Introduction to java: Java is a high-level, object-oriented, robust, and secure programming language. It was developed by Sun Microsystems in 1995 and is now one of the most popular programming languages in the world. Java is used to develop a wide variety of applications, including web applications, mobile apps, desktop applications, and embedded systems. It is also used in the cloud and in big data processing.

• James Gosling is known as the father of Java.

• Flavours of Java: Core Java, Advanced Java, Android Java

- **Features of java programming language:**
  - **Open source:** Java is an open-source language, which means that it is free to use and distribute. There is a large community of Java developers who contribute to the language and its ecosystem.
  - **Object-Oriented:** Java is a fully object-oriented programming language, promoting the use of classes and objects, which make it easier to organize and manage code.
  - **Platform Independence:** Java is a platform-independent language, which means that Java programmes can run on any operating system that has a JVM. This makes Java a good choice for developing cross-platform applications.
  - **Embedded:** Java can be used to develop embedded systems, such as those found in cars, TVs, medical devices, and various other types of embedded hardware.
  - **Large library & framework:** Java has a large and comprehensive library of classes and frameworks. This makes it easy to develop a wide variety of applications.
  - **Compiler and interpreter:** Java is both a compiled and interpreted language. Java programs are first compiled into bytecode, which is then interpreted by the JVM.
  - **Simple and Clear Syntax:** Java has a relatively simple and readable syntax, making it accessible to developers and reducing the likelihood of errors.
  - **Robust:** Java incorporates features like strong type checking, automatic memory management, garbage collection, and exception handling, which contribute to its robustness and help to prevent common programming errors and vulnerabilities.
  - **Security:** Java has built-in security features such as classloaders, bytecode verification, a security manager, and secure sockets to ensure secure execution of code.
    - **Bytecode verification:** Java bytecode is verified before it is executed, which helps to prevent malicious code from being executed.
    - **Security manager:** The security manager allows the system administrator to control what resources an application can access.
    - **Secure sockets:** Java provides secure sockets, which provide a secure way for applications to communicate over a network.
  - **Strong type checking:** Java has strong type checking, which means that the compiler ensures that the types of variables and expressions are compatible. This helps to prevent errors at compile time and can help to improve the reliability of programs.
  - **Garbage collection:** Java Garbage collection used to automatically manage memory allocation and deallocation. This frees developers from having to worry about memory management, which can help to improve the reliability of programs.

- Analyze the output for both scenarios: when x is a static variable and when it is an instance variable.

If x is a static variable, it means that there is only one copy of x shared among all instances of the class B. Let's analyze the code with x as a static variable:

```
class B {
   private static int x;

   void fun1() {
      x = 2;
   }

   void fun2() {
      System.out.print(x);
   }

   void fun3() {
      x = 7;
   }
}

class A {
   public static void main(String[] args) {
      B obj = new B();
      B obj1 = new B();
      obj.fun1();       // Sets x to 2
      obj1.fun3();    // Sets x to 7
      obj.fun2();    // Prints the value of x, which is 7
   }
}
```

If x is an instance variable, each instance of class B will have its own copy of x. Let's analyze the code with x as an instance variable:

```
class B {
   private int x;
   void fun1() {
      x = 2;
   }

   void fun2() {
      System.out.print(x);
   }

   void fun3() {
      x = 7;
   }
}

class A {
   public static void main(String[] args) {
      B obj = new B();
      B obj1 = new B();
      obj.fun1();         // Sets obj's x to 2
      obj1.fun3();      // Sets obj1's x to 7
      obj.fun2();      // Prints the value of obj's x, which is 2
   }
}
```

Note: In a single Java file, there can be only one public class, and the name of the file must match the name of that public class. However, We can have multiple non-public (package-private, protected, or private) classes in the same file along with the one public class. The main function, which is the entry point of the program, can indeed be in any class, but it's often placed within the public class for organization and clarity.

Note: In Java, top-level classes (classes defined directly within a file) can only have public or default (package-private) access modifiers. Private and protected access modifiers are not allowed for top-level classes. However, a class that is a nested or inner class can indeed be private or protected, allowing more fine-grained control over its visibility.

- **Wrapper classes** in Java are classes that provide a way to represent primitive data types as objects. They "wrap" or encapsulate primitive types, such as **int**, **char**, **float**, etc., within an object. The primary purpose of wrapper classes is to provide utility methods, convert primitive types to objects, and enable interoperability with Java's object-oriented features like collections and generics.

  Here are some common wrapper classes:

  Integer (for int)
  Character (for char)
  Boolean (for boolean)
  Double (for double)
  Float (for float)
  Long (for long)
  Short (for short)
  Byte (for byte)

  Structure of Warp classes:

```
public class Integer {
   public static int parseInt(String s) {
      // Parses a String and returns an int
   }
   public static Integer valueOf(int i) {
      // Converts an int to an Integer object
   }
   public Integer(int value) {
      // Constructor to create an Integer object
   }
}
```

Example:

```
class Warp {
  public static void main(String args[]) {
    int x = Integer.parseInt("123");      // Convert String to int using parseInt

    Integer obj = Integer.valueOf("1011", 2);        // Convert binary String to Integer using valueOf

    int y = obj.intValue();    // Convert Integer to int using intValue

    System.out.print(x + "   " + y);          // Print the results
  }
}                                                          Output: 123   11
```

- The command-line arguments are values that are passed to a Java program when it is invoked from the command line or terminal. These arguments allow us to provide input to our Java program when it starts running. When we run a Java program, any additional information we provide after the Java command itself is considered a command-line argument.

```
// This is a Java class named CLS
class CLS {              // The main method is the entry point of the program
  public static void main(String args[]) {
    // Enhanced for loop to iterate through each command-line argument
    for (String s : args) {
      System.out.print(s);            // Print each command-line argument
    }
  }
}
```

Compile: javac CLS.java↵ Run: java CLS ankit Abhi↵

- **Package programme:**

```
package Package1;
public class Manipulation {          // Class for manipulating integers
  int a, b;

    public void setData(int x, int y) {       // Method to set the values of integers
    a = x;
    b = y;
  }

  public void printData() {            // Method to print the multiplication of the entered numbers
    int m = a * b;
    System.out.print("Multiplication of entered numbers= " + m);
  }
}

package Package2;
import Package1.Manipulation;
import java.util.Scanner;

class Handle {      // Class to handle user input and manipulation of integers
  public static void main(String args[]) {
    Scanner scan = new Scanner(System.in);            // Scanner object to take user input

    System.out.print("Enter two integer numbers= ");        // Prompting the user to enter two integer numbers
    int a = scan.nextInt();
    int b = scan.nextInt();

    Manipulation obj = new Manipulation();              // Creating an object of Manipulation class

   obj.setData(a, b);         // Setting the entered data using setData() method
```

```
        obj.printData();        // Printing the multiplication result using printData() method
    }
}
```

```
D:\java>javac -d . Manipulation.java        // Compilation of Package1

D:\java>javac -d . Handle.java              // Compilation of Package2

D:\java>java Package2.Handle                // Run Package2
Enter two integer number= 6
7
Multiplication of entered numbers= 42
```

- **Note:** In Java, starting from version 8, we can define static methods within interfaces. These methods can have a body implemented within the interface itself. We call these static methods using the name of the interface.

```
interface MyInterface {
    void myMethod();     // Abstract method

    static void myStaticMethod() {
        System.out.println("Static method in interface");
    }
}

class MyClass implements MyInterface {
    public void myMethod() {
        System.out.println("Implementing myMethod");
    }
}

public class Main {
    public static void main(String[] args) {
        MyInterface.myStaticMethod();       // Calling static method using interface name
        MyClass obj = new MyClass();
        obj.myMethod();      // Implementing class method call
    }
}
```

- **Note:** In Java, static variables and static methods are not inherited in the traditional sense, as they belong to the class itself rather than to instances of the class. However, subclasses can access static variables and methods of their superclass using the superclass name directly. Additionally, static members of the superclass can be accessed using the subclass name from other classes**.**

- **Multi-threading** enables the creation of programs that execute multiple threads concurrently. This can be achieved through two primary approaches:

1. **Implementing the Runnable interface:**
   - Define a class that implements the Runnable interface.
   - Implement the run() method from the Runnable interface, specifying the code that will run in the new thread.
   - Create an instance of this class.
   - Pass the instance to the constructor of the Thread class.
   - Call the start() method on the Thread instance to start execution.

2. **Extending the Thread class:**
   - Define a class that extends the Thread class.
   - Override the run() method within this class to specify the code that will run in the new thread.
   - Create an instance of this class and call its start() method to begin execution.

Example 1:

```
class Process1 implements Runnable {
    // Override the run method which defines the task for this thread
    public void run(){
        for(int i = 1; i < 5; i++) {
```

```
                System.out.println("Process1: " + i);
            }
        }
    }

    class Process2 implements Runnable {
        // Override the run method which defines the task for this thread
        public void run() {
            for(int i = 1; i < 5; i++) {
                System.out.println("Process2: " + i);
            }
        }
    }

    // Define a class MultiT (Multithreading) which contains the main method
    class MultiT {
        public static void main(String[] args) {
            Process1 p1 = new Process1();           // Create instances of Process1 and Process2
            Process2 p2 = new Process2();

            // Create threads using the Thread class, passing instances of Process1 and Process2 as arguments
            Thread t1 = new Thread(p1); // Thread for Process1
            Thread t2 = new Thread(p2); // Thread for Process2

            // Start the execution of threads
            t1.start();          // Start the execution of the thread associated with Process1
            t2.start();         // Start the execution of the thread associated with Process2
        }
    }
```

Example2:

```
class Process1 extends Thread
{
  public void run()
   {
    for(int i=1; i<5; i++)
     {
       System.out.println("Process1: "+i);
     }
   }
}

class Process2 extends Thread
{
  public void run()
   {
    for(int i=1; i<5; i++)
     {
       System.out.println("Process2: "+i);
     }
   }
}

class MultiT {
 public static void main(String[] args) {
  Process1 p1=new Process1();
  Process2 p2=new Process2();

  p1.start();
  p2.start();
 }
}
```

- **Synchronization** in Java is used to control access to shared resources in a multithreaded environment. When multiple threads are operating concurrently, there can be scenarios where they may access and modify shared data simultaneously, leading to race conditions, data corruption, and unpredictable behavior. Synchronization helps to prevent such issues by ensuring that only one thread can access the shared resource at a time, thus maintaining data consistency and integrity.

  Reasons for use:

  Thread safety: Ensuring threads don't interfere with each other's access to shared resources.

  Preventing race conditions: Avoiding unpredictable outcomes due to concurrent access to shared data.

  Maintaining data consistency: Guaranteeing the integrity of shared data by ensuring correct updates.

  Critical sections: Protecting sensitive code blocks that require exclusive access.

  Preventing deadlocks: Avoiding situations where threads wait indefinitely for locked resources.

  Coordinating thread execution: Synchronizing access to shared resources for orderly execution.

Switch:

```
import java.util.Scanner;

public class MonthExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the month number (1-12): ");
        int monthNumber = scanner.nextInt();

        String monthName = switch (monthNumber) {
            case  1 -> "January";
            case  2 -> "February";
            case  3 -> "March";
            case  4 -> "April";
            case  5 -> "May";
            case  6 -> "June";
            case  7 -> "July";
            case  8 -> "August";
            case  9 -> "September";
            case  10 -> "October";
            case  11 -> "November";
            case  12 -> "December";
            default -> "Invalid month number";
        };

        System.out.println("The month is " + monthName);
        scanner.close();
    }
}
```

- **Java Swing:** Swing is a Java library used for creating graphical user interfaces (GUIs) in desktop applications. It offers a comprehensive set of components like buttons, labels, text fields, checkboxes, radio buttons, menus, and more, which developers can combine and customize to create sophisticated user interfaces. Swing is platform-independent, highly customizable, and supports event-driven programming. It is commonly used for developing desktop applications in Java.

Example1:

```
import javax.swing.*;       // Import the JFrame class from the javax.swing package

class JavaSwing {          // Define a class named JavaSwing

  public static void main(String args[]) {      // Define the main method, which is the entry point for the Java application

    JFrame j1 = new JFrame();          // Create a new JFrame instance
```

```
        j1.setSize(500, 500);          // Set the size of the JFrame
        j1.setVisible(true);           // Make the JFrame visible
        j1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);      // Set the default close operation
      }
    }
```

Example2:

```
    import javax.swing.*;
    public class JavaSwing extends JFrame {
      JavaSwing(String s1) {          // Constructor that takes a string argument to set the title of the JFrame
        super(s1);
      }

        JavaSwing() {       // Default constructor that sets a default title for the JFrame
        super("Java Swing Application");
      }

  void setComponents() {      // Method to set up the components of the JFrame
          JLabel l1 = new JLabel("WELCOME");          // Create a JLabel with the text "WELCOME"
          setLayout(null);  // Set the layout of the JFrame to null, which means you will manually set the bounds of each component
          l1.setBounds(200, 200, 100, 40);    // Set the bounds of the JLabel to position it at (200, 200) with a size of 100x40 pixels
        add(l1);     // Add the JLabel to the JFrame
        l1.setVisible(true);          // Make the JLabel visible
      }
      public static void main(String args[]) {       // Main method to create and display the JFrame
        // Create a new instance of JavaSwing with the title "Welcome to Java Swing"
        JavaSwing js = new JavaSwing("Welcome to Java Swing");
        js.setSize(500, 500);      // Set the size of the JFrame to 500x500 pixels
        js.setVisible(true);          // Make the JFrame visible

        // Set the default close operation to exit the application when the JFrame is closed
        js.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        js.setComponents();     // Call the setComponents method to add the JLabel to the JFrame
      }
    }
```

- **File Handling:** File handling in Java refers to the process of working with files stored on the filesystem. It involves tasks such as creating, reading, writing, deleting, and manipulating files and directories using classes like File, FileInputStream, FileOutputStream, BufferedReader, BufferedWriter, etc.

Example:

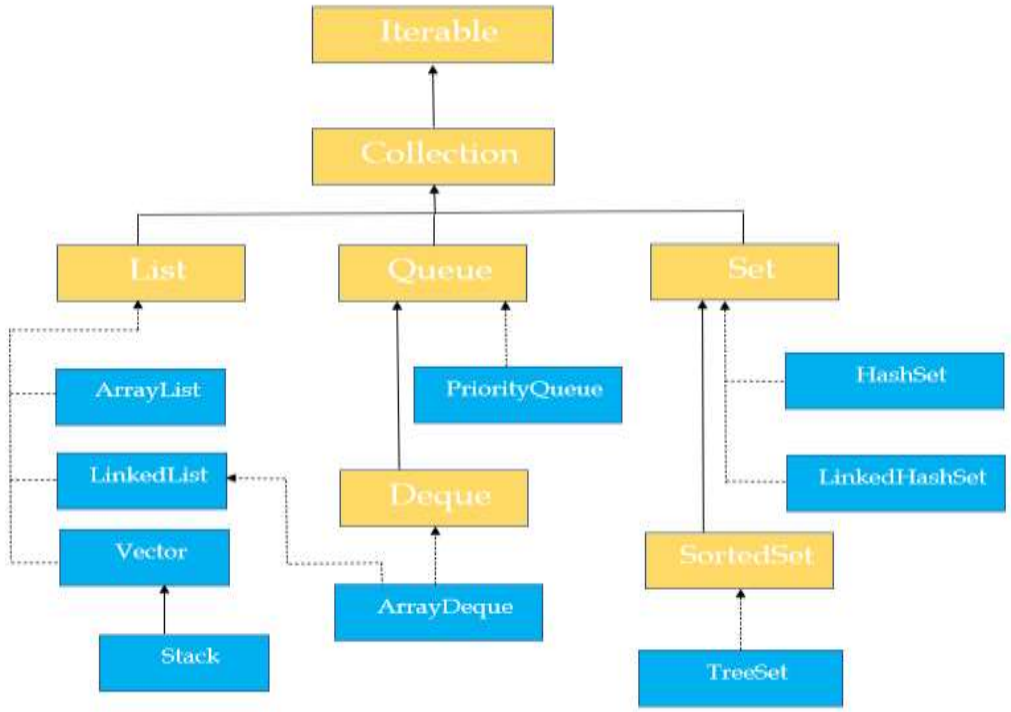
```
import java.io.*;
class FileJ {
  public static void main(String args[]) {
    File f1 = new File("C:\\Users\\ankit.shukla\\Desktop\\java_logging\\excelfiles\\ID1_file.txt");

    System.out.print(f1.exists());
    System.out.println(f1.canWrite());                    //f1.createNewFile();
    System.out.print(f1.length());
  }
}
```

Output: true

- **FileOutputStream** in Java is a class used for writing data to a file. It is part of the **java.io** package. This class is typically used to create a new file and write data to it or to overwrite existing data in a file.

- **A collection is a single unit of objects that are grouped together**. Collections are used to store, retrieve, manipulate, and communicate aggregate data. They are part of the Java Collections Framework, which is a unified architecture for representing and manipulating

collections. The framework provides a rich set of interfaces and classes for working with collections, including lists, sets, and maps.



Classes:

- **ArrayList:** Implements the List interface with a resizable-array implementation, offering fast random access to elements.
- **LinkedList:** Implements the List and Deque interfaces as a doubly-linked list.
- **HashSet:** Implements the Set interface using a hash table, without guaranteeing any specific order of elements.
- **LinkedHashSet:** Extends HashSet and maintains a doubly-linked list running through all of its entries, defining the iteration ordering based on insertion order.
- **TreeSet:** Implements the Set interface with a tree-based data structure, guaranteeing ascending element order within the set.
    - ✓ Java Full course: https://youtu.be/32DLasxoOiM?si=guoy_JWypVjjybKL

॥ॐ नमः शिवाय॥

|| ॐ तत्पुरुषाय विद्महे, महादेवाय धीमहि, तन्नो रूद्र प्रचोदयात् || ॐ गं गणपतये नमः ||

♨ ☆ ♨

Thank you