1.

2.

3.89

4.Attributes, also known as fields or properties, represent the state of an object. They describe what the object is or has. For example, consider a "Car" object. Its attributes could include properties such as color, make, model, and year.

Methods, on the other hand, represent the actions or behaviors that an object can perform. They define what the object can do. Sticking with our "Car" example, methods could include behaviors such as "startEngine()", "accelerate()", "brake()", and "turnOnHeadlights()".

5.The class name of the attribute is not given.

6.Volume : 0
length of box : 0
width of box : 0
height of box : 0

7.Volume : 180
length of box : 12
width of box : 5
height of box : 3

8.default constructor
length of box : 2
width of box : 2
height of box : 2

9.Parameterized constructor
Volume : 60
Parameterized constructor
Volume : 180

10.Compile Error

11.Line4,5,6

12.In Java, a constructor is a special type of method that is automatically called when an object of a class is instantiated (i.e., created). Its primary purpose is to initialize the newly created object by setting initial values for its attributes or performing any necessary setup operations.

13.In Java, constructors and methods are both types of functions within a class, but they serve different purposes.

Constructors:
- Constructors are special methods used for initializing objects when they are created.
- They are called automatically when an object is instantiated using the `new` keyword.
- Constructors typically initialize the state of the object by setting initial values to its attributes.
- They do not have a return type, not even `void`.
- Constructors are named after the class and can be overloaded.
- Example: In a "Car" class, a constructor would be used to initialize attributes like make, model, and year when a new car object is created.

Methods:
- Methods represent actions or behaviors that an object can perform.

- They are invoked explicitly by calling their name followed by parentheses.
- Methods can have a return type (including `void` for methods that don't return anything).
- They can accept parameters to perform actions based on the provided inputs.
- Methods can be overloaded and overridden.
- Example: In a "Car" class, methods such as "startEngine()", "accelerate()", and "brake()" would represent the actions a car can perform.

In summary, constructors are used to initialize objects upon creation, while methods represent the behaviors or actions that objects can perform.

14.B

15.Line 2 and 3 compile error

16.C

17.A

18.100 101
0 0

19.Code : 3001

20.Code : 2001

21.B

22.A

23.Encapsulation is one of the fundamental principles of object-oriented programming (OOP) that involves bundling the data (attributes) and methods (behaviors) that operate on the data into a single unit called a class. This concept hides the internal state of an object from the outside world and only allows access to it through well-defined interfaces.

Here's a breakdown of encapsulation with examples:

1. **Data Hiding**: Encapsulation allows data hiding, meaning the internal state of an object is hidden from the outside world. Access to the data is controlled by providing public methods (getters and setters) to read and modify the data, rather than allowing direct access to the attribut

3. **Isolation of Behavior**: Encapsulation allows for the isolation of an object's behavior from its implementation details. Users of the class only need to know how to use the public methods; they don't need to know how those methods are implemented internally.

Encapsulation helps in creating modular, maintainable, and reusable code by keeping the internal implementation details hidden and providing well-defined interfaces for interaction with objects. It also enhances security by preventing unauthorized access to sensitive data.

24.The terms "tightly encapsulated" and "loosely encapsulated" describe the degree to which the internal details of a class are hidden or exposed, respectively. Here's a comparison with examples:

Tightly Encapsulated:

In a tightly encapsulated class:
- The internal state of the class, including its attributes and methods, is well-protected and hidden from outside interference.
- Access to the class's attributes is restricted to only through public methods, ensuring that the integrity of the object is maintained.

- Changes to the internal implementation details of the class do not affect the external code that uses the class.

Example:

In this example, the `balance` attribute is encapsulated and can only be accessed or modified through the `getBalance()`, `deposit()`, and `withdraw()` methods. The internal state of the `BankAccount` object is tightly controlled, ensuring data integrity and security.

Loosely Encapsulated:

In a loosely encapsulated class:
- The internal state of the class is more exposed, allowing direct access to its attributes from outside the class.
- Methods may not provide strict control over data manipulation, leading to potential issues such as data corruption or unintended modifications.

In this example, the `name` and `age` attributes are public, meaning they can be accessed and modified directly from outside the class. This approach lacks encapsulation because there's no control over how these attributes are accessed or modified, potentially leading to unintended consequences.

In summary, tightly encapsulated classes enforce strict control over access to their internal state through well-defined interfaces, promoting data integrity and security. On the other hand, loosely encapsulated classes expose their internal state more freely, which can lead to issues with data manipulation and maintenance.

25.D

26.
```
// Date.java
public class Date {
    private int year = 1970;
    private int month = 1;
    private int day = 1;

    public void printDate() {
        System.out.println(year + "-" + month + "-" + day);
    }

    public void setYear(int year) {
        this.year = year;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public void setDay(int day) {
        this.day = day;
    }

    public int getYear() {
        return year;
    }

    public int getMonth() {
        return month;
    }
```

```
    public int getDay() {
        return day;
    }
}
```

27.

28.B

29.E

30.
```
// Rectangle.java
public class Rectangle {
    private double length = 1.0;
    private double width = 1.0;

    // Constructor
    public Rectangle() {}

    // Getter and setter methods for length
    public double getLength() {
        return length;
    }

    public void setLength(double length) {
        if (isValidDimension(length)) {
            this.length = length;
        } else {
            System.out.println("Invalid length. Length must be a floating-point
number larger than 0.0 and less than 20.0.");
        }
    }

    // Getter and setter methods for width
    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        if (isValidDimension(width)) {
            this.width = width;
        } else {
            System.out.println("Invalid width. Width must be a floating-point
number larger than 0.0 and less than 20.0.");
        }
    }

    // Method to calculate perimeter
    public double calculatePerimeter() {
        return 2 * (length + width);
    }

    // Method to calculate area
    public double calculateArea() {
        return length * width;
    }

    // Helper method to validate dimensions
    private boolean isValidDimension(double dimension) {
        return dimension > 0.0 && dimension < 20.0;
    }
}
```

31.A

32.E

33.Line 5 and 8

34.Static variables and instance variables are both types of fields in Java classes, but they have different characteristics and usage:

1. **Static Variables:**
   - Static variables, also known as class variables, are shared among all instances of a class. There is only one copy of a static variable regardless of how many objects of the class are created.
   - Static variables are declared using the `static` keyword and typically initialized outside of any method, often at the class level.
   - They are accessed using the class name followed by the dot operator (`ClassName.variableName`), rather than through object references.
   - Static variables are initialized once when the class is loaded into memory and retain their values until the program exits.
   - They are useful for defining constants, counters, or properties shared by all instances of a class.
   - Example: `public static int totalCount;`

2. **Instance Variables:**
   - Instance variables, also known as non-static variables, belong to individual instances (objects) of a class. Each object has its own copy of instance variables.
   - Instance variables are declared without the `static` keyword and are initialized inside the class, typically within the class's constructor or directly at the point of declaration.
   - They are accessed using object references (`objectName.variableName`).
   - Instance variables are initialized each time an object of the class is created and exist as long as the object exists in memory.
   - They represent the state of individual objects and hold unique values for each object.
   - Example: `private int length;`

In summary, the main differences between static variables and instance variables lie in their scope, memory allocation, initialization, and usage. Static variables are shared across all instances of a class and retain their values throughout the program's execution, while instance variables are specific to individual objects and their values vary from one object to another.

35.A

36.

37.Instance Methods
Belong to Instances:

Instance methods are associated with an instance of a class (an object). They are called on specific instances of the class.
Access to Instance Data:

Instance methods can access and modify instance variables (attributes specific to an object). They can also call other instance methods.
Static Methods
Belong to the Class:

Static methods are associated with the class itself, not with any particular instance. They are called on the class rather than on instances of the class.
No Access to Instance Data:

Static methods do not have access to instance variables or instance methods. They can, however, access other static methods and static variables.

No Self Parameter:

Static methods do not take a self parameter because they are not called on an instance. In Python, static methods are defined using the @staticmethod decorator.

38.Box is loaded into memory

39.A box object is created..
A box object is created..

40.
```
public class Cylinder {
    private double length;
    private double radius;

    // Default constructor
    public Cylinder() {
        this.length = 0.0;
        this.radius = 0.0;
    }

    // Parameterized constructor
    public Cylinder(double length, double radius) {
        this.length = length;
        this.radius = radius;
    }

    // Getter for length
    public double getLength() {
        return length;
    }

    // Setter for length
    public void setLength(double length) {
        this.length = length;
    }

    // Getter for radius
    public double getRadius() {
        return radius;
    }

    // Setter for radius
    public void setRadius(double radius) {
        this.radius = radius;
    }

    // Method to calculate volume
    public double calculateVolume() {
        return Math.PI * Math.pow(radius, 2) * length;
    }

    // Method to calculate surface area
    public double calculateSurfaceArea() {
        return 2 * Math.PI * radius * (radius + length);
    }
}
```

41.Box is loaded into memory
A box object is created..
A box object is created..
A box object is created..

42.D

43.A

44.B

45.Call by Value: A copy of the actual parameterâ ™s value is passed. Changes inside the method do not affect the original variable. Java uses call by value for both primitive types and object references.
Call by Reference: A reference to the actual parameter is passed. Changes inside the method affect the original variable. Java can mimic this behavior with objects since object references are passed by value, but they refer to the same object.

46.Line 3 and 4

47.

48.
```
public class Employee {
    private String firstName;
    private String lastName;
    private double monthlySalary;

    // Constructor to initialize instance variables
    public Employee(String firstName, String lastName, double monthlySalary) {
        this.firstName = firstName;
        this.lastName = lastName;
        setMonthlySalary(monthlySalary);  // Using the setter to validate the
salary
    }

    // Getter for firstName
    public String getFirstName() {
        return firstName;
    }

    // Setter for firstName
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    // Getter for lastName
    public String getLastName() {
        return lastName;
    }

    // Setter for lastName
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    // Getter for monthlySalary
    public double getMonthlySalary() {
        return monthlySalary;
    }

    // Setter for monthlySalary
    public void setMonthlySalary(double monthlySalary) {
        if (monthlySalary > 0) {
            this.monthlySalary = monthlySalary;
        }
    }

    // Method to calculate yearly salary
```

```
    public double getYearlySalary() {
        return monthlySalary * 12;
    }

    // Method to give a raise
    public void giveRaise(double percentage) {
        if (percentage > 0) {
            monthlySalary += monthlySalary * percentage / 100;
        }
    }
}
```

49.code : 1001
code : 1001

50.

51.D

52.MyClass()
MyClass(int,int)
MyClass(int)

53.