Data types:

Collection of information is called data.

There are two types of data types in java.

1.primitive   2. not primitive data types.

1> Primitive data types : it is already predefined in java.

Byte, Short, int, int, long, Float, Double, Boolean.

2> Non primitive data types :can created by programmer . They are

String, Classes, Arrays

Byte -> is a key word is used to declare a variable that can hold 8 bit data values.

Short -> is a key word is used to declare a variable that can hold 16 bit data values, it means it can hold byte data type.

Int -> is a key word is used to declare a variable that can hold 36 bit data values , it means it can hold "short" and "byte" data type.

Long -> is a key word is used to declare a variable that can hold 64 bit data values, it means it can hold "byte", "short", "int" data types.

Float -> is a key word in java used to declare a variable that can hold a 32 bit number and it could accept float point.

Char -> if we have to hold any single character then we can use this char data type.

Flow controls :

Def: control flow statements let you control the flow of the execution of the code in your program . In java programming language, you can control the flow of execution of the by placing the decision making, branching , looping and adding conditional blocks.

Control flow statements are :

1. Decision making statements.

1. If statement.

2.if-else statement.

3.if-else-if statement.

4.switch

2. looping statements

For loop

While loop

Do-while loop

3.branching statements

Break statements

Continue statements

Return statements

<conditional statement>

Def: In Java, the main conditional statements are if, else if, else, and the switch statement.

1. if Statement: The if statement is used to execute a block of code only if a specified condition is true. It has the following

syntax:

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

Example:

```
int x = 10;
if (x > 5) {
   System.out.println("x is greater than 5");
}
```

2. if-else Statement: The if-else statement allows you to specify two blocks of code—one to be executed if the condition is true, and another if the condition is false. It has the following

syntax:

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

Example:

```
int x = 3;
if (x > 5) {
   System.out.println("x is greater than 5");
} else {
   System.out.println("x is not greater than 5");
}
```

3. if-else if-else Statement: The if-else if-else statement allows you to specify multiple conditions and execute different blocks of code based on the first true condition. It has the following

syntax:

```
if (condition1) {
   // code to be executed if condition1 is true
} else if (condition2) {
   // code to be executed if condition2 is true
} else {
   // code to be executed if none of the conditions are true
}
```

Example:

```
int x = 7;
if (x > 10) {
   System.out.println("x is greater than 10");
} else if (x > 5) {
   System.out.println("x is greater than 5 but not 10");
} else {
   System.out.println("x is 5 or less");
}
```

4. switch Statement: The switch statement is used to select one of many code blocks to be executed. It evaluates an expression and compares it with the values of different case labels. It has the following syntax:

```
switch (expression) {
  case value1:
    // code to be executed if expression == value1
    break;
  case value2:
    // code to be executed if expression == value2
    break;
  // ... more cases ...
  default:
    // code to be executed if none of the cases match
}
```

Example:

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

< looping statements >

In Java, looping statements are used to execute a block of code repeatedly as long as a certain condition is true. There are three main types of looping statements in Java: for, while, and do-while.

1. for Loop: The for loop is used when the number of iterations is known in advance. It has the following syntax:

```
for (initialization; condition; update) {
   // code to be executed }
```

2. while Loop: The while loop is used when the number of iterations is not known in advance, and the loop continues as long as a specified condition is true. It has the following syntax:

```
while (condition) {
   // code to be executed
}
```

Example :

```
int i = 0;
while (i < 5) {
   System.out.println("Iteration: " + i);
   i++;
\\here the code will execute int 0 to 4}
```

3. do-while Loop: The do-while loop is similar to the while loop, but it ensures that the block of code is executed at least once, as the condition is checked after the execution of the loop. It has the following syntax:

```
do {
   // code to be executed
} while (condition);
```

Example :

```
int i = 0;
```

```
do {
   System.out.println("Iteration: " + i);
   i++;
} while (i < 5); \\ this loop execute the block code first and later checks the
condition
```

>Arrays

In Java, an array is a data structure that allows you to store multiple values of the same data type under a single variable name.

>object

In object-oriented programming (OOP), an object is an instance of a class. A class is a blueprint or template that defines the attributes (fields or properties) and behaviors (methods) that objects of that type will have. Objects are created from classes and represent real-world entities or concepts in a program.

>class

A class is a user-defined data type that defines a blueprint for objects. It encapsulates data (attributes or fields) and the methods (functions or behaviors) that operate on that data.

>instance variables

Instance variables, also known as fields or properties, are the attributes of an object. They represent the state of an object.

```
myCar.make = "Toyota";
myCar.model = "Camry";
myCar.year = 2022;
```

Method Declaration:

A method is declared within a class and includes the method's signature, return type, name, and parameters (if any). The signature includes the method name and the types of its parameters.

>Method declaration without parameters

```
public void printHello() {
   System.out.println("Hello, World!");
}
```

> Method declaration with parameters

```
public int add(int num1, int num2) {
   return num1 + num2;
}
```

Return Type:

The return type specifies the type of data that the method will return. If a method does not return any value, the return type is specified as void.

// Method with a return type of int

```
public int add(int num1, int num2) {
   return num1 + num2;
}
```

> Method with a return type of void (no return value)

```
public void printHello() {
   System.out.println("Hello, World!");
}
```

```
// Calling the printHello method
printHello();
```

Parameters:

Parameters are variables that are passed into a method. They allow methods to receive input values when called.

// Method with parameters

```
public int add(int num1, int num2) {
   return num1 + num2;
}
// Calling the add method with arguments
int sum = add(5, 3);
```

Method Invocation:

To execute a method, you invoke or call it. The method is called by its name, and if it has parameters, you pass values or variables as arguments.

>Method Overloading:

Method overloading allows a class to have multiple methods with the same name but different parameter lists (number or types of parameters). The compiler determines which method to call based on the method signature.

>method overloading

```
public int add(int num1, int num2) {
    return num1 + num2;
}

public double add(double num1, double num2) {
    return num1 + num2;
}
```

>Static Methods:

Static methods belong to the class rather than an instance of the class. They are called using the class name and can be invoked without creating an object.

\\Static method

```
public static void staticMethod() {
    System.out.println("This is a static method.");
}
```

// Calling the static method

```
MyClass.staticMethod();
```

>Void Methods:

A method with a return type of void does not return any value. It performs an action or task without producing a result.

// Void method

```
public void printHello() {
    System.out.println("Hello, World!");
}
```

>static block

The static block is executed when the class is loaded into the Java Virtual Machine (JVM) and is executed only once, regardless of the number of instances created or the number of times the class is referenced. It provides a way to perform static initialization before any other code in the class is executed.

Here is the syntax for a static block:

```
public class MyClass {
   // Static variable
   static int staticVariable;

   // Static block
   static {
      // Code for static initialization
      staticVariable = 42;
      System.out.println("Static block executed");
   }

   // Other members of the class
   // ...
```

>instance block

Instance blocks are also known as instance initializers, and they are used for performing initialization tasks for instance variables when an object of the class is created. Each time an object is instantiated, the instance block is executed before the constructor.

Here is the syntax for an instance block:

```
public class MyClass {
   // Instance variable
   int instanceVariable;

   // Instance block
   {
      // Code for instance initialization
```

```
        instanceVariable = 10;
        System.out.println("Instance block executed");
    }

    // Constructor
    public MyClass() {
        // Code for the constructor
        System.out.println("Constructor executed");
    }

    // Other members of the class
    // ...
}
```

>encapsulation

Def: encapsulation is hiding data behind the setter and getter methods it consists a private variables too

Getter Methods:

Getter methods, also known as accessor methods, are used to retrieve the values of private attributes. They provide read-only access to the attributes.

```
public class MyClass {
    private int myAttribute;

    // Getter method for myAttribute
    public int getMyAttribute() {
        return myAttribute;
    }
}
```

In this example, getMyAttribute() is a getter method that allows external code to retrieve the value of the private attribute myAttribute.

Setter Methods:

Setter methods, also known as mutator methods, are used to modify the values of private attributes. They provide a way to update the state of an object.

```
public class MyClass {
   private int myAttribute;

   // Setter method for myAttribute
   public void setMyAttribute(int newValue) {
      myAttribute = newValue;
   }
}
```

>constructor

A constructor is a special method in a class that is called when an object is created. It is used to initialize the object's state. And a constructor will starts with class name if we were not created then java will create default constructor in background.

```
public Car(String make, String model, int year) {
   this.make = make;
   this.model = model;
   this.year = year;
}

// Creating an object with a constructor
Car myCar = new Car("Toyota", "Camry", 2022);
```

>INHERITANCE

Inheritance is one of the four fundamental principles of object-oriented programming (OOP) and is a mechanism that allows a new class (subclass or derived class) to inherit properties and behaviors from an existing class (superclass or base class). Inheritance promotes code reuse and the creation of a hierarchical relationship among classes, where a subclass can extend or specialize the functionality of its superclass.

>singgle level inheritance

In other words, there is a direct parent-child relationship between two classes, and the subclass extends the functionality of a single superclass.

EXAMPLE:

```
// Superclass
class Animal {
   void eat() {
      System.out.println("Animal is eating");
   }
}

// Subclass (Single-level Inheritance)
class Dog extends Animal {
   void bark() {
      System.out.println("Dog is barking");
   }
}

// Main class
public class Main {
   public static void main(String[] args) {
      // Creating an object of the subclass
```

```
        Dog myDog = new Dog();

        // Accessing methods from the superclass
  myDog.eat();

        // Accessing methods from the subclass
        myDog.bark();
    }
}
```

>multi level inheritance

Multi-level inheritance is a type of inheritance in object-oriented programming where a class extends another class, and then another class extends the second class. This creates a chain or hierarchy of classes, with each class inheriting attributes and behaviors from its immediate parent class. The depth of the hierarchy can extend to multiple levels.

EXAMPLE:

```
// Superclass
class Animal {
    void eat() {
        System.out.println("Animal is eating");
    }
}

// Subclass 1 (inherits from Animal)
class Mammal extends Animal {
    void breathe() {
        System.out.println("Mammal is breathing");
    }
}

// Subclass 2 (inherits from Mammal)
class Dog extends Mammal {
    void bark() {
        System.out.println("Dog is barking");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        // Creating an object of the subclass
        Dog myDog = new Dog();

        // Accessing methods from the superclass and its ancestors
        myDog.eat();     // Inherited from Animal
        myDog.breathe();  // Inherited from Mammal
        myDog.bark();     // Specific to Dog
    }
}
```

>hierarchical inheritance

Hierarchical inheritance is a type of inheritance in object-oriented programming where a single base class (superclass) is extended by multiple derived classes (subclasses). In this scenario, each subclass inherits from the same superclass, creating a hierarchical structure or tree-like relationship.

EXAMPLE:

```
// Superclass
class Animal {
   void eat() {
      System.out.println("Animal is eating");
   }
}

// Subclass 1 (inherits from Animal)
class Cat extends Animal {
   void meow() {
      System.out.println("Cat is meowing");
   }
}

// Subclass 2 (inherits from Animal)
class Dog extends Animal {
   void bark() {
      System.out.println("Dog is barking");
   }
}

// Main class
public class Main {
   public static void main(String[] args) {
      // Creating objects of the subclasses
      Cat myCat = new Cat();
      Dog myDog = new Dog();

      // Accessing methods from the superclass and its subclasses
      myCat.eat();    // Inherited from Animal
      myCat.meow();   // Specific to Cat

      myDog.eat();    // Inherited from Animal
      myDog.bark();   // Specific to Dog
   }
}
```

>MULTIPLE INHERITANCE

Multiple inheritance is a feature in object-oriented programming that allows a class to inherit properties and behaviors from more than one class. In multiple inheritance, a class can have more than one direct superclass, and it inherits attributes and methods from all of them. While multiple inheritance can be a powerful tool for code reuse, it can also introduce complexities and challenges, such as the diamond problem.

EXAMPLE:

WITH CLASS:

```
// First superclass
class Animal {
   void eat() {
      System.out.println("Animal is eating");
```

```
    }
}

// Second superclass
class Mammal {
    void breathe() {
        System.out.println("Mammal is breathing");
    }
}

// Subclass (inherits from both Animal and Mammal)
class Dog extends Animal, Mammal {
    void bark() {
        System.out.println("Dog is barking");
    }
}
```

EXAMPLE:

WITH INTERFACE:

```
// First interface
interface Animal {
    void eat();
}

// Second interface
```

```
interface Mammal {
    void breathe();
}

// Class implementing both interfaces (achieves multiple inheritance)
class Dog implements Animal, Mammal {
    @Override
    public void eat() {
        System.out.println("Dog is eating");
    }

    @Override
    public void breathe() {
        System.out.println("Dog is breathing");
    }

    void bark() {
        System.out.println("Dog is barking");
    }
}
```

>HYBRID

Hybrid inheritance is a combination of two or more types of inheritance within a single program. It typically involves a mix of different types of inheritance, such as single inheritance, multiple inheritance (through interfaces), hierarchical inheritance, or multilevel inheritance. Hybrid inheritance is a way to take advantage of the benefits of various inheritance types to model relationships in a more flexible manner.

EXAMPLE:

```
// Superclass
class Animal {
   void eat() {
      System.out.println("Animal is eating");
   }
}

// Interface 1
interface Mammal {
   void breathe();
}

// Interface 2
interface Pet {
   void play();
}

// Subclass implementing both interfaces and inheriting from Animal
class Dog extends Animal implements Mammal, Pet {
    @Override
    public void breathe() {
        System.out.println("Dog is breathing");
    }

    @Override
    public void play() {
        System.out.println("Dog is playing");
    }

    void bark() {
        System.out.println("Dog is barking");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        // Creating an object of the subclass
        Dog myDog = new Dog();

        // Accessing methods from the superclass, interfaces, and the subclass
        myDog.eat();    // Inherited from Animal
        myDog.breathe(); // Implemented from Mammal
        myDog.play();    // Implemented from Pet
        myDog.bark();    // Specific to Dog
    }
}
```

>POLYMORPHYSM

DEF:

Polymorphism is one of the four fundamental principles of object-oriented programming (OOP) and refers to the ability of a single entity (such as a method, operator, or object) to take on multiple forms. There are two main types of polymorphism in Java: “compile-time (or static) polymorphism” and “runtime (or dynamic) polymorphism.”

Compile-Time Polymorphism (Method Overloading):

Compile-time polymorphism is also known as method overloading. It occurs when a class has multiple methods with the same name but different parameter lists (different types or numbers of parameters).

The compiler determines which method to call based on the number and types of arguments at compile time.

EXAMPLE:

```
public class Calculator {
    // Method with two int parameters
    public int add(int a, int b) {
        return a + b;
    }
    // Method with three double parameters

    public double add(double a, double b, double c) {

        return a + b + c;

    }

}
```

Runtime Polymorphism (Method Overriding):

DEF:

Runtime polymorphism is also known as method overriding. It occurs when a subclass provides a specific implementation for a method that is already defined in its superclass.

The decision about which method to call is made at runtime, based on the actual type of the object.

EXAMPLE:

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

// Subclass overriding the makeSound method
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}
```

//Polymorphism through Interfaces:

Interfaces in Java also support polymorphism. A class can implement multiple interfaces, and objects of that class can be treated as instances of those interfaces.

```
// Interface
interface Shape {
    void draw();
}

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

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

In this example, both Circle and Square implement the Shape interface, and objects of these classes can be treated as instances of the Shape interface.

>ACCESS MODIFIERS

Access modifiers in Java are keywords that determine the visibility or accessibility of classes, fields, methods, and other members within a Java program. These modifiers control the level of access that other classes or components have to the declared elements. There are four main access modifiers in Java:

public:

The public modifier allows unrestricted access to the associated class, method, or field. Members marked as public can be accessed from any other class in the same program or even from external programs.

Example: public class MyClass { ... }, public void myMethod() { ... }

private:

The private modifier restricts access to the associated class, method, or field to only within the same class. It provides the highest level of encapsulation, ensuring that the member is not accessible from outside the class.

Example: private int myField;, private void myMethod() { ... }

protected:

The protected modifier allows access to the associated class, method, or field within the same package and by subclasses, regardless of the package. It provides a level of access between public and private.

Example: protected int myField;, protected void myMethod() { ... }

default (Package-Private):

If no access modifier is specified (also known as package-private or default), the member is accessible only within the same package. It provides a level of access between public and private and is the default access level if no modifier is specified.

Example: int myField;, void myMethod() { ... }

>ABSTRACT

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 both abstract methods (methods without a body) and concrete methods (methods with an implementation). Abstract classes are typically used as base classes in inheritance hierarchies.

The abstract keyword is used to declare an abstract class.

Example:

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

   // Concrete method with an implementation
   void resize() {
      System.out.println("Resizing the shape");
   }
}
```

Abstract Method:

An abstract method is a method declared without a body in an abstract class. Abstract methods do not provide an implementation in the abstract class; instead, the responsibility to implement them is delegated to the concrete subclasses that extend the abstract class.

Abstract methods are declared using the abstract keyword and followed by a method signature (return type, method name, and parameter list) without a method body.

Example:

javaCopy code

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