
  t.start( );
  try {
      t.join( );
      Sout("Thread has finished execution.");
  } catch (InterruptedException e) {
      e.printStackTrace();
  }
```

Thread Priorities

Threads can be assigned priorities that affect the order in which they are scheduled for execution.Java thread priorities range from 1 (MIN_PRIORITY) to 10 (MAX_PRIORITY), with the default being 5 (NORM_PRIORITY).

Example:

```
class MyRunnable implements Runnable {
    public void run( ) {
        for (int i = 0; i < 5; i++) {
            Sout(Thread.currentThread().getName() + " - Priority: " +
Thread.currentThread().getPriority());
        }
    }
}
public class ThreadPriority {
    public static void main(String[ ] args) {
        Thread t1 = new Thread(new MyRunnable(), "Thread-1");
        Thread t2 = new Thread(new MyRunnable(), "Thread-2");
        Thread t3 = new Thread(new MyRunnable(), "Thread-3");
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.NORM_PRIORITY);
        t3.setPriority(Thread.MAX_PRIORITY);
        t1.start( );
        t2.start( );
        t3.start( );
    }
}
```

Synchronization

Synchronization in Java is used to control the access of multiple threads to shared resources.

Without synchronization, it is possible for one thread to modify a shared resource,

while another thread is in the process of using or updating the same resource, leading to inconsistent results.

Synchronized Methods

A synchronized method ensures that only one thread can execute it at a time.

Example:

```
class Counter {
    private int count = 0;
    public synchronized void increment( ) {
        count++;
    }
    public int getCount( ) {
        return count;
    }
}
class CounterThread extends Thread {
    private Counter counter;
    public CounterThread(Counter counter) {
        this.counter = counter;
    }
    public void run( ) {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
public class SynchronizedMethod {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new CounterThread(counter);
        Thread t2 = new CounterThread(counter);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Sout("Final count: " + counter.getCount());
    }
}
```

Synchronized Block

A synchronized block is a block of code that can be synchronized on a specific object.

Example:

```
class Counter {
    private int count = 0;
    public void increment( ) {
        synchronized(this) {
            count++;
        }
    }
    public int getCount( ) {
        return count;
    }
}
class CounterThread extends Thread {
    private Counter counter;
    public CounterThread(Counter counter) {
        this.counter = counter;
    }
    public void run( ) {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
public class SynchronizedBlock{
    public static void main(String[] args) {
        Counter counter = new Counter();
        Thread t1 = new CounterThread(counter);
        Thread t2 = new CounterThread(counter);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Sout("Final count: " + counter.getCount());
    }
}
```

Summary

- **Java Thread Model:** Supports concurrent execution of threads sharing the same memory space.

- **Main Thread:** The initial thread from which other threads are spawned.

- **Creating Threads:** Done by implementing `Runnable` or extending `Thread`.

- **Multiple Threads:** Running multiple threads concurrently.

- **isAlive( ) and join( ):** Methods to check thread status and wait for thread completion.

- **Thread Priorities:** Affect thread scheduling, with priorities ranging from 1 to 10.

- **Synchronization:** Controls access to shared resources, ensuring consistent results through synchronized methods and blocks.

Generics in Java

Generics allow you to create classes, interfaces, and methods that operate on types specified by the client code.

Generics provide type safety and eliminate the need for type casting.

Intro to Generics:

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods.

The key benefit of generics is to enable code reusability and to ensure type safety at compile-time.

Example:

```
import java.util.ArrayList;

public class Generic {

   public static void main(String[] args) {

      // Without Generics

      ArrayList list = new ArrayList();

      list.add("Hello");

      list.add(123); // No error at compile time

      String str = (String) list.get(0); // Type casting required

      // String num = (String) list.get(1); // Causes ClassCastException at
runtime

      // With Generics

      ArrayList<String> genericList = new ArrayList<>();

      genericList.add("Hello");

      // genericList.add(123); // Compile-time error

      String genericStr = genericList.get(0); // No type casting required

   }
}
```

Generic Class

A generic class can work with any data type specified at the time of instantiation.

Syntax:

```
class GenericClass<T> {
    private T data;
    public GenericClass(T data) {
        this.data = data;
    }
    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
}
```

Example:

```
public class GenericClass {
    public static void main(String[] args) {
        GenericClass<Integer> intObj = new GenericClass<>(10);
        Sout("Generic Class Integer Value: " + intObj.getData());
        GenericClass<String> stringObj = new GenericClass<>("Hello");
        Sout("Generic Class String Value: " + stringObj.getData());
    }
}
```

Generic Method

Generic methods allow a method to be parameterized with types.

Syntax:

```
public <T> void genericMethod(T data) {
    System.out.println("Generic Method Data: " + data);
}
```

Example:

```
public class GenericMethod {
    public <T> void genericMethod(T data) {
        Sout("Generic Method Data: " + data);
    }
    public static void main(String[] args) {
        GenericMethod example = new GenericMethod();
        example.genericMethod(10);
        example.genericMethod("Hello");
    }
```

Generic Constructor

A constructor can be declared as generic even if its class is not generic.

Syntax:

```
class GenericConstructor {
    private double value;
    public <T extends Number> GenericConstructor(T arg) {
        value = arg.doubleValue();
    }
    public void showValue() {
        Sout("Value: " + value);
    }
}
```

Example:

```
public class GenericConstructor {
    public static void main(String[] args) {
        GenericConstructor obj1 = new GenericConstructor(100);
        obj1.showValue();
        GenericConstructor obj2 = new GenericConstructor(123.45);
        obj2.showValue();
    }
}
```

Generic Interfaces

Interfaces can be declared generic to be implemented by generic or non-generic classes.

Syntax:

```
interface GenericInterface<T> {
    void method(T data);
}
```

Example:

```
class GenericClass implements GenericInterface<Integer> {
    public void method(Integer data) {
        Sout("Generic Interface Method Data: " + data);
    }
}
public class GenericInterface {
    public static void main(String[] args) {
        GenericClass obj = new GenericClass();
        obj.method(100);
    }
}
```

Collections Framework

The Java Collections Framework (JCF) provides a set of classes and interfaces that implement commonly reusable collection data structures.

Intro to Collections Framework

The JCF provides a standard way to handle groups of objects.

It includes classes and interfaces for lists, sets, queues, and maps.

It provides algorithms for searching, sorting, and manipulating these collections.

Collection Interfaces

The main collection interfaces are `Collection`, `List`, `Set`, and `Queue`.

List Interface

A `List` is an ordered collection that can contain duplicate elements.

It provides positional access and insertion of elements.

Common Implementations: `ArrayList` **and** `LinkedList`

Example:

```
import java.util.ArrayList;

import java.util.List;

public class List {

    public static void main(String[] args) {

        List<String> list = new ArrayList<>();

        list.add("A");

        list.add("B");

        list.add("C");

        for (String str : list) {

            Sout(str);

        }

    }

}
```

Queue Interface

A `Queue` is a collection designed for holding elements prior to processing.

It typically orders elements in a FIFO (first-in, first-out) manner.

Common Implementations: `LinkedList` **and** `PriorityQueue`

Example:

```
import java.util.LinkedList;
import java.util.Queue;
public class Queue {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("A");
        queue.add("B");
        queue.add("C");
        while (!queue.isEmpty()) {
            Sout(queue.poll()); // Retrieves and removes the head of the queue
        }
    }
}
```

Summary

- **Generics:** Provide type safety and reusability for classes, methods, constructors, and interfaces.

- **Generic Class:** A class that can operate on objects of various types.

- **Generic Method:** A method that can operate on objects of various types.

- **Generic Constructor:** A constructor declared with generic parameters.

- **Generic Interface:** An interface that can be implemented with various types.

- **Collections Framework:** A set of classes and interfaces for handling collections of objects.

- **List Interface:** An ordered collection that can contain duplicates.

- Queue Interface: A collection designed for holding elements prior to processing in a FIFO manner.

Collections Framework in Java

The Java Collections Framework (JCF) provides a unified architecture for representing and manipulating collections, enabling operations such as searching, sorting, insertion, and deletion.

Intro to Collections Framework:

The JCF is designed to handle groups of objects and provides:

- Interfaces: Define the operations that can be performed on collections.

- Implementations: Provide concrete implementations of the collection interfaces.

- Algorithms: Provide methods for manipulating and processing collections.

Collection Interfaces

Set Interface (Overview)

A `Set` is a collection that does not allow duplicate elements.

It models the mathematical set abstraction.

The `Set` interface does not guarantee any specific order of elements.

Common Implementations:

- HashSet: Provides constant time performance for basic operations and does not guarantee any order.

- Linked Hash Set: Maintains insertion order by using a linked list, which enables predictable iteration order.

- Tree Set: Implements a navigable set that uses a Red-Black tree, which sorts the elements according to their natural ordering or by a specified comparator.

Example:

```
import java.util.HashSet;

import java.util.Set;

public class Set {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("A");
        set.add("B");
        set.add("C");
        set.add("A"); // Duplicate entry will not be added
        for (String str : set) {
            Sout(str); // Order is not guaranteed
        }
    }
}
```

Collection Classes

Array List:

It is a resizable array implementation of the `List` interface.

It allows random access and maintains insertion order.

Key Points:

- Dynamic resizing of the underlying array.
- Provides fast access and slow insertions/removals.
- Allows `null` values.

Example:

```
import java.util.ArrayList;
import java.util.List;
public class ArrayList {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");
        Sout.("ArrayList: " + list);
    }
}
```

LinkedList

It is a doubly-linked list implementation of the `List` and `Deque` interfaces.

It allows for fast insertions and deletions.

Key Points:

- Better performance on frequent insertions and deletions compared to `ArrayList`.

- Implements `List` and `Deque` interfaces.

- Provides access to both ends of the list.

Example:

```
import java.util.LinkedList;
import java.util.List;
public class LinkedList {
    public static void main(String[] args) {
        List<String> list = new LinkedList<>();
        list.add("A");
        list.add("B");
        list.add("C");
        Sout("LinkedList: " + list);
    }
}
```

Collection Interfaces (Continued)

Iterator Interface

The `Iterator` interface provides a way to access elements of a collection sequentially without exposing the underlying structure.

Key Methods:

- **has Next( ):** Returns `true` if the iteration has more elements.

- **`next( )`:** Returns the next element in the iteration.

- **`remove( )`:** Removes from the underlying collection the last element returned by the iterator.

Example:

```
import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class Iterator{

    public static void main(String[] args) {

        List<String> list = new ArrayList<>();

        list.add("A");

        list.add("B");

        list.add("C");

        Iterator<String> iterator = list.iterator();

        while (iterator.hasNext()) {

            Sout(iterator.next());

        }

    }

}
```

Working with Maps (Overview)

The `Map` interface represents a collection of key-value pairs.

Keys are unique, and each key maps to exactly one value.

Common Implementations:

- **HashMap:** Provides constant time performance for basic operations and does not guarantee any order.

- **Linked Hash Map:** Maintains insertion order or access order.

- Tree Map: Implements a navigable map that sorts the keys according to their natural ordering or by a specified comparator.

Example:

```
import java.util.HashMap;

import java.util.Map;

public class Map {

    public static void main(String[] args) {

        Map<String, Integer> map = new HashMap<>();

        map.put("Alice", 30);

        map.put("Bob", 25);

        map.put("Charlie", 35);

        for (Map.Entry<String, Integer> entry : map.entrySet()) {

            Sout(entry.getKey() + ": " + entry.getValue());

        }

    }

}
```

Comparable & Comparator

Comparable Interface

It allows objects to be compared to each other for sorting.

Key Method:

- compareTo(T o): Compares the current object with the specified object.

Example:

```
class Person implements Comparable<Person> {

    private String name;

    private int age;

    public Person(String name, int age) {

        this.name = name;

        this.age = age;
    }
    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }
    @Override
    public String toString() {
        return name + ": " + age;
    }
}
public class Comparable {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));
        Collections.sort(people);
        Sout(people);
    }
}
```

Comparator Interface

It provides a way to compare objects of different types or with custom comparison logic.

Key Methods:

- compare(T o1, T o2): Compares its two arguments for order.

- equals(Object obj): Checks if the comparator is equal to another object.

Example:

```
import java.util.*;
```

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int getAge() {
        return age;
    }
    @Override
    public String toString() {
        return name + ": " + age;
    }
}
class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
}
public class Comparator {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 30));
        people.add(new Person("Bob", 25));
        people.add(new Person("Charlie", 35));
        Collections.sort(people, new AgeComparator());
        Sout(people);
    }
}
```

Summary

Generics: Enhance type safety and reusability in classes, methods, and interfaces.

Collections Framework: Provides interfaces and classes to handle groups of objects, including:

- **Set:** Collection with unique elements. Implementations include `HashSet`, `Linked Hash Set`, and `Tree Set`.
- **List:** Ordered collection with duplicates allowed. Implementations include `Array List` and `LinkedList`.
- **Queue:** Collection for holding elements prior to processing. Implementations include `LinkedList` and `PriorityQueue`.
- **Iterator:** Provides a way to access elements sequentially.
- **Map:** Represents key-value pairs. Implementations include `HashMap`, `Linked Hash Map`, and `Tree Map`.
- **Comparable:** Interface for natural ordering of objects.
- **Comparator:** Interface for custom ordering of objects.

Collections Framework in Java

The Java Collections Framework (JCF) provides a comprehensive architecture for managing and manipulating collections of objects.

It includes various interfaces, classes, and algorithms to work with data structures efficiently.

Collection Interfaces: Arrays, Vector, Stack

Arrays

Fundamental data structures that hold fixed-size sequences of elements of the same type.

They are not part of the Collections Framework but are frequently used in conjunction with it.

Key Points:

- **Fixed Size:** The size of an array is defined when it is created and cannot be changed.

- **Access:** Allows random access to elements using an index.

- **Types:** Can be single-dimensional or multidimensional.

Example:

```
public class Array {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};      // Single-dimensional array
        for (int number : numbers) {
            Sout(number);
        }
        int[][] matrix = {        // Multidimensional array
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };
        for (int[] row : matrix) {
            for (int num : row) {
                Sout(num + " ");
            }
            Sout( );
 }
    }
}
```

Vector

It is a resizable array implementation of the `List` interface.

It is synchronized, meaning it is thread-safe.

Key Points:

- **Resizable:** Automatically expands as elements are added.

- **Synchronized:** Thread-safe but can be less efficient due to synchronization overhead.

- **Legacy:** Part of the original Java 1.0 but still in use.

Example:

```
import java.util.Vector;

public class Vector {

    public static void main(String[] args) {

        Vector<String> vector = new Vector<>();

        vector.add("A");

        vector.add("B");

        vector.add("C");

        Sout("Vector: " + vector);

    }

}
```

Stack

It is a class that models a last-in, first-out (LIFO) stack of objects.

It extends `Vector` and provides methods to perform stack operations.

Key Points:

- **LIFO Order:** The last element added is the first to be removed.

- **Methods:** Includes `push()`, `pop()`, `peek()`, and `empty()`.

Example:

```
import java.util.Stack;

public class Stack {

    public static void main(String[] args) {

        Stack<String> stack = new Stack<>();

        stack.push("A");
        stack.push("B");
        stack.push("C");
        Sout("Top element: " + stack.peek()); // Outputs C
        while (!stack.empty( )) {
            Sout(stack.pop()); // Outputs C, B, A
        }
    }
}
```

IO Streams

Java provides a rich set of classes for input and output operations.

The `java.io` package contains classes for reading from and writing to files, network connections, and other sources.

Stream Classes: Byte Streams, Character Streams, Stream Tokenizer

Byte Streams

It handles I/O of raw binary data.

They are used for reading and writing binary data like image files and audio files.

Key Classes:

- **Input Stream:** Abstract class for byte input streams.
- **Output Stream:** Abstract class for byte output streams.
- **File Input Stream:** Reads bytes from a file.
- **File Output Stream:** Writes bytes to a file.

Example:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class ByteStream{
    public static void main(String[ ] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt");
             FileInputStream fis = new FileInputStream("output.txt"))
{
            fos.write("Hello, World!".getBytes( ));
            int byteRead;
            while ((byteRead = fis.read()) != -1) {
                Sout((char) byteRead);
            }
        } catch (IOException e) {
            e.printStackTrace( );
        }
    }
}
```

Character Streams

It handles I/O of characters and are designed for handling text data.

They automatically handle character encoding.

Key Classes:

- **Reader:** Abstract class for character input streams.
- **Writer:** Abstract class for character output streams.
- **File Reader:** Reads characters from a file.
- **File Writer:** Writes characters to a file.
- **Buffered Reader:** Reads text from a character-based input stream efficiently.
- **Buffered Writer:** Writes text to a character-based output stream efficiently.

Example:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class CharacterStream {

    public static void main(String[ ] args) {

        try (BufferedWriter bw = new BufferedWriter(new
FileWriter("output.txt"));

             BufferedReader br = new BufferedReader(new
FileReader("output.txt"))) {

            bw.write("Hello, World!");

            String line;

            while ((line = br.readLine()) != null) {

                Sout(line);

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

}
```

Stream Tokenizer

It is a utility class that can be used to parse streams of text into tokens.

It provides a way to read text input and break it into logical tokens (words, numbers, etc.).

Key Methods:

- **next Token( ):** Reads the next token from the stream.

- **t type( )**: Returns the type of the current token.

- **s val( )**: Returns the string value of the current token.

- **n val( )**: Returns the numeric value of the current token.

Example:

```
import java.io.FileReader;
import java.io.IOException;
import java.io.StreamTokenizer;
public class StreamTokenizer {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("example.txt")) {
            StreamTokenizer st = new StreamTokenizer(fr);
            int tokenType;
            while ((tokenType = st.nextToken()) != StreamTokenizer.TT_EOF) {
                switch (tokenType) {
                    case StreamTokenizer.TT_WORD:
                        Sout("Word: " + st.sval);
                        break;
                    case StreamTokenizer.TT_NUMBER:
                        Sout("Number: " + st.nval);
                        break;
                    default:
                        Sout("Other token: " + (char) tokenType);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Summary

Collections Framework: Provides a standardized way to handle groups of objects.

- **Arrays:** Fixed-size collections of elements.
- **Vector:** A synchronized resizable array.
- **Stack:** A last-in, first-out collection extending `Vector`.

IO Streams: Facilitate reading and writing of data.

- **Byte Streams:** Handle raw binary data (e.g., `FileInputStream`, `FileOutputStream`).
- **Character Streams:** Handle text data (e.g., `FileReader`, `FileWriter`).
- **Stream Tokenizer:** Parses streams into tokens for text processing.

JDBC Connectivity in Java

Java Database Connectivity (JDBC) is a Java-based API that allows Java applications to interact with relational databases.

It provides a standard interface for connecting to databases and executing SQL queries.

Intro to JDBC

JDBC provides a set of interfaces and classes for connecting to databases, executing SQL statements, and processing the results.

It abstracts the underlying database interactions and provides a uniform way to access different database systems.

Architecture of JDBC:

JDBC architecture consists of several components that work together to enable database connectivity:

1. **JDBC API:**

  Provides Java interfaces and classes for interacting with databases.

Key interfaces include `DriverManager`, `Connection`, `Statement`, `PreparedStatement`, `CallableStatement`, `ResultSet`, and `SQLException`.

2. JDBC Driver Manager:

Manages a list of database drivers.

Establishes a connection to the database using the appropriate driver.

3. JDBC Drivers:

Implement the JDBC API to communicate with a specific database.

Types of drivers:

Type 1: JDBC-ODBC Bridge Driver   **Type 2:** Native-API Driver

Type 3: Network Protocol Driver   **Type 4:** Thin Driver

4. **Database:**

The actual database system (e.g., MySQL, Oracle, SQL Server) that stores the data.

JDBC Architecture Diagram:

```
Application
   |
JDBC API
   |
JDBC Driver Manager
   |
JDBC Drivers (Type 1, 2, 3, 4)
   |
Database
```

Establishing JDBC Database Connection

To establish a JDBC database connection, you need to follow these steps:

1. **Load the JDBC Driver:**

Use `Class.forName()` to load the JDBC driver class.

This step registers the driver with the `DriverManager`.

Eg: Class.forName("com.mysql.cj.jdbc.Driver");

2. **Create a Connection:**

Use `DriverManager.getConnection()` to establish a connection to the database.

You need to provide the JDBC URL, username, and password.

Eg: Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase", "username", "password");

3. Create a Statement:

Create a `Statement`, `PreparedStatement`, or `CallableStatement` object to execute SQL queries.

Eg: Statement stmt = conn.createStatement();

4. **Execute a Query:**

Use the `Statement` object to execute SQL queries and obtain a `ResultSet` for query results.

Eg: ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");

5. **Process the ResultSet:** Iterate through the `ResultSet` to retrieve the data.

Eg:

```
 while (rs.next()) {
     int id = rs.getInt("id");
     String name = rs.getString("name");
     System.out.println("ID: " + id + ", Name: " + name);
 }
```

6. **Close the Resources:**

Close the `ResultSet`, `Statement`, and `Connection` to release the database resources.

Eg: rs.close(); stmt.close(); conn.close();

Complete Example:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class JDBC {
    public static void main(String[] args) {
        String jdbcUrl = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "root";
        String password = "password";
        try {
            Class.forName("com.mysql.cj.jdbc.Driver");   // Load JDBC Driver
 // Establish Connection
Connection conn = DriverManager.getConnection(jdbcUrl, username,
password);
       Statement stmt = conn.createStatement();        // Create Statement
ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");  // Execute
Query
            // Process ResultSet
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                Sout("ID: " + id + ", Name: " + name);
            }
            // Close Resources
            rs.close();
            stmt.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Summary

JDBC: Java API for database connectivity, providing a standard interface to interact with relational databases.

JDBC Architecture:

JDBC API: Interfaces and classes for database interaction.

JDBC Driver Manager: Manages database drivers and establishes connections.

JDBC Drivers: Specific implementations to connect to various databases.

Database: The actual database system.

Steps to Establish JDBC Connection:

1. Load the JDBC Driver: Register the driver class.

2. Create a Connection: Use `DriverManager.getConnection()` with JDBC URL, username, and password.

3. Create a Statement: Use `Statement`, `PreparedStatement`, or `CallableStatement`.

4. Execute a Query: Use the `Statement` object to run SQL queries.

5. Process the ResultSet: Retrieve and process the data from the `ResultSet`.

6. Close Resources: Release database resources by closing `ResultSet`, `Statement`, and `Connection`.