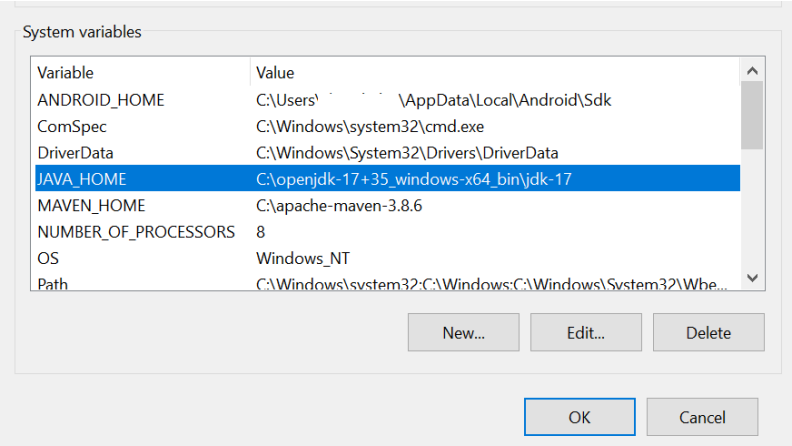
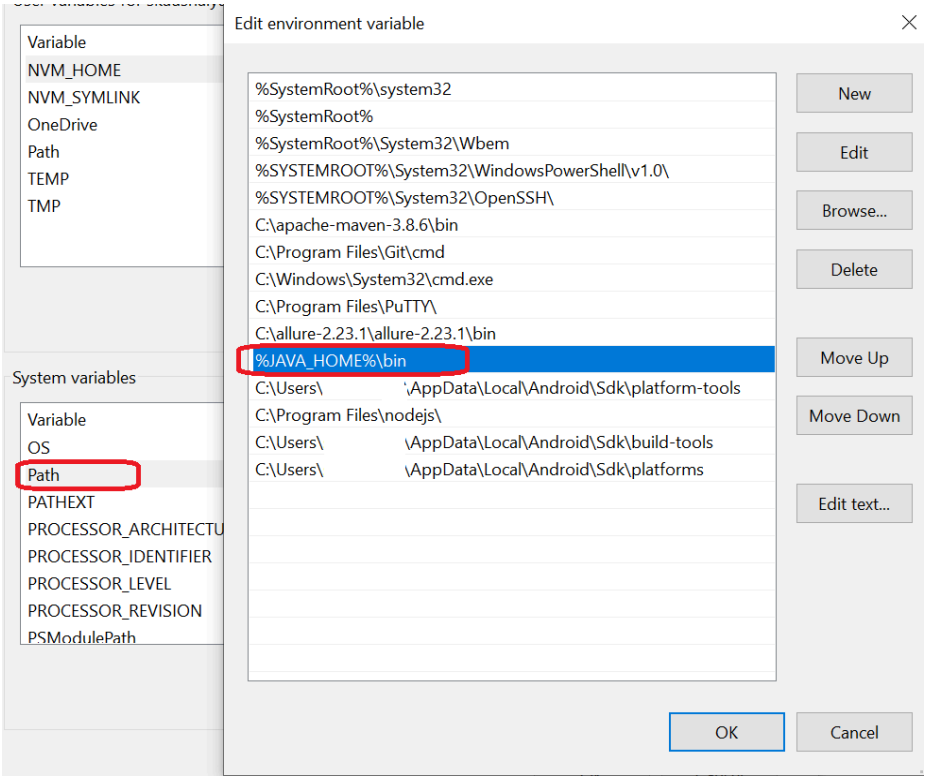
# Java and Maven Setup

- **Take java and setup the java path, and java home**

  You can take Open jdk from https://jdk.java.net/archive/ . You can use another open java development kit or oracle java.

  If you take open jdk from the above site, extract it somewhere in your machine and set up the java path and JAVA_HOME as follows.






Mac users can install openJDK as a brew install.

brew update
brew install openjdk@[version]

Further Mac users can download OpenJDK from https://jdk.java.net/archive/ according to your processor chip and then configure.

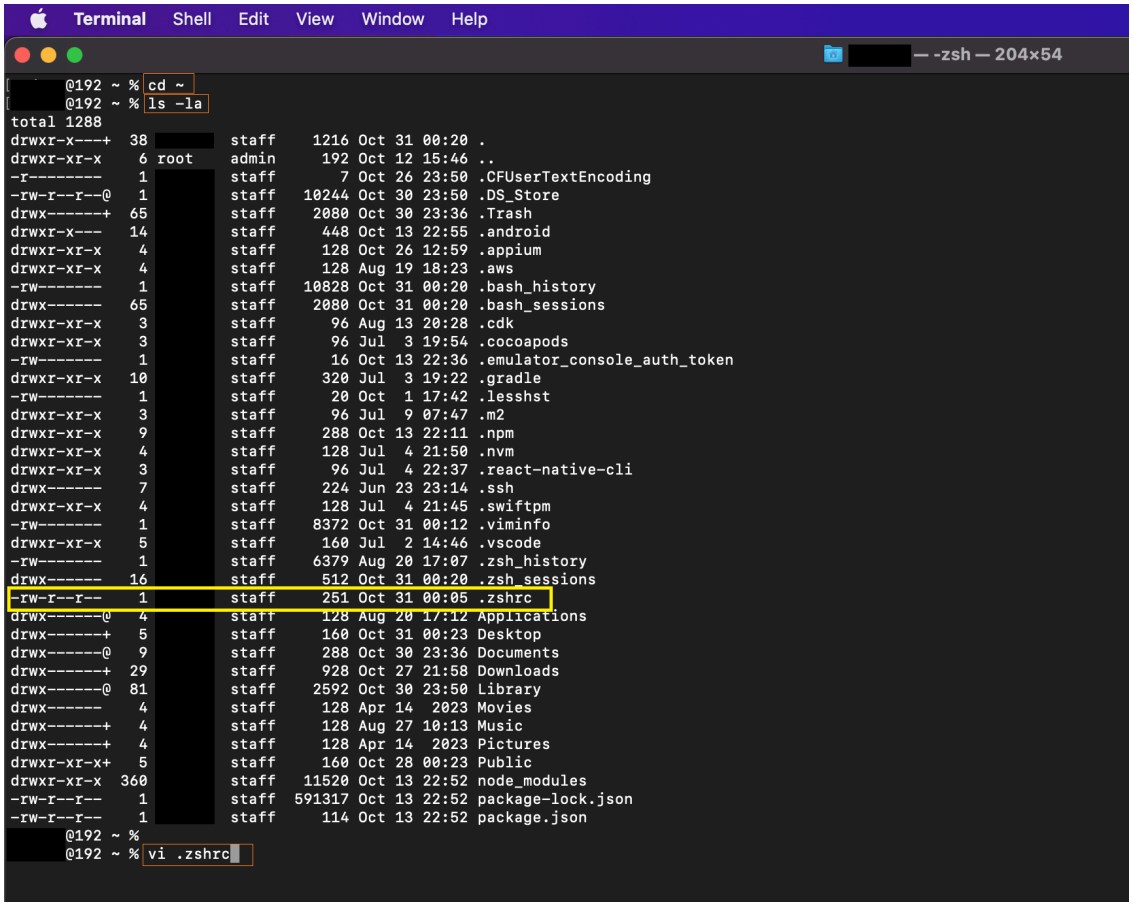
Mac users can set up the path as follows.

First you need to find the java installed directory. You need to open the finder window and then click “go-> go to folder” in the menu bar. You can find it under the “/Library/Java” folder.

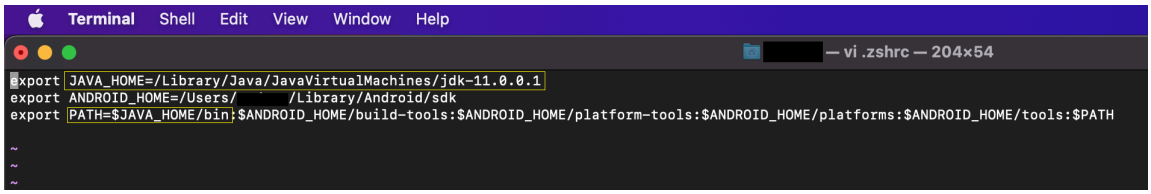
Then you can open the terminal and go to home using “*cd ~*”

List all the files using the “*ls -la*” command, and check whether the .zshrc file exists; or else create it using the “t*ouch .zshrc*” command.

Open the “.zshrc” file using your preferred editor. For this tutorial I used the “*vi .zshrc*” command to open.




Then you can set the paths as follows. Save and then exit the editor.




Once you have done, run “source ~/.zshrc” command to push the changes.

Once you have done the above steps, you can check the java version by typing “java --version” on cmd or terminal.

## Variables

In Java, variables can be classified into three main categories: **local variables, instance variables, and static variables.**

**Local variable :** Local variables are declared within a method or a block of code and have limited scope. They are only accessible within the method or block where they are declared.

```
public class LocalVariableExample {
    public static void main(String[] args) {
        int localVar = 10; // This is a local variable
        System.out.println("Local Variable: " + localVar);
    }
}
```

**Instance variable :** Instance variables are declared within a class but outside of any method, constructor, or block. They are associated with instances (objects) of the class and have a separate copy for each instance.

```
public class InstanceVariableExample {
    int instanceVar = 20; // This is an instance variable

    public static void main(String[] args) {
        InstanceVariableExample obj1 = new InstanceVariableExample();
        InstanceVariableExample obj2 = new InstanceVariableExample();

        System.out.println("Instance Variable in obj1: " + obj1.instanceVar);
        System.out.println("Instance Variable in obj2: " + obj2.instanceVar);
    }
}
```

**Static variable :** Static variables are declared using the static keyword and are shared among all instances of the class. They are associated with the class itself rather than with instances.

```
public class StaticVariableExample {
    static int staticVar = 30; // This is a static variable

    public static void main(String[] args) {
        StaticVariableExample obj1 = new StaticVariableExample();
        StaticVariableExample obj2 = new StaticVariableExample();

        System.out.println("Static Variable in obj1: " + obj1.staticVar);
        System.out.println("Static Variable in obj2: " + obj2.staticVar);

        // You can also access a static variable using the class name
        System.out.println("Static Variable using class name: " +
StaticVariableExample.staticVar);

        // You can also modify the static variable using any object
        obj1.staticVar = 40;
        System.out.println("Modified Static Variable in obj2: " +
obj2.staticVar);
    }
}
```

# Data types

In Java, data types can be categorized into two main groups: **primitive data types and non-primitive data types** (also known as reference data types).

**Primitive :** Primitive data types represent simple, single values and are predefined in the Java language.
They are also called value types because they directly store the actual data value.
Examples of primitive data types include int, double, char, boolean, byte, short, long, and float.

**Non-primitive :** Non-primitive data types represent more complex data structures and are not predefined in the Java language.
They are also called reference types because they store references (memory addresses) to objects in memory rather than the actual data.
Examples of non-primitive data types include classes, interfaces, arrays, and enumerated types.

**Points :::** Primitive data types have fixed sizes, which are platform-dependent but well-defined. Non-primitive data types do not have fixed sizes because they depend on the content and structure of the objects they reference.
Primitive data types have default values even if you don't explicitly initialize them. Non-primitive data types have a default value of null if not explicitly initialized.
Primitive data types support simple arithmetic and logical operations directly. Non-primitive data types represent objects, and operations on them often involve method calls and interactions with the object's behavior.
Primitive data types are passed by value when used as method parameters. Non-primitive data types are passed by reference when used as method parameters.

**Ranges in primitive data types :**
In Java, the size of primitive data types is platform-dependent, meaning it can vary depending on the architecture and the Java Virtual Machine (JVM) implementation being used. However, Java's language specification defines minimum size requirements for each primitive data type, and these are generally followed by most JVM implementations.

*byte:*
Size: 8 bits (1 byte)
Range: -128 to 127
byte myByte = 100; // Valid
byte invalidByte = 128; // Invalid, exceeds the maximum value (should be in the range -128 to 127)

*short:*
Size: 16 bits (2 bytes)
Range: -32,768 to 32,767

short myShort = 20000; // Valid
short invalidShort = 35000; // Invalid, exceeds the maximum value (should be in the range -32,768 to 32,767)

*int:*
Size: 32 bits (4 bytes)
Range: -2,147,483,648 to 2,147,483,647
int myInt = 1000000; // Valid
int invalidInt = 2147483648; // Invalid, exceeds the maximum value (should be in the range -2,147,483,648 to 2,147,483,647)

*long:*
Size: 64 bits (8 bytes)
Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
long myLong = 10000000000L; // Valid, note the 'L' suffix to indicate a long literal
long invalidLong = 92233720368547758O8L; // Invalid, exceeds the maximum value (should be in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

*float:*
Size: 32 bits (4 bytes)
Range: Approximately ±3.4E-38 to ±3.4E+38
float myFloat = 3.14159f; // Valid, note the 'f' suffix to indicate a float literal
float invalidFloat = 3.5E+39f; // Invalid, exceeds the maximum value (should be within the specified range)

*double:*
Size: 64 bits (8 bytes)
Range: Approximately ±1.7E-308 to ±1.7E+308
double myDouble = 3.14159265359; // Valid
double invalidDouble = 2E+309; // Invalid, exceeds the maximum value (should be within the specified range)

*char:*
Size: 16 bits (2 bytes)
Range: Represents a single Unicode character
char myChar = 'A'; // Valid
char invalidChar = 'AB'; // Invalid, can only represent a single character (should be a single character enclosed in single quotes)

*boolean:*
Size: Not strictly defined, typically JVM-specific
Represents a binary value - true or false
boolean isTrue = true; // Valid
boolean invalidBoolean = 42; // Invalid, can only be assigned 'true' or 'false'

# Condition

**If :** The if statement is used to execute a block of code only if a specified condition is true.
It does not have an else part, so if the condition is false, nothing specific happens.

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

**If-else :** The if-else statement is used to execute one block of code if a condition is true and another block of code if the condition is false.
It provides an alternative action to take when the condition is false.

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

**If-else if-else :** The if-else if-else statement allows you to test multiple conditions one after the other and execute different blocks of code based on the first true condition encountered.
It provides a series of alternatives and a final else block for any case that doesn't match the previous conditions.

```
if (condition1) {
    // Code to execute if condition1 is true
} else if (condition2) {
    // Code to execute if condition2 is true
} else if (condition3) {
    // Code to execute if condition3 is true
} else {
    // Code to execute if none of the conditions are true
}
```

**Nested if :** A nested if statement is an if statement inside another if or else block.
It is used to check multiple conditions sequentially, and the inner if statement is only executed if the outer if condition is true.
This allows for more complex conditional logic.

```
if (condition1) {
    // Code to execute if condition1 is true
    if (condition2) {
        // Code to execute if condition2 is true
```

```
    }
}
```

**Switch :** In Java, the switch statement is a control flow construct that allows you to select one of several code blocks to execute based on the value of an expression. It provides an alternative to using multiple if-else if statements when you have a series of conditions to check.

```
switch (expression) {
    case value1:
        // Code to execute if expression matches value1
        break; // Optional - used to exit the switch block
    case value2:
        // Code to execute if expression matches value2
        break;
    // More case statements as needed
    default:
        // Code to execute if none of the cases match expression
}
```

**Example :**

```
int day = 3;
String dayName;

switch (day) {
    case 1:
        dayName = "Sunday";
        break;
    case 2:
        dayName = "Monday";
        break;
    /
}

System.out.println("Day is " + dayName);
```

# Loops

**For loop**

In Java, the for loop and the enhanced for loop (also known as the "for-each" loop) are used for iterating through collections, arrays, or sequences of elements. They serve different purposes and have distinct syntax.

The traditional for loop is used when you know the number of iterations you want to perform.

It consists of three parts: initialization, condition, and increment/decrement.

```
for (int i = 0; i < 5; i++) {
    System.out.println("Iteration " + i);
}
```

The enhanced for loop is used for iterating through collections, arrays, or any object that implements the Iterable interface.
It simplifies iteration by abstracting away the initialization, condition, and increment/decrement.
It works well when you don't need the index of the elements but just want to iterate through them.

```
int[] numbers = {1, 2, 3, 4, 5};

for (int num : numbers) {
    System.out.println(num);
}

List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
names.add("Charlie");

for (String name : names) {
    System.out.println(name);
}
```

**while**
In Java, the while loop and the do-while loop are used for repetitive execution of a block of code as long as a specified condition is true. They differ in when the condition is evaluated relative to the execution of the loop body.

**The while loop** is used when you want to execute a block of code as long as a certain condition remains true.
It consists of a condition, and the loop body is executed repeatedly as long as the condition evaluates to true.
The condition is evaluated before the loop body is executed.

```
int count = 0;
while (count < 5) {
    System.out.println("Count is: " + count);
    count++;
}
```

**The do-while loop** is used when you want to execute a block of code at least once, and then continue executing it as long as a specified condition remains true.
It consists of a condition, and the loop body is executed first before the condition is evaluated. This guarantees that the loop body is executed at least once, even if the condition is initially false.

```
int count = 0;
do {
    System.out.println("Count is: " + count);
    count++;
} while (count < 5);
```

# Break and continue

**Break**
The break statement is used to exit or terminate a loop prematurely. When encountered inside a loop, it immediately terminates the loop, and control is transferred to the code following the loop.
It can be used in both for and while loops.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Exit the loop when i is equal to 5
    }
    System.out.println(i);
}
```

**continue**
The continue statement is used to skip the current iteration of a loop and continue with the next iteration. When encountered inside a loop, it skips the remaining code in the current iteration and jumps to the next iteration.
It can be used in both for and while loops.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // Skip the current iteration when i is equal to 5
    }
    System.out.println(i);
}
```

# OOP(Object oriented programming)

**Class**

A class is a blueprint or a template for creating objects. It defines the structure and behavior that objects of that class will have.
In a class, you can define fields (also known as member variables or instance variables) to store data, and methods to define the behavior or actions that objects can perform.
Classes are defined using the class keyword in Java.

```
public class Person {
    // Fields (instance variables)
    String name;
    int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method
    public void sayHello() {
        System.out.println("Hello, my name is " + name + " and I am " + age + "
years old.");
    }
}
```

**Object**

An object is an instance of a class. It is a concrete, tangible entity that can be created based on the blueprint provided by the class.
Objects have state (represented by the values of their fields) and behavior (represented by the methods defined in the class).
You create objects from a class using the new keyword in Java.

```
public class Main {
    public static void main(String[] args) {
        // Creating objects of the Person class
        Person person1 = new Person("Alice", 30);
        Person person2 = new Person("Bob", 25);

        // Using object methods
        person1.sayHello(); // Output: Hello, my name is Alice and I am 30 years
old.
        person2.sayHello(); // Output: Hello, my name is Bob and I am 25 years
old.
    }
}
```

**Encapsulation**

Encapsulation is one of the four fundamental concepts in object-oriented programming (OOP), the others being inheritance, polymorphism, and abstraction. It refers to the practice of bundling the data (attributes or fields) and methods (functions or behaviors) that operate on the data into a single unit, known as a class. Access to the data is restricted to the methods defined within the class. This concept helps in hiding the internal implementation details of a class and exposing only the necessary functionalities, promoting data security and code maintainability.

In Java, encapsulation is achieved through the following mechanisms

**Access Modifiers**

***private: Members are only accessible within the same class.***

```
public class MyClass {
    private int privateField;

    private void privateMethod() {
        // Implementation...
    }

    public void accessPrivateMembers() {
        privateField = 42;        // Valid, accessed from within the same class
        privateMethod();          // Valid, called from within the same class
    }
}
```

***protected: Members are accessible within the same class and its subclasses.***

```
public class MyBaseClass {
    protected int protectedField;

    protected void protectedMethod() {
        // Implementation...
    }
}

public class MySubClass extends MyBaseClass {
    public void accessProtectedMembers() {
        protectedField = 42;       // Valid, accessed from a subclass
        protectedMethod();         // Valid, called from a subclass
    }
}
```

***default (no modifier): Members are accessible within the same package.***

```
class PackagePrivateClass {
    int packagePrivateField;
```

```
    void packagePrivateMethod() {
        // Implementation...
    }
}

public class OtherClassInSamePackage {
    public void accessPackagePrivateMembers() {
        PackagePrivateClass obj = new PackagePrivateClass();
        obj.packagePrivateField = 42;   // Valid, accessed from the same package
        obj.packagePrivateMethod();     // Valid, called from the same package
    }
}
```

***public: Members are accessible from anywhere.***

```
public class PublicClass {
    public int publicField;

    public void publicMethod() {
        // Implementation...
    }
}

public class AnotherClassInDifferentPackage {
    public void accessPublicMembers() {
        PublicClass obj = new PublicClass();
        obj.publicField = 42;   // Valid, accessed from anywhere
        obj.publicMethod();     // Valid, called from anywhere
    }
}
```

**Private Fields and Getter/Setter Methods**

To encapsulate data, class fields (instance variables) are often declared as private to restrict direct access.
Getter methods (accessor methods) are used to retrieve the value of a private field.
Setter methods (mutator methods) are used to modify the value of a private field while enforcing validation or other logic.

```
public class Student {
    private String name; // Private field

    // Getter method
    public String getName() {
        return name;
    }
```

```
    // Setter method
    public void setName(String name) {
        // Additional validation logic can be applied here
        this.name = name;
    }
}
```

**Constructor Initialization**

Constructors can be used to initialize the object's state (private fields) during object creation.

This allows you to ensure that the object starts with valid data.

```
public class Person {
    private String name;

    // Constructor for initializing name
    public Person(String name) {
        this.name = name;
    }

    // Other methods and encapsulated fields...
}
```

**Package-private (Default) Access**

If no access modifier is specified (i.e., no private, protected, or public), the member has package-private access.

Members with package-private access are accessible within the same package, which can be useful for encapsulation within a specific package.

```
class MyClass {
    // Package-private field and methods
    int data;

    void doSomething() {
        // Implementation...
    }
}
```

**Inheritance**

Inheritance is one of the fundamental concepts in object-oriented programming (OOP) that allows you to create a new class (subclass or derived class) based on an existing class (superclass or base class). Inheritance enables code reuse and the creation of a hierarchy of classes with shared attributes and behaviors. In Java, you can implement inheritance using the extends keyword.

```
public class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }
}

public class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    public void bark() {
        System.out.println(name + " is barking.");
    }
}
```

**has-a(aggregation)**

In object-oriented programming, the "has-a" relationship, also known as aggregation, is a type of association between classes where one class contains an instance of another class as a member. This means that one class has a reference to another class within it.

```
class Address {
    private String street;
    private String city;

    public Address(String street, String city) {
        this.street = street;
        this.city = city;
    }

    public String getStreet() {
        return street;
```

```
    }

    public String getCity() {
        return city;
    }
}

public class Person {
    private String name;
    private Address address;

    public Person(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public Address getAddress() {
        return address;
    }
}
```

**Polymorphism**

Polymorphism is one of the four fundamental principles of object-oriented programming (OOP), along with encapsulation, inheritance, and abstraction. In Java, polymorphism refers to the ability of different classes to be treated as instances of their common superclass. This allows you to write code that works with objects of multiple classes in a unified way. Polymorphism is achieved through method overriding and method overloading.

***Compile-Time Polymorphism (Static Binding):*** Also known as method overloading, compile-time polymorphism occurs when there are multiple methods in the same class with the same name but different parameter lists (number or types of parameters).
The compiler determines which method to call based on the method's signature (name and parameter types) at compile time.

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

    public double add(double a, double b) {
        return a + b;
    }
```

```
}
```

***Run-Time Polymorphism (Dynamic Binding):*** Also known as method overriding, run-time polymorphism occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
The method to be executed is determined at runtime based on the actual object type (subclass) that invokes the method.

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

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

**Abstraction**

Abstraction is the process of hiding the implementation details of an object and exposing only the relevant features or behavior to the users.
It focuses on "what" an object does rather than "how" it does it.

In Java, abstraction is achieved through abstract classes and methods.
An abstract class is a class that cannot be instantiated on its own. It serves as a blueprint for other classes that extend it.
An abstract method is a method that is declared in an abstract class but does not have an implementation in the abstract class. Subclasses must provide the implementation.

```
abstract class Shape {
   abstract void draw(); // Abstract method with no implementation
}

class Circle extends Shape {
   @Override
   void draw() {
      System.out.println("Drawing a circle.");
   }
}
```

```
class Rectangle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
```

**Interface**

An interface is a reference type in Java that defines a contract of methods that a class implementing the interface must provide.
It specifies "what" methods a class should have, but not "how" they should be implemented.

An interface is declared using the interface keyword in Java.
Classes implement an interface using the implements keyword.

```
interface Shape {
    void draw(); // Method signature without implementation
}

class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a circle.");
    }
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a rectangle.");
    }
}
```

**Constructor**

In Java, a constructor is a special type of method that is used to initialize objects. Constructors are called automatically when an object of a class is created, and they are responsible for setting up the initial state of the object by initializing its fields and performing any other necessary setup.

Here are some key points about constructors in Java:

***Constructor Signature:***

Constructors have the same name as the class they belong to.
Constructors do not have a return type, not even void.

***Default Constructor:***
If a class does not explicitly define any constructors, Java provides a default constructor with no arguments.
The default constructor initializes fields to default values (e.g., numeric fields to 0, references to null).

***Parameterized Constructor:***
You can define one or more constructors with parameters to allow for custom initialization of objects.

***Overloaded Constructors:***
A class can have multiple constructors, as long as they have different parameter lists (overloaded constructors).
This allows objects to be created with various sets of initial values.

***Constructor Chaining:***
Constructors can call other constructors in the same class using the this() keyword (constructor chaining) to reuse code.
Constructors can also call constructors from the superclass using the super() keyword.

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

    // Default constructor
    public Person() {
        name = "Unknown";
        age = 0;
    }

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

    // Getter methods
    public String getName() {
        return name;
    }

    public int getAge() {
```

```
        return age;
    }
}
```

# Static

In Java, the static keyword is used to define class-level (or static) members, which are associated with the class itself rather than with instances (objects) of the class. Static members are shared among all instances of the class and can be accessed using the class name, without creating an object of the class.

***Static variables*** are shared among all instances of a class.
They are declared using the static keyword.
Static variables are also known as class variables because they belong to the class, not to individual objects.

```
public class MyClass {
    static int staticVar = 10;

    // Other members...
}
```

***Static methods*** belong to the class rather than to instances of the class.
They are declared using the static keyword.
Static methods can be called using the class name, without creating an object.

```
public class MathUtils {
    public static int add(int a, int b) {
        return a + b;
    }

    // Other methods...
}
```

***Static blocks*** are used for static initialization of a class.
They are executed only once when the class is loaded into memory, typically before any instances of the class are created.
Static blocks are useful for performing tasks like initializing static variables or setting up resources.

```
public class MyInitializer {
    static {
        // Static initialization code
```

```
        System.out.println("Static block executed.");
    }
}
```

# This

In Java, the “this” keyword is a reference to the current instance of a class. It can be used within instance methods to refer to the object on which the method was invoked. The primary purpose of the “this” keyword is to distinguish between instance variables and parameters or local variables with the same name.

Example 01

```
public class MyClass {
    private int value;

    public void setValue(int value) {
        // Use "this" to access the instance variable
        this.value = value;
    }
}
```

Example 02

```
public class MyClass {
    private int value;

    // Parameterized constructor
    public MyClass(int value) {
        this.value = value;
    }

    // Default constructor that calls the parameterized constructor
    public MyClass() {
        this(0); // Call the parameterized constructor
    }
}
```

Example 03

```
public class MyClass {
    private int value;

    public void doSomething() {
        AnotherClass.process(this);
    }
}
```

```
class AnotherClass {
    public static void process(MyClass obj) {
        // Process the MyClass object
    }
}
```

Example 04

```
public class MyClass {
    private int value;

    public MyClass setValue(int value) {
        this.value = value;
        return this; // Return the current object
    }
}
```