Unit– 1: Object Oriented Programming Concepts

Introduction to Java:

Java is a high-level, versatile, and object-oriented programming language developed by James Gosling and his team at Sun Microsystems in the 1990s. It is designed to be platform-independent, allowing developers to write code that can run on any device that has the Java Virtual Machine (JVM). Java has become one of the most widely used programming languages, particularly in enterprise-level applications, mobile development (Android), and web development.

Brief History of Java:

1991-1992: The project that would become Java started at Sun Microsystems under the codename "Green." James Gosling and his team aimed to create a programming language for consumer electronic devices.

1995: Java 1.0 was released to the public. It included features like applets for web browsers, which contributed to Java's early popularity.

1996: Sun Microsystems released Java 1.1 with significant improvements, including inner classes and JDBC (Java Database Connectivity).

2004: Java 5 (or J2SE 5.0) was a major release introducing generics, metadata annotations, the enhanced for loop, and the Java Virtual Machine (JVM) improvements.

2011: Oracle Corporation acquired Sun Microsystems. Subsequent versions of Java continued to be developed and released under the Oracle brand.

2014: Java 8 was a significant release with the introduction of lambda expressions, the Stream API, and the java.time package for modern date and time handling.

2017: Java 9 introduced the module system, enhancing modularity and encapsulation.

2018: Java 10 and Java 11 brought features like local-variable type inference, the HTTP client API, and long-term support (LTS) for Java 11.

2020: Java 14, Java 15, and subsequent versions introduced features like records, pattern matching, and enhancements to the garbage collector.

Java Features:

Simple:

Java was designed to be easy to use and understand. It eliminates complex features such as operator overloading, pointers, and explicit memory management.
Object-Oriented:

Java follows the object-oriented programming (OOP) paradigm, allowing developers to create modular and reusable code through the use of classes and objects.

Platform-Independent:

Java programs can run on any device with a Java Virtual Machine (JVM), making them platform-independent. The "Write Once, Run Anywhere" (WORA) principle is a key feature of Java.

Distributed Computing:

Java supports distributed computing by providing built-in networking capabilities. RMI (Remote Method Invocation) and Java RMI-IIOP (Java Remote Method Invocation over Internet Inter-ORB Protocol) are examples.

Multithreaded:

Java supports concurrent programming with multithreading. This allows multiple tasks to be executed simultaneously, enhancing the performance of applications.

Dynamic:

Java is designed to adapt to an evolving environment. It supports dynamic loading of classes, dynamic compilation, and automatic memory management through garbage collection.

Secure:

Java provides a secure runtime environment with features like bytecode verification, runtime security checks, and a robust access control mechanism.

High Performance:

While Java is not as low-level as languages like C or C++, it still offers good performance through Just-In-Time (JIT) compilation and optimization.

Java Applications:

Enterprise-Level Applications:

Java is widely used for developing large-scale enterprise applications, including banking systems, customer relationship management (CRM) software, and human resource management systems.

Web Development:

Java is used for building dynamic and scalable web applications. Popular frameworks like Spring and JavaServer Faces (JSF) are widely used in web development.

Mobile Applications (Android):

Java is the primary programming language for Android app development. Android applications are typically written in Java and run on the Android Runtime (ART).

Desktop Applications:

Java is used to develop desktop applications with graphical user interfaces (GUIs). The Swing and JavaFX libraries facilitate the creation of cross-platform desktop applications.

Embedded Systems:

Java's platform independence makes it suitable for embedded systems, where devices with different architectures may run the same Java code.

Scientific and Research Applications:

Java is used in scientific research and applications, providing tools for data analysis, simulations, and complex algorithms.

Cloud-Based Applications:

Java is used to build applications that run on cloud platforms. Java applications can be deployed on cloud services like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP).

Java's versatility, combined with its features like platform independence, object-oriented nature, and security, has contributed to its enduring popularity in various domains of software development.

Java Components:

Java Virtual Machine (JVM):

Definition: The JVM is an integral part of the Java Runtime Environment (JRE) and executes Java bytecode. It provides a platform-independent abstraction between the compiled Java program and the underlying hardware and operating system.

Key Functions:

Loads and executes Java bytecode.
Manages memory (heap and stack).
Provides runtime environment for Java applications.
Performs just-in-time (JIT) compilation for improved performance.
Importance: Enables the portability of Java applications across different platforms by executing the same bytecode on any system with a compatible JVM.

Java Runtime Environment (JRE):

Definition: The JRE is a package of software components that provides the runtime environment for Java applications. It includes the JVM, libraries, and other components required for executing Java programs.

Key Components:

Java Virtual Machine (JVM)
Java Class Libraries
Java Runtime API
Importance: Allows end-users to run Java applications without the need for development tools. Provides the necessary runtime support for Java programs.

Java Development Kit (JDK):

Definition: The JDK is a comprehensive software development kit that includes the JRE along with additional tools and utilities needed for Java application development.

Key Components:

Java Compiler (javac): Translates Java source code into bytecode.
Java Debugger (jdb): Helps in debugging Java programs.
Java Documentation Generator (javadoc): Generates API documentation.
Other development tools and utilities.
Importance: Essential for Java developers as it provides all the tools needed for writing, compiling, and testing Java applications.

Importance of Bytecode:

Portability:

Bytecode is an intermediate representation of the Java source code that is platform-independent. It allows Java applications to run on any device with a compatible JVM, promoting the "Write Once, Run Anywhere" (WORA) principle.
Security:

Bytecode is executed by the JVM, which performs various security checks during runtime. This helps in creating a secure execution environment by preventing malicious activities.
Efficiency:

The use of bytecode enables just-in-time (JIT) compilation. Bytecode is translated to machine code at runtime, optimizing the execution speed and efficiency of Java programs.

Garbage Collection:
Automatic Memory Management:

Garbage collection is a feature in Java that automatically manages the memory by reclaiming the memory occupied by objects that are no longer referenced or in use.
Prevents Memory Leaks:

Java's garbage collector identifies and removes objects that are no longer reachable, preventing memory leaks and ensuring efficient memory utilization.
Simplifies Memory Management:

Developers don't have to manually allocate and deallocate memory, reducing the risk of memory-related errors. Garbage collection simplifies memory management and makes Java applications more robust.
Promotes Productivity:

With automatic garbage collection, developers can focus more on writing application logic and less on memory management tasks. This promotes productivity and code quality.
Optimizes Application Performance:

Garbage collection algorithms aim to minimize pauses and optimize the performance of Java applications. Modern garbage collectors, such as the G1 Garbage Collector, provide a good balance between throughput and low-latency requirements.

In summary, the Java Virtual Machine (JVM), Java Runtime Environment (JRE), and Java Development Kit (JDK) form the foundation for Java programming, providing the runtime environment and tools necessary for application development. Bytecode facilitates platform independence, while garbage collection automates memory management, enhancing the reliability and efficiency of Java applications.

Primitive Data Types : byte, short, int, long, float, double, char, Boolean

In Java, primitive data types are the basic building blocks for storing and manipulating data. They are predefined by the language and are not objects. Here are the commonly used primitive data types in Java:

byte:

Size: 8 bits
Range: -128 to 127
Example: byte myByte = 100;

short:

Size: 16 bits
Range: -32,768 to 32,767
Example: short myShort = 2000;

int:

Size: 32 bits
Range: -2^31 to 2^31 - 1
Example: int myInt = 100000;

long:

Size: 64 bits
Range: -2^63 to 2^63 - 1
Example: long myLong = 1000000000L;

float:

Size: 32 bits
Example: float myFloat = 3.14f;

double:

Size: 64 bits
Example: double myDouble = 3.14;

char:

Size: 16 bits
Range: 0 to 65,535
Example: char myChar = 'A';

boolean:

Size: Not precisely defined
Values: true or false
Example: boolean isJavaFun = true;

Notes:
Size: The size indicates the number of bits used to represent the data type in memory.
Range: Specifies the minimum and maximum values that can be stored in the data type.
Literals: The examples provided show how to declare variables and assign values to them using literals.

Additional Information:

Floating-Point Literals:

If you write a decimal literal (e.g., 3.14), it is treated as a double by default. To specify a float literal, you can append an 'f' or 'F' (e.g., 3.14f).
Character Literals:

Characters are enclosed in single quotes (e.g., 'A'). Unicode characters can be represented using the escape sequence (e.g., '\u0041' for 'A').
Boolean Literals:

true and false are the only boolean literals.

These primitive data types provide a way to represent and work with basic data in Java. They are essential for storing variables, performing calculations, and creating the foundation for more complex data structures and classes.

Identifiers, Declarations of constants:

Identifiers:
Identifiers are names given to various elements in a Java program, such as classes, variables, methods, and labels. Identifiers are user-defined and follow certain rules:

Rules for Identifiers in Java:

Must begin with a letter (uppercase or lowercase), underscore _, or dollar sign $.
Subsequent characters can be letters, digits, underscores, or dollar signs.
Java is case-sensitive, so myVar and MyVar are different identifiers.
Cannot be a reserved keyword (e.g., class, if, int) as identifiers.
Should not be the same as Java standard library classes or methods to avoid naming conflicts.

Examples of Identifiers:

```
int myVar;
String userName;
double _totalAmount;
```

Declarations of Constants:
Constants are variables whose values do not change during the execution of a program. In Java, constants are typically declared using the final keyword. Constants are often written in uppercase letters with underscores separating words (conventionally).

Declaration of Constants:

```
final double PI = 3.14159;
final int MAX_VALUE = 100;
```

Key Points:

The final keyword indicates that the variable's value cannot be modified after it is assigned.
Constants are often declared as static final variables within classes when they are shared among instances of the class.
Naming conventions for constants usually involve uppercase letters and underscores (e.g., MAX_SIZE, DEFAULT_COLOR).

Example:

```
public class Circle {
   static final double PI = 3.14159;
   final double radius;

   public Circle(double radius) {
      this.radius = radius;
   }

   public double calculateArea() {
      return PI * radius * radius;
   }
}
```

Using final with Methods:

The final keyword can also be used to declare that a method in a class cannot be overridden by subclasses. For example:

```
public class Example {
   final void myMethod() {
      // Method implementation
   }
}
```

Constants play a crucial role in making programs more readable and maintainable by providing meaningful names for values that are not meant to change during the program's execution.

variables, Type Conversion and Type Casting, Scope of variables

Variables:
In Java, variables are containers used to store data values. They have a specific data type, which defines the type of data they can hold. Variables are declared with a name and a data type and can be assigned values, which can change during the program's execution.

Example of Variable Declaration and Assignment:

```
int age;         // Declaration of an integer variable named 'age'
age = 25;        // Assignment of a value to the 'age' variable
```

Common Data Types for Variables:

int: Integer (e.g., 42)
double: Double-precision floating-point (e.g., 3.14)
char: Character (e.g., 'A')
boolean: Boolean (either true or false)
String: String of characters (e.g., "Hello")

Type Conversion and Type Casting:

1. Type Conversion (Implicit or Automatic):
Java automatically converts one data type to another when needed, but it only does so if there is no loss of information or precision.

```
int intValue = 42;
double doubleValue = intValue;  // Implicit conversion (int to double)
```

2. Type Casting (Explicit or Manual):
Type casting is the process of converting a variable from one data type to another manually. This is required when there may be a loss of precision.

```
double doubleValue = 3.14;
int intValue = (int) doubleValue;  // Explicit casting (double to int)
```

Scope of Variables:

1. Local Variables:
Declared inside a method, constructor, or block.
Only accessible within the block where they are declared.
Not accessible outside the method or block.

```
public void exampleMethod() {
   int localVar = 10;  // Local variable
   // localVar is only accessible within this method
}
```

2. Instance Variables (Non-Static Variables):
Declared within a class but outside any method, constructor, or block.
Each instance (object) of the class has its own copy.
Accessible throughout the class.

```
public class Example {
   int instanceVar = 20;  // Instance variable
   // instanceVar is accessible within the entire class
}
```

3. Class Variables (Static Variables):
Declared with the static keyword within a class but outside any method, constructor, or block.
Shared by all instances of the class.
Accessible using the class name.

```
public class Example {
   static int classVar = 30;  // Class variable
   // classVar is shared among all instances of the class
}
```

4. Parameters:
Variables passed as arguments to methods.
Similar to local variables in terms of scope.

Their scope is limited to the method or block they are passed to.

```
public void methodWithParameters(int param1, double param2) {
   // param1 and param2 are parameters with local scope in this method
}
```

The scope of a variable determines where it can be accessed and modified within a program. Understanding the scope is essential for writing modular and maintainable code. Local variables have limited scope, while instance and class variables have broader scopes depending on their context in the class.

Arrays of Primitive Data Types, Types of Arrays : one-dimensional and two dimensional array

Arrays of Primitive Data Types:
In Java, an array is a container object that holds a fixed number of values of a single data type. Arrays can be created for primitive data types, such as int, double, char, etc. They provide a way to group elements of the same type under a single name.

One-Dimensional Array:
A one-dimensional array is a collection of elements stored in a linear sequence. Each element is identified by its index, starting from 0. In Java, the declaration and initialization of a one-dimensional array are as follows:

```
// Declaration
dataType[] arrayName;

// Initialization
arrayName = new dataType[length];

// Example
int[] numbers = new int[5];
```

Example of One-Dimensional Array:

```
// Declaration and Initialization
int[] numbers = new int[5];

// Assigning values
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;

// Accessing values
int thirdElement = numbers[2]; // Returns 30
```

Two-Dimensional Array:

A two-dimensional array is an array of one-dimensional arrays. It is like a matrix with rows and columns. In Java, a two-dimensional array is declared and initialized as follows:

```
// Declaration
dataType[][] arrayName;

// Initialization
arrayName = new dataType[rows][columns];

// Example
int[][] matrix = new int[3][4];
```

Example of Two-Dimensional Array:

```
// Declaration and Initialization
int[][] matrix = new int[3][4];

// Assigning values
matrix[0][0] = 1;
matrix[0][1] = 2;
matrix[0][2] = 3;
matrix[0][3] = 4;

matrix[1][0] = 5;
matrix[1][1] = 6;
matrix[1][2] = 7;
matrix[1][3] = 8;

matrix[2][0] = 9;
matrix[2][1] = 10;
matrix[2][2] = 11;
matrix[2][3] = 12;

// Accessing values
int element = matrix[1][2]; // Returns 7
```

Key Points:

Arrays in Java are zero-indexed, meaning the first element is at index 0.
The length of an array is fixed upon initialization and cannot be changed.
Arrays provide a convenient way to work with collections of data.
For two-dimensional arrays, each element is identified by two indices (row and column).

Arrays are an essential part of Java programming, offering a structured way to store and manipulate data. They are used in a variety of applications ranging from simple data storage to complex algorithms and data structures.

Different Operators: Arithmetic, Bitwise, Rational, Logical, Assignment, Conditional, Ternary, Increment and Decrement

Arithmetic Operators:
Arithmetic operators are used for performing mathematical operations on numeric values.

Addition (+): Adds two operands.

Subtraction (-): Subtracts the right operand from the left operand.

Multiplication (*): Multiplies two operands.

Division (/): Divides the left operand by the right operand.

Modulus (%): Returns the remainder of the division of the left operand by the right operand.

Bitwise Operators:
Bitwise operators perform operations at the bit level. They are used to manipulate individual bits of integer operands.

Bitwise AND (&): Performs a bitwise AND operation.

Bitwise OR (|): Performs a bitwise OR operation.

Bitwise XOR (^): Performs a bitwise exclusive OR operation.

Bitwise NOT (~): Inverts the bits of its operand.

Left Shift (<<): Shifts the bits of the left operand to the left by a specified number of positions.

Right Shift (>>): Shifts the bits of the left operand to the right by a specified number of positions. The sign bit is used for the shift.

Relational Operators:
Relational operators are used to compare values and return a boolean result.

Equal to (==): Checks if two operands are equal.

Not equal to (!=): Checks if two operands are not equal.

Greater than (>): Checks if the left operand is greater than the right operand.

Less than (<): Checks if the left operand is less than the right operand.

Greater than or equal to (>=): Checks if the left operand is greater than or equal to the right operand.

Less than or equal to (<=): Checks if the left operand is less than or equal to the right operand.

Logical Operators:
Logical operators perform logical operations on boolean values.

Logical AND (&&): Returns true if both operands are true.

Logical OR (||): Returns true if at least one of the operands is true.

Logical NOT (!): Inverts the boolean value of its operand.

Assignment Operators:
Assignment operators are used to assign values to variables.

Assignment (=): Assigns the value of the right operand to the left operand.

Add and Assign (+=): Adds the right operand to the left operand and assigns the result to the left operand.

Subtract and Assign (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.

Multiply and Assign (*=): Multiplies the left operand by the right operand and assigns the result to the left operand.

Divide and Assign (/=): Divides the left operand by the right operand and assigns the result to the left operand.

Modulus and Assign (%=): Calculates the modulus of the left operand by the right operand and assigns the result to the left operand.

Conditional (Ternary) Operator:
The ternary operator (? :) is a shorthand for an if-else statement.

```
int result = (condition) ? valueIfTrue : valueIfFalse;
```
If the condition is true, valueIfTrue is assigned to result; otherwise, valueIfFalse is assigned.

Increment and Decrement Operators:
Increment (++): Increases the value of the operand by 1.

Decrement (--): Decreases the value of the operand by 1.

These operators can be used as prefix (++x, --x) or postfix (x++, x--). The difference is in the order of operation: prefix increments/decrements before the current value is used, while postfix increments/decrements after the current value is used.

These operators are fundamental to performing various operations in Java and are used extensively in coding for arithmetic, logical, and bitwise manipulations.

Decision & Control Statements:

In Java, decision and control statements are used to alter the flow of execution based on certain conditions or to repeatedly execute a block of code. These statements help in creating more dynamic and flexible programs.

Selection Statements:
1. if Statement:
The if statement allows conditional execution of a block of code.

```
if (condition) {
   // Code to be executed if the condition is true
}
```

2. if...else Statement:
The if...else statement allows for two possible outcomes based on a condition.

```
if (condition) {
   // Code to be executed if the condition is true
} else {
   // Code to be executed if the condition is false
}
```

3. switch Statement:
The switch statement is used to select one of many code blocks to be executed.

```
int day = 3;
switch (day) {
   case 1:
      System.out.println("Monday");
      break;
   case 2:
      System.out.println("Tuesday");
      break;
   // ... other cases ...
   default:
      System.out.println("Invalid day");
}
```

Loops:
1. while Loop:
The while loop repeatedly executes a block of code as long as the specified condition is true.

```
while (condition) {
   // Code to be executed in each iteration
}
```

2. do-while Loop:
The do-while loop is similar to the while loop, but the condition is checked after the block of code is executed, guaranteeing at least one execution.

```
do {
   // Code to be executed in each iteration
} while (condition);
```

3. for Loop:
The for loop is used for iterating over a range of values.

```
for (initialization; condition; update) {
   // Code to be executed in each iteration
}
```

Jump Statements:
1. break Statement:
The break statement is used to terminate the loop or switch statement.

```
for (int i = 0; i < 10; i++) {
   if (i == 5) {
      break; // Terminates the loop when i equals 5
   }
   // Code to be executed in each iteration
}
```

2. continue Statement:
The continue statement is used to skip the rest of the code inside a loop for the current iteration and jump to the next iteration.

```
for (int i = 0; i < 10; i++) {
   if (i % 2 == 0) {
      continue; // Skips even numbers and continues with the next iteration
   }
   // Code to be executed for odd numbers
}
```

3. return Statement:
The return statement is used to exit a method and optionally return a value.

```
public int add(int a, int b) {
   return a + b; // Exits the method and returns the sum of a and b
}
```

These decision and control statements provide the foundation for creating complex and dynamic programs by allowing the execution of specific blocks of code based on conditions or by repeatedly executing a block of code.

Unit– 2: Object Oriented Programming Concepts

Procedure-Oriented vs. Object Oriented Programming concept

Procedure-oriented programming (POP) and object-oriented programming (OOP) are two different programming paradigms that guide the organization and structure of code. Let's explore the key concepts of each:

Procedure-Oriented Programming (POP):

Focus on Procedures or Functions:

POP is centered around procedures or functions that perform specific tasks.
The program is organized as a series of functions that operate on data.

Data is Separate from Procedures:

Data and procedures are kept separate in POP.
Data is often organized in structures or records, and functions operate on this data.

Global Data:

Data is usually global and accessible to all functions.
Functions may modify global data, leading to potential issues with data integrity.

Top-Down Approach:

POP often follows a top-down approach where the program is designed as a sequence of steps from the main procedure.

Procedural Abstraction:

Procedural abstraction is used to break down a program into smaller, manageable procedures or functions.

Object-Oriented Programming (OOP):
Focus on Objects:

OOP revolves around objects, which are instances of classes.
An object encapsulates data and behavior (methods) into a single unit.

Encapsulation:

Encapsulation involves bundling data and methods that operate on that data within a single unit (object).

Access to the data is controlled through the methods of the object.
Inheritance:

Inheritance allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class).
This promotes code reuse and establishes an "is-a" relationship between classes.

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common base class.
This facilitates flexibility and extensibility in the code.

Abstraction:

Abstraction involves simplifying complex systems by modeling classes based on real-world entities and their interactions.

It focuses on hiding the unnecessary details and exposing only what is essential.
Message Passing:

Objects communicate with each other through message passing.
Objects send messages to request services or invoke methods on other objects.

Comparison:
Data Handling:

POP: Data is separate and often global.
OOP: Data is encapsulated within objects, and access is controlled through methods.

Code Organization:

POP: Code is organized as a set of functions.
OOP: Code is organized around classes and objects.

Flexibility and Reusability:

POP: Limited flexibility and reusability.
OOP: Promotes flexibility, modularity, and code reuse.

Complexity Management:

POP: May become complex as the program size increases.
OOP: Provides better tools for managing complexity through encapsulation, inheritance, and polymorphism.

Basics of OOP: Class, Object, Encapsulation, Polymorphism Abstraction, Inheritance

1. Class:
Definition: A class is a blueprint or template for creating objects. It defines a set of attributes (data members) and methods (functions) that the objects of the class will have.

Example:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def display_info(self):
        print(f"{self.make} {self.model}")
```

2. Object:
Definition: An object is an instance of a class. It is a concrete entity created based on the blueprint provided by the class, possessing the attributes and behaviors defined by the class.

Example:

```
my_car = Car("Toyota", "Camry")
my_car.display_info()  # Output: Toyota Camry
```

3. Encapsulation:
Definition: Encapsulation is the bundling of data (attributes) and the methods (functions) that operate on the data within a single unit, i.e., a class. It helps in controlling access to the data and ensures data integrity.

Example:

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number   # Encapsulation using double
underscores
        self.__balance = balance

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")
```

4. Polymorphism:
Definition: Polymorphism allows objects of different classes to be treated as objects of a common base class. It provides a way to use a single interface for different types of objects.

Example:

```
# Polymorphism through method overriding
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Usage
my_dog = Dog()
my_cat = Cat()
print(my_dog.speak())  # Output: Woof!
print(my_cat.speak())  # Output: Meow!
```

5. Abstraction:
Definition: Abstraction involves simplifying complex systems by modeling classes based on real-world entities and their interactions. It focuses on hiding the unnecessary details and exposing only what is essential.

Example:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

6. Inheritance:

Definition: Inheritance allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class). It promotes code reuse and establishes an "is-a" relationship between classes.

Example:

```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Usage
my_dog = Dog()
my_cat = Cat()
print(my_dog.speak())  # Output: Woof!
print(my_cat.speak())  # Output: Meow!
```

These concepts collectively form the foundation of OOP and are crucial for creating modular, maintainable, and reusable code. Each concept plays a specific role in modeling and organizing code in a way that mirrors real-world scenarios and relationships.

Defining classes, fields and methods, creating objects:

1. Defining a Class:
A class is a blueprint for creating objects. It defines the attributes (fields) and behaviors (methods) that the objects of the class will have.

```
class Person:
    # Fields (attributes)
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Methods (behaviors)
    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

In this example, the Person class has two fields (name and age) and one method (greet).

2. Creating Objects (Instances):

Once a class is defined, you can create objects (instances) of that class. Objects are concrete instances based on the blueprint provided by the class.

```
# Creating objects
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# Accessing fields
print(person1.name)  # Output: Alice
print(person2.age)   # Output: 30

# Calling methods
person1.greet()      # Output: Hello, my name is Alice and I am 25 years old.
person2.greet()      # Output: Hello, my name is Bob and I am 30 years old.
```

Here, person1 and person2 are instances of the Person class, each with its own set of attributes (name and age). The greet method can be called on each object to perform the specified behavior.

3. Class Inheritance:
Inheritance allows a new class to inherit the attributes and methods of an existing class.

```
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def study(self):
        print(f"{self.name} is studying.")
```

Here, the Student class inherits from the Person class, and it has an additional field (student_id) and a new method (study).

4. Creating Objects of Inherited Class:
You can create objects of the inherited class, which will have both the attributes and methods of the base class.

```
# Creating objects of the inherited class
student1 = Student("Eve", 20, "12345")

# Accessing fields from both base and derived classes
print(student1.name)        # Output: Eve
print(student1.student_id)  # Output: 12345

# Calling methods from both base and derived classes
student1.greet()            # Output: Hello, my name is Eve and I am 20 years old.
student1.study()            # Output: Eve is studying.
```

In this example, student1 is an object of the Student class, which inherits from the Person class. It can access fields and methods from both classes.

Accessing rules : public, private, protected, default:

Public:

Description: Public members are accessible from anywhere. They can be accessed from within the class, from outside the class, and even from other classes.
Keyword (example in Python): public (In many languages, including Python, members are public by default if no access modifier is specified).

```
class Example:
    def __init__(self):
        self.public_field = "I am public"

obj = Example()
print(obj.public_field)  # Accessing public field
```

Private:

Description: Private members are accessible only within the class that declares them. They are not accessible from outside the class or even from derived classes.
Keyword (example in Python): private (In Python, a single leading underscore _ is often used to indicate that a field or method is intended to be private).

```
    def __init__(self):
        self._private_field = "I am private"

obj = Example()
# Accessing private field directly would typically result in an error in some languages,
# but Python doesn't enforce strict access control.
print(obj._private_field)
```

Protected:

Description: Protected members are accessible within the class and by subclasses (derived classes). They are not accessible from outside the class hierarchy.
Keyword (example in Python): protected (In Python, a single leading underscore _ is sometimes used to indicate that a field or method is intended to be protected, but it's more of a convention).

```
class Example:
    def __init__(self):
        self._protected_field = "I am protected"

class DerivedExample(Example):
    def __init__(self):
        super().__init__()
        print(self._protected_field)  # Accessing protected field in a derived class
```

```
derived_obj = DerivedExample()
```
Default (Package-Private in Java):

Description: This is not a distinct keyword in many programming languages, but it refers to the default access level when no access modifier is specified. Members with default access are visible within the same package or module but not outside of it.

Example in Java:

```
// In Java, without an access modifier, it is package-private
class Example {
    String packagePrivateField = "I am package-private";
}
```
In this example, packagePrivateField is accessible within the same package, but not from outside.

It's important to note that the actual syntax and conventions for access modifiers may vary between programming languages, but the fundamental concepts remain similar. Access modifiers help enforce encapsulation, control access to class members, and contribute to the overall design and security of the code.

this keyword, static keyword, final keyword

this Keyword:

Definition: The this keyword refers to the current instance of the class. It is often used within class methods to refer to the instance variables of the class.
Usage:

```
class Example {
    private int number;

    public void setNumber(int number) {
        this.number = number; // 'this' refers to the instance variable 'number'
    }
}
```
In this example, the use of this helps distinguish between the instance variable number and the method parameter number.

2. static Keyword:

Definition: The static keyword is used to declare class-level members (variables and methods) that belong to the class rather than instances of the class. These members are shared among all instances of the class.

Usage:

```
class Example {
    static int staticVariable = 10;
    static void staticMethod() {
        // This method is a class-level method
    }
}
```

Here, staticVariable and staticMethod belong to the class itself, and they can be accessed using the class name (Example.staticVariable) rather than through an instance.

3. final Keyword:

Definition:

For Variables: When applied to a variable, the final keyword indicates that the variable's value cannot be changed after initialization, making it a constant.
For Methods: When applied to a method, it indicates that the method cannot be overridden by subclasses.
For Classes: When applied to a class, it indicates that the class cannot be subclassed.

Usage:

```
class Example {
    final int constantVariable = 5;

    final void finalMethod() {
        // This method cannot be overridden
    }
}

final class FinalClass {
    // This class cannot be subclassed
}
```

The final keyword ensures immutability for variables, prevents method overriding, and restricts class inheritance, depending on where it is applied.

These keywords are fundamental in object-oriented programming and help in controlling and managing the behavior of classes and objects. The usage may vary slightly between programming languages, but the core concepts remain consistent.

Constructors: Default constructors, Parameterized constructors, Copy constructors, Passing object as a parameter

Constructors are special methods in object-oriented programming that are used for initializing objects. They are called when an object of a class is created. There are different types of constructors, each serving a specific purpose.

1. Default Constructor:
Definition: A default constructor is a constructor with no parameters. If a class doesn't have any constructor defined, the compiler automatically provides a default

constructor. It initializes the object with default values (e.g., numeric types with 0, reference types with null).

Example:

```
class Example {
   // Default constructor (provided by the compiler)
}
```

2. Parameterized Constructor:
Definition: A parameterized constructor is a constructor with parameters. It allows you to initialize the object with specific values passed as arguments during object creation.

Example:

```
class Person {
   String name;
   int age;

   // Parameterized constructor
   public Person(String n, int a) {
      name = n;
      age = a;
   }
}

// Creating an object with a parameterized constructor
Person person1 = new Person("Alice", 25);
```

3. Copy Constructor:
Definition: A copy constructor is a constructor that creates a new object by copying the values from an existing object of the same class. It is useful for creating a duplicate or clone of an object.

Example:

```
class Person {
   String name;
   int age;

   // Copy constructor
   public Person(Person original) {
      name = original.name;
      age = original.age;
   }
}

// Creating an object using a copy constructor
Person person1 = new Person("Alice", 25);
Person person2 = new Person(person1); // Creates a copy of person1
```

4. Passing Object as a Parameter:
Definition: Constructors, like any other methods, can accept objects as parameters. This allows you to initialize an object using the values of another object.

Example:

```
class Rectangle {
    int length;
    int width;

    // Parameterized constructor with another object as a parameter
    public Rectangle(Rectangle other) {
        length = other.length;
        width = other.width;
    }
}

// Creating an object and passing another object as a parameter
Rectangle rectangle1 = new Rectangle();
rectangle1.length = 5;
rectangle1.width = 3;

Rectangle rectangle2 = new Rectangle(rectangle1); // Creates a new object with the
values of rectangle1
```

Constructors play a crucial role in initializing objects with meaningful values, and their use depends on the specific requirements of the class and the desired behavior during object creation.

Method overloading, constructor overloading

Method overloading and constructor overloading are both techniques used in object-oriented programming to create multiple methods or constructors with the same name but different parameters. Let's explore each concept:

Method Overloading:
Definition:
Method overloading is a feature that allows a class to have multiple methods with the same name but different parameters or argument lists. These methods can perform similar or different tasks based on the parameters they receive.

Example:

```
class Calculator {
    // Method overloading
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b) {
        return a + b;
    }
}

// Usage
Calculator calc = new Calculator();
int sum1 = calc.add(5, 3);          // Calls the int version of add
double sum2 = calc.add(4.5, 3.2);  // Calls the double version of add
```

Key Points:

Method overloading improves code readability and reusability by providing multiple ways to call a method with different parameter types or numbers.
Overloaded methods must have different parameter lists (number, type, or order of parameters).

Constructor Overloading:
Definition:
Constructor overloading is a technique where a class can have multiple constructors with different parameter lists. This allows objects to be initialized in various ways depending on the parameters provided during object creation.

Example:

```
class Person {
    String name;
    int age;

    // Constructor overloading
    public Person() {
        name = "Unknown";
        age = 0;
    }

    public Person(String n, int a) {
        name = n;
        age = a;
    }
}

// Usage
Person person1 = new Person();            // Calls the default constructor
Person person2 = new Person("Alice", 25); // Calls the parameterized constructor
```

Key Points:

Constructor overloading provides flexibility in object initialization by allowing multiple ways to create objects.

Each constructor must have a unique parameter list to distinguish it from other constructors.

Similarities:

Both method overloading and constructor overloading involve creating multiple versions of methods or constructors with the same name.
They improve code flexibility, readability, and reusability by providing different ways to achieve similar tasks.

Method overloading and constructor overloading are powerful features of object-oriented programming languages like Java and C++, allowing developers to write cleaner and more expressive code.

Wrapper Classes, String Class and its methods: chatAt(), contains(), format(), length(), split():

Wrapper Classes:
In Java, primitive data types (int, char, float, etc.) are not objects, but sometimes it's necessary to treat them as objects. Wrapper classes provide a way to convert primitive data types into objects. Each primitive type has a corresponding wrapper class in Java.

Example:
```
int primitiveInt = 42;

// Using the Integer wrapper class
Integer wrappedInt = Integer.valueOf(primitiveInt);

// Autoboxing (automatic conversion from primitive to wrapper)
Integer autoBoxedInt = primitiveInt;
```

Some commonly used wrapper classes:

Integer for int
Double for double
Character for char
Boolean for boolean

String Class:
The String class in Java represents a sequence of characters. It is widely used for manipulating textual data and comes with various methods for performing operations on strings.

Common String Methods:
charAt(int index):

Returns the character at the specified index.

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

contains(CharSequence sequence):

Checks if the string contains the specified sequence of characters.

```
String str = "Hello";
boolean contains = str.contains("ell"); // Returns true
```

format(String format, Object... args):

Returns a formatted string using the specified format string and arguments.

```
String formatted = String.format("Hello, %s!", "World");
// Returns "Hello, World!"
```

length():

Returns the length (number of characters) of the string.

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

split(String regex):

Splits the string into an array of substrings based on the specified regular expression.

```
String sentence = "This is a sample sentence.";
String[] words = sentence.split("\\s+"); // Splits by whitespace
// Returns ["This", "is", "a", "sample", "sentence."]
```

These methods provide basic functionality for manipulating and working with strings in Java. The String class has many more methods for various string operations, making it a versatile and widely used class in Java programming.

User Input: Scanner class and Command Line Arguments

1. Scanner Class:
The Scanner class is part of the java.util package and is commonly used to read input from the user. It provides methods to read various data types, such as nextInt(), nextDouble(), and nextLine(). The Scanner class can be used to read input from the console or other input sources.

Example:

```
import java.util.Scanner;

public class UserInputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter an integer: ");
        int userInputInt = scanner.nextInt();

        System.out.print("Enter a double: ");
        double userInputDouble = scanner.nextDouble();

        scanner.nextLine(); // Consume the newline character left by nextDouble()

        System.out.print("Enter a string: ");
        String userInputString = scanner.nextLine();

        System.out.println("Entered values: " + userInputInt + ", " + userInputDouble + ",
" + userInputString);

        scanner.close(); // Close the scanner to release resources
    }
}
```

2. Command Line Arguments:
Command line arguments are parameters that are passed to a Java program when it is executed from the command line. The main method of the Java program can accept these arguments as an array of strings.

Example:

```
public class CommandLineArgumentsExample {
    public static void main(String[] args) {
        // args is an array containing command line arguments
        // It can be accessed based on the index (args[0], args[1], ...)

        if (args.length >= 2) {
            String arg1 = args[0];
            int arg2 = Integer.parseInt(args[1]);

            System.out.println("Argument 1: " + arg1);
            System.out.println("Argument 2: " + arg2);
        } else {
            System.out.println("Insufficient command line arguments.");
        }
    }
}
```

Running from the command line:

```
java CommandLineArgumentsExample argument1 42
```

In this example, "argument1" and "42" are passed as command line arguments.

Important Points:

The Scanner class is suitable for interactive user input from the console.

Command line arguments are useful when parameters need to be passed when launching a Java program.
Both approaches have their use cases, and the choice between them depends on the requirements of the specific program.

Unit– 3: Inheritance, Packages & Interfaces

Basics of Inheritance, Types of inheritance: single, multiple, multilevel, hierarchical and hybrid inheritance.

Basics of Inheritance:
Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties and behaviors from another class. The class that is being inherited is called the superclass or base class, and the class that inherits from it is called the subclass or derived class. Inheritance promotes code reusability and establishes a relationship between classes.

Types of Inheritance:

1. Single Inheritance:
In single inheritance, a class inherits from only one superclass.

```
class A {
   // Class A members
}

class B extends A {
   // Class B inherits from class A
}
```

2. Multiple Inheritance:
Multiple inheritance allows a class to inherit from more than one superclass. While Java does not support multiple inheritance for classes, it does support it for interfaces through interface inheritance.

```
interface A {
   // Interface A members
}

interface B {
   // Interface B members
}

class C implements A, B {
   // Class C implements both interfaces A and B
}
```

3. Multilevel Inheritance:
In multilevel inheritance, a class inherits from another class, and then another class inherits from that derived class.

```
class A {
   // Class A members
}

class B extends A {
   // Class B inherits from class A
}

class C extends B {
   // Class C inherits from class B
}
```

4. Hierarchical Inheritance:
Hierarchical inheritance involves multiple classes inheriting from a single superclass.

```
class A {
   // Class A members
}

class B extends A {
   // Class B inherits from class A
}

class C extends A {
   // Class C also inherits from class A
}
```

5. Hybrid Inheritance:
Hybrid inheritance is a combination of two or more types of inheritance within a program.

```
class A {
   // Class A members
}

class B extends A {
   // Class B inherits from class A
}

interface X {
   // Interface X members
}

class C extends B implements X {
   // Class C inherits from class B and implements interface X
}
```

Key Points:

Inheritance allows the reuse of code and the creation of a hierarchy of classes.
The keyword extends is used in Java to establish an inheritance relationship between classes.

Multiple inheritance is not directly supported for classes in Java due to the "diamond problem" (ambiguity when a class inherits from two classes with the same method signature).

Interfaces in Java provide a form of multiple inheritance, allowing a class to implement multiple interfaces.
The relationships between classes in inheritance can be visualized as a hierarchy or tree structure.
Inheritance is a powerful mechanism in OOP that helps create a modular and organized code structure, facilitating code reuse and maintainability. The different types of inheritance provide flexibility in designing class relationships based on the requirements of a specific program or application.

Method overriding, Object class and overriding its methods : equals(), toString(), finalize(), hashCode()

Method Overriding:
Method overriding is a feature in object-oriented programming that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. The overridden method in the subclass should have the same signature (name, return type, and parameters) as the method in the superclass. This allows a more specialized behavior to be implemented in the subclass.

Example of Method Overriding:

```
class Animal {
   void makeSound() {
      System.out.println("Generic animal sound");
   }
}

class Dog extends Animal {
   @Override
   void makeSound() {
      System.out.println("Woof!");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog myDog = new Dog();
      myDog.makeSound();  // Calls the overridden method in Dog class
   }
}
```
Object Class and Overriding Its Methods:
In Java, every class is implicitly a subclass of the Object class. The Object class provides several methods that can be overridden in subclasses to customize their behavior.

1. equals() Method:

The equals() method is used to compare the content of two objects for equality. The default implementation in the Object class compares object references, but it is often overridden in custom classes for meaningful content comparison.

```
@Override
public boolean equals(Object obj) {
   // Custom implementation for comparing objects
}
```

2. toString() Method:
The toString() method returns a string representation of an object. By default, it returns the class name followed by the "@" symbol and the object's hashcode. It is commonly overridden to provide a more meaningful string representation.

```
@Override
public String toString() {
   // Custom implementation for creating a string representation
}
```

3. hashCode() Method:
The hashCode() method returns a hash code value for an object. The default implementation in the Object class returns the object's memory address. It is often overridden to provide a more suitable hash code based on the object's content.

```
@Override
public int hashCode() {
   // Custom implementation for generating a hash code
}
```

4. finalize() Method:
The finalize() method is called by the garbage collector before an object is reclaimed by memory management. It can be overridden to include custom cleanup code, but its usage is discouraged due to the unpredictability of when it will be called.

```
@Override
protected void finalize() throws Throwable {
   // Custom cleanup code
   super.finalize();
}
```

Key Points:

Method overriding allows customization of behavior in subclasses.
The Object class is the root class for all Java classes.
Overriding methods like equals(), toString(), hashCode(), and finalize() in custom classes provides control over their behavior.
By understanding and appropriately using method overriding and overriding methods from the Object class, developers can create more flexible and meaningful classes in their Java applications.

Defining interface, implementing interface, multiple inheritance using interface

Defining Interface:
An interface in Java is a collection of abstract methods (methods without a body) and constant declarations. It defines a contract for classes that implement it, specifying a set of methods that the implementing classes must provide. In addition to abstract methods, interfaces can also include default methods (methods with a default implementation) and static methods.

Example of Defining Interface:

```
// Defining an interface named "Drawable"
interface Drawable {
   void draw(); // Abstract method
   void resize(); // Another abstract method
   default void info() {
      System.out.println("This is a drawable object.");
   }
}
```

Implementing Interface:
To implement an interface in a class, the implements keyword is used. The implementing class must provide concrete implementations for all the abstract methods declared in the interface. It can also override default methods if needed.

Example of Implementing Interface:

```
// Implementing the Drawable interface in a class named "Circle"
class Circle implements Drawable {
   @Override
   public void draw() {
      System.out.println("Drawing a circle");
   }

   @Override
   public void resize() {
      System.out.println("Resizing the circle");
   }
}
```

Multiple Inheritance using Interface:
Java supports multiple inheritance through interfaces. A class can implement multiple interfaces, allowing it to inherit the abstract methods from each interface. This is a way to achieve multiple inheritance in Java without the issues associated with multiple inheritance of classes (diamond problem).

Example of Multiple Inheritance using Interface:

```
// Another interface named "Resizable"
interface Resizable {
   void resize();
}

// Implementing both "Drawable" and "Resizable" interfaces in a class named
"ResizableCircle"
class ResizableCircle implements Drawable, Resizable {
   @Override
   public void draw() {
      System.out.println("Drawing a resizable circle");
   }

   @Override
   public void resize() {
      System.out.println("Resizing the resizable circle");
   }
}
```

Key Points:

An interface is a blueprint for a set of methods that classes can implement.
The implements keyword is used to indicate that a class is implementing an interface.
A class can implement multiple interfaces, allowing it to inherit from multiple sources without the complications of multiple inheritance of classes.
By using interfaces, Java enables the creation of flexible and modular code structures, promoting better code organization and reuse. Multiple inheritance through interfaces allows developers to design and implement classes with a wide range of behaviors.

Abstract class and final class

Abstract Class:
An abstract class in Java is a class that cannot be instantiated on its own and is meant to be subclassed by other classes. It may contain abstract methods (methods without a body) that must be implemented by its concrete subclasses. Abstract classes can also have concrete methods with a defined implementation. Abstract classes provide a way to define a common interface for a group of related classes while allowing some level of implementation to be deferred to the subclasses.

Example of Abstract Class:
```
// Abstract class named "Shape"
abstract class Shape {
   // Abstract method
   abstract void draw();

   // Concrete method
   void info() {
      System.out.println("This is a shape.");
   }
}
```

Final Class:
A final class in Java is a class that cannot be subclassed. It is marked with the final keyword, and once a class is declared as final, it cannot be extended by any other class. This is often done to prevent further modification or extension of a class, ensuring that its behavior remains unchanged. Final classes are commonly used for utility classes or classes that should not be altered or extended.

Example of Final Class:

```
// Final class named "Utility"
final class Utility {
   // Methods and properties of the utility class
}
```

Key Points:

Abstract Class:
An abstract class is declared using the abstract keyword.
Abstract classes can have both abstract and concrete methods.
Objects cannot be instantiated from an abstract class directly.
Subclasses must provide concrete implementations for abstract methods.

Final Class:
A final class is declared using the final keyword.
Final classes cannot be subclassed.
Once a class is declared as final, its methods cannot be overridden in subclasses.
Final classes are often used to create immutable classes or classes with fixed behavior.
When to Use Abstract and Final Classes:

Abstract Classes:

Use abstract classes when you want to provide a common interface for a group of related classes and enforce certain methods to be implemented by subclasses.
Abstract classes are suitable when you want to share code among multiple related classes.

Final Classes:

Use final classes when you want to prevent further modification or extension of a class.
Final classes are suitable for utility classes or classes with fixed behavior that should not be altered.
Both abstract and final classes serve different purposes and are used based on the design requirements of a program.