Here are the Java programs for each of your requests, complete with their aim, algorithm, pseudocode, code, and output.

---

1. Implement Constructor in Java

Aim

The aim is to demonstrate the fundamental concept of a **constructor** in Java. A constructor is a special method used for initializing newly created objects. This program will show how to define and use a parameterized constructor to set initial values for an object's instance variables.

Algorithm

1. Define a class, for example, Car.
2. Declare instance variables within the Car class, such as String make and int year.
3. Create a public **constructor** named Car that matches the class name. This constructor will accept parameters (e.g., String carMake, int carYear) corresponding to the instance variables.
4. Inside the constructor, use the this keyword (this.make = carMake;) to assign the parameter values to the instance variables, distinguishing them from local parameters.
5. In the main method, create an object (e.g., myCar) of the Car class, passing the required values to the constructor during instantiation (new Car("Toyota", 2022)).
6. Print the values of the object's instance variables to verify that the constructor correctly initialized them.

Pseudocode

```
CLASS Car
    make: String
    year: Integer

    // Constructor
    CONSTRUCTOR(carMake, carYear)
        this.make = carMake
        this.year = carYear
    END CONSTRUCTOR

END CLASS

MAIN
    // Create a Car object using the constructor
    myCar = NEW Car("Toyota", 2022)

    // Print details of the created car
    PRINT "Car Make: " + myCar.make
    PRINT "Car Year: " + myCar.year
END MAIN
```

Code

Java

```java
class Car {
    // Instance variables
    String make;
    int year;

    // Constructor with parameters
    public Car(String carMake, int carYear) {
        System.out.println("Car constructor called!");
        this.make = carMake; // Initialize 'make'
        this.year = carYear; // Initialize 'year'
    }

    // Method to display car details
    public void displayCarDetails() {
        System.out.println("Make: " + make + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of Car class, which calls the constructor
        System.out.println("Creating myCar object...");
        Car myCar = new Car("Honda", 2023); // Constructor is invoked here

        System.out.println("\nAccessing myCar details:");
        myCar.displayCarDetails(); // Displaying details
    }
}
```

Output

```
Creating myCar object...
Car constructor called!

Accessing myCar details:
Make: Honda, Year: 2023
```

---

2. Implement Class Concept in Java

Aim

The aim is to illustrate the fundamental **class concept** in Java, which serves as a blueprint for creating objects. This program will define a class with attributes (instance variables) and behaviors (methods), then create an object from this class and interact with its attributes and methods.

Algorithm

1. Define a class, for example, Laptop.
2. Declare instance variables within the Laptop class to represent its attributes, such as String brand, String processor, and int ramGB.
3. Implement a constructor for the Laptop class to initialize these attributes when an object is created.
4. Define a public method (behavior), such as displaySpecs(), inside the Laptop class. This method will print the values of the instance variables.
5. In the main method, create an **object** (an instance) of the Laptop class using the new keyword and the constructor (e.g., new Laptop("Dell", "Intel i7", 16)).
6. Call the displaySpecs() method on the created object to demonstrate accessing its behaviors.

Pseudocode

```
CLASS Laptop
  brand: String
  processor: String
  ramGB: Integer

  // Constructor to initialize attributes
  CONSTRUCTOR(laptopBrand, laptopProcessor, laptopRamGB)
    this.brand = laptopBrand
    this.processor = laptopProcessor
    this.ramGB = laptopRamGB
  END CONSTRUCTOR
```

```
    // Method to display specifications
    METHOD displaySpecs()
        PRINT "Brand: " + brand
        PRINT "Processor: " + processor
        PRINT "RAM: " + ramGB + "GB"
    END METHOD

END CLASS

MAIN
    // Create a Laptop object
    myLaptop = NEW Laptop("HP", "AMD Ryzen 5", 8)

    // Call a method on the object
    PRINT "My Laptop Specifications:"
    myLaptop.displaySpecs()
END MAIN
```

Code

Java

```java
class Laptop {
    // Attributes (instance variables)
    String brand;
    String processor;
    int ramGB;

    // Constructor to initialize attributes
    public Laptop(String brand, String processor, int ramGB) {
        this.brand = brand;
        this.processor = processor;
        this.ramGB = ramGB;
        System.out.println("Laptop object created: " + brand);
    }
    // Behavior (method)
    public void displaySpecs() {
        System.out.println("  Brand: " + brand);
        System.out.println("  Processor: " + processor);
        System.out.println("  RAM: " + ramGB + "GB");
    }

    // Another behavior (method)
    public void powerOn() {
        System.out.println(brand + " laptop is powering on...");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object (instance) of the Laptop class
        Laptop gamingLaptop = new Laptop("Alienware", "Intel Core i9", 32);

        System.out.println("\n--- Interacting with Gaming Laptop ---");
        gamingLaptop.displaySpecs(); // Calling a method on the object
        gamingLaptop.powerOn();     // Calling another method
    }
}
```

Output

```
Laptop object created: Alienware

--- Interacting with Gaming Laptop ---
  Brand: Alienware
  Processor: Intel Core i9
  RAM: 32GB
Alienware laptop is powering on...
```

3. Implement Method Overloading Using Static Methods

Aim

The aim is to demonstrate **method overloading** using static methods within a single class. Method overloading allows multiple methods to have the same name, provided they have different parameter lists (number, type, or order of parameters). static methods belong to the class itself, not to specific objects.

Algorithm

1. Define a class, for example, Calculator.
2. Inside the Calculator class, create multiple **static methods** with the same name, e.g., add.
3. Each add method must have a **different signature** (different number or types of parameters).
    - static int add(int a, int b)
    - static double add(double a, double b)
    - static int add(int a, int b, int c)
4. In the main method, call these static add methods directly using the class name (e.g., Calculator.add(5, 10)), passing different types or numbers of arguments.
5. Observe how Java automatically selects the correct overloaded method based on the arguments provided during the call (this is **compile-time polymorphism**).

Pseudocode

```
CLASS Calculator
```

```
    // Static method to add two integers
    STATIC METHOD add(a: Integer, b: Integer)
        RETURN a + b
    END METHOD

    // Static method to add two doubles (overloaded)
    STATIC METHOD add(a: Double, b: Double)
        RETURN a + b
    END METHOD

    // Static method to add three integers (overloaded)
    STATIC METHOD add(a: Integer, b: Integer, c: Integer)
        RETURN a + b + c
    END METHOD

END CLASS

MAIN
    // Call overloaded static methods
    result1 = Calculator.add(5, 10)
    result2 = Calculator.add(5.5, 10.5)
    result3 = Calculator.add(1, 2, 3)

    PRINT "Sum of two integers: " + result1
    PRINT "Sum of two doubles: " + result2
    PRINT "Sum of three integers: " + result3
END MAIN
```

Code

Java

```java
class Calculator {
    // Static method to add two integers
    public static int add(int a, int b) {
        System.out.println("Calling add(int, int)");
        return a + b;
    }

    // Overloaded static method to add two doubles
    public static double add(double a, double b) {
        System.out.println("Calling add(double, double)");
        return a + b;
    }

    // Overloaded static method to add three integers
    public static int add(int a, int b, int c) {
        System.out.println("Calling add(int, int, int)");
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("--- Demonstrating Static Method Overloading ---");

        // Call the first overloaded method (int, int)
        int sum1 = Calculator.add(10, 20);
        System.out.println("Result of adding 10 and 20: " + sum1);

        System.out.println(); // New line for separation

        // Call the second overloaded method (double, double)
        double sum2 = Calculator.add(15.5, 25.3);
        System.out.println("Result of adding 15.5 and 25.3: " + String.format("%.2f", sum2));

        System.out.println(); // New line for separation

        // Call the third overloaded method (int, int, int)
        int sum3 = Calculator.add(5, 10, 15);
        System.out.println("Result of adding 5, 10, and 15: " + sum3);
    }
}
```

Output

```
--- Demonstrating Static Method Overloading ---
Calling add(int, int)
Result of adding 10 and 20: 30

Calling add(double, double)
Result of adding 15.5 and 25.3: 40.80

Calling add(int, int, int)
Result of adding 5, 10, and 15: 30
```

---

4. Implement Method Overloading in Single Class

Aim

The aim is to demonstrate **method overloading** (also known as **compile-time polymorphism**) within a single Java class using non-static methods. This allows methods to share the same name but differ in their parameter lists, enabling the same operation name to apply to different data types or quantities.

Algorithm

1. Define a class, for example, Printer.
2. Inside the Printer class, create multiple non-static methods with the same name, e.g., print.
3. Each print method must have a **different signature** (different number or types of parameters).
   - void print(int num)
   - void print(String text)
   - void print(int num, String text)
4. In the main method, create an object of the Printer class.

5. Call the overloaded print methods on this object, providing arguments that match the different signatures.
6. Observe how the compiler selects the appropriate method based on the arguments provided.

Pseudocode

```
CLASS Printer
  // Method to print an integer
  METHOD print(num: Integer)
    PRINT "Printing integer: " + num
  END METHOD

  // Overloaded method to print a string
  METHOD print(text: String)
    PRINT "Printing string: " + text
  END METHOD

  // Overloaded method to print an integer and a string
  METHOD print(num: Integer, text: String)
    PRINT "Printing integer: " + num + " and string: " + text
  END METHOD

END CLASS

MAIN
  myPrinter = NEW Printer()

  // Call overloaded methods
  myPrinter.print(123)
  myPrinter.print("Hello World")
  myPrinter.print(456, "Java")
END MAIN
```

Code

Java

```java
class Printer {
    // Method to print an integer
    public void print(int num) {
        System.out.println("Printing an integer: " + num);
    }

    // Overloaded method to print a string
    public void print(String text) {
        System.out.println("Printing a string: \"" + text + "\"");
    }

    // Overloaded method to print a double
    public void print(double dbl) {
        System.out.println("Printing a double: " + String.format("%.2f", dbl));
    }

    // Overloaded method to print an integer and a string
    public void print(int num, String text) {
        System.out.println("Printing integer " + num + " and string \"" + text + "\"");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("--- Demonstrating Method Overloading in a Single Class ---");

        Printer myPrinter = new Printer();

        // Calling overloaded methods with different arguments
        myPrinter.print(100);
        myPrinter.print("Java is powerful.");
        myPrinter.print(3.14159);
        myPrinter.print(99, "Overloading Example");
    }
}
```

Output

```
--- Demonstrating Method Overloading in a Single Class ---
Printing an integer: 100
Printing a string: "Java is powerful."
Printing a double: 3.14
Printing integer 99 and string "Overloading Example"
```

---

5. Implement Simple Inheritance in Java

Aim

The aim is to demonstrate **simple inheritance** in Java, where one class (**subclass/child class**) inherits properties and behaviors from another class (**superclass/parent class**). This promotes code reusability and establishes an "is-a" relationship.

Algorithm

1. Define a **superclass** (parent class), for example, Animal.
2. In the Animal class, declare common attributes (e.g., String species) and a common method (e.g., eat()).
3. Define a **subclass** (child class), for example, Dog, that extends the Animal class. This keyword signifies inheritance.
4. In the Dog class, add specific attributes (e.g., String breed) and methods (e.g., bark()) that are unique to Dog but not to all Animals.

5. In the main method, create an object of the Dog subclass.
6. Demonstrate that the Dog object can access both its own methods (bark()) and the inherited method (eat()) and inherited attributes (species) from the Animal class.

Pseudocode

```
CLASS Animal
  species: String

  CONSTRUCTOR(speciesParam)
    this.species = speciesParam
  END CONSTRUCTOR

  METHOD eat()
    PRINT species + " is eating."
  END METHOD

END CLASS

CLASS Dog EXTENDS Animal
  breed: String

  CONSTRUCTOR(speciesParam, breedParam)
    SUPER(speciesParam) // Call parent constructor
    this.breed = breedParam
  END CONSTRUCTOR

  METHOD bark()
    PRINT breed + " barks: Woof! Woof!"
  END METHOD

END CLASS

MAIN
  myDog = NEW Dog("Mammal", "Golden Retriever")
```

```
    // Call inherited method
    myDog.eat()

    // Call its own method
    myDog.bark()

    PRINT "My dog's species: " + myDog.species
END MAIN
```

Code

Java

```java
// Parent class (Superclass)
class Animal {
    String species;

    public Animal(String species) {
        this.species = species;
        System.out.println("Animal constructor called for species: " + species);
    }

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

    public void sleep() {
        System.out.println(species + " is sleeping.");
    }
}

// Child class (Subclass) inheriting from Animal
class Dog extends Animal {
    String breed;

    public Dog(String species, String breed) {
        // Call the constructor of the superclass (Animal)
        super(species);
        this.breed = breed;
        System.out.println("Dog constructor called for breed: " + breed);
    }

    // Dog's specific method
    public void bark() {
        System.out.println(breed + " barks loudly: Woof! Woof!");
    }

    // Dog can also inherit and use methods from Animal
    // No need to re-declare eat() or sleep()
}

public class Main {
    public static void main(String[] args) {
        System.out.println("--- Demonstrating Simple Inheritance ---");

        // Create an object of the subclass (Dog)
        Dog myDog = new Dog("Canine", "Poodle");

        System.out.println("\n--- Interacting with myDog object ---");
        // Access inherited attribute (public or protected)
        System.out.println("My dog's species: " + myDog.species);

        // Call inherited methods from Animal class
        myDog.eat();
        myDog.sleep();

        // Call Dog's specific method
        myDog.bark();
    }
}
```

Output

```
--- Demonstrating Simple Inheritance ---
Animal constructor called for species: Canine
Dog constructor called for breed: Poodle

--- Interacting with myDog object ---
My dog's species: Canine
Canine is eating food.
Canine is sleeping.
Poodle barks loudly: Woof! Woof!
```

---

6. Implement Method Overriding in Java

Aim

The aim is to demonstrate **method overriding** in Java. Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This is a key aspect of **runtime polymorphism**. The @Override annotation is typically used for clarity and compile-time checks.

Algorithm

1. Define a **superclass** (parent class), for example, Vehicle.
2. In the Vehicle class, define a method, e.g., drive(), that prints a generic message.
3. Define a **subclass** (child class), for example, Car, that extends the Vehicle class.
4. In the Car class, define a method with the **exact same signature** (name, return type, and parameters) as the drive() method in the Vehicle class.
5. Use the @Override annotation above the drive() method in the Car class to indicate that it's overriding a superclass method.
6. Inside the overridden drive() method in Car, provide a specific implementation (e.g., printing "Car is driving on roads.").
7. In the main method, create objects of both Vehicle and Car and call their respective drive() methods to show the different behaviors.

Pseudocode

```
CLASS Vehicle
  METHOD drive()
    PRINT "Vehicle is driving."
  END METHOD

END CLASS

CLASS Car EXTENDS Vehicle
  @Override
  METHOD drive() // Same method signature as in Vehicle
    PRINT "Car is driving on roads."
  END METHOD

END CLASS

MAIN
  myVehicle = NEW Vehicle()
  myCar = NEW Car()

  myVehicle.drive() // Calls Vehicle's drive()
  myCar.drive()     // Calls Car's overridden drive()
END MAIN
```

Code

Java

```
// Parent class (Superclass)
class Vehicle {
  public void drive() {
    System.out.println("Vehicle is driving.");
  }

  public void start() {
    System.out.println("Vehicle started.");
  }
}

// Child class (Subclass) inheriting from Vehicle
class Car extends Vehicle {
  // @Override annotation is optional but recommended for clarity and compile-time checks
  @Override
  public void drive() {
    System.out.println("Car is driving on the highway.");
  }

  // We can also override other methods from the parent class if needed
  @Override
  public void start() {
    System.out.println("Car engine ignited with a key.");
  }

  public void honk() {
    System.out.println("Car honks: Beep beep!");
  }
}

public class Main {
  public static void main(String[] args) {
    System.out.println("--- Demonstrating Method Overriding ---");

    // Create an object of the Superclass
    Vehicle genericVehicle = new Vehicle();
    System.out.println("\nCalling methods on genericVehicle:");
    genericVehicle.drive(); // Calls Vehicle's drive()
    genericVehicle.start(); // Calls Vehicle's start()

    // Create an object of the Subclass
    Car myCar = new Car();
    System.out.println("\nCalling methods on myCar:");
```

```
        myCar.drive();      // Calls Car's overridden drive()
        myCar.start();      // Calls Car's overridden start()
        myCar.honk();       // Calls Car's specific method
    }
}
```

Output

```
--- Demonstrating Method Overriding ---

Calling methods on genericVehicle:
Vehicle is driving.
Vehicle started.

Calling methods on myCar:
Car is driving on the highway.
Car engine ignited with a key.
Car honks: Beep beep!
```

---

7. Call Base Class Constructor Using super Keyword

Aim

The aim is to demonstrate how to explicitly call a **superclass's constructor** from a subclass's constructor using the super() keyword. This is crucial for initializing inherited instance variables from the parent class.

Algorithm

1. Define a **superclass** (parent class), e.g., Person, with a parameterized constructor that initializes some attributes (e.g., name).
2. Define a **subclass** (child class), e.g., Student, that extends Person.
3. In the Student class, create its own constructor.
4. The **first statement** inside the Student constructor must be super(...), passing the necessary arguments to match one of the Person class's constructors. This calls the parent's constructor.
5. After the super() call, initialize any attributes specific to the Student class.
6. In the main method, create an object of the Student class. Observe that both the Student's constructor and, consequently, Person's constructor are invoked.

Pseudocode

```
CLASS Person
  name: String

  CONSTRUCTOR(personName)
    this.name = personName
    PRINT "Person constructor called for " + personName
  END CONSTRUCTOR

END CLASS

CLASS Student EXTENDS Person
  studentId: String

  CONSTRUCTOR(studentName, id)
    SUPER(studentName) // Call Person's constructor
    this.studentId = id
    PRINT "Student constructor called for ID " + id
  END CONSTRUCTOR

END CLASS
```

```
MAIN
    // Create a Student object
    myStudent = NEW Student("Alice", "S12345")

    PRINT "Student Name: " + myStudent.name
    PRINT "Student ID: " + myStudent.studentId
END MAIN
```

Code

Java

```java
// Parent class (Superclass)
class Person {
    String name;

    public Person(String name) {
        this.name = name;
        System.out.println("1. Person constructor called with name: " + name);
    }
}

// Child class (Subclass) inheriting from Person
class Student extends Person {
    int studentId;

    public Student(String name, int studentId) {
        // Call to superclass constructor MUST be the first statement
        super(name); // Calls the Person(String name) constructor
        this.studentId = studentId;
        System.out.println("2. Student constructor called with ID: " + studentId);
    }

    public void displayStudentInfo() {
        System.out.println("  Student Name: " + name);
        System.out.println("  Student ID: " + studentId);
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("--- Demonstrating super() to call Base Class Constructor ---");

        // When a Student object is created, the Student constructor is called.
        // The Student constructor then explicitly calls the Person constructor using super().
        Student student1 = new Student("Bob Johnson", 1001);

        System.out.println("\n--- Displaying Student Info ---");
        student1.displayStudentInfo();
    }
}
```

Output

```
--- Demonstrating super() to call Base Class Constructor ---
1. Person constructor called with name: Bob Johnson
2. Student constructor called with ID: 1001

--- Displaying Student Info ---
 Student Name: Bob Johnson
 Student ID: 1001
```

---

8. Call Base Class Method Using super Keyword

Aim

The aim is to demonstrate how to explicitly call a **superclass's method** from within a subclass using the super keyword. This is particularly useful when a subclass **overrides** a method but still needs to execute the superclass's version of that method, perhaps as part of its own extended logic.

Algorithm

1. Define a **superclass** (parent class), e.g., Shape, with a method, e.g., draw(), that prints a generic drawing message.
2. Define a **subclass** (child class), e.g., Circle, that extends Shape.
3. In the Circle class, **override** the draw() method.
4. Inside the overridden draw() method of the Circle class, use super.draw(); to invoke the draw() method from the Shape (parent) class.
5. After the super.draw() call, add additional logic specific to the Circle's drawing.
6. In the main method, create an object of the Circle subclass and call its draw() method. Observe that both the parent's and child's drawing logic are executed.

Pseudocode

```
CLASS Shape
  METHOD draw()
    PRINT "Drawing a generic shape."
  END METHOD

END CLASS

CLASS Circle EXTENDS Shape
  radius: Double

  CONSTRUCTOR(r)
    this.radius = r
  END CONSTRUCTOR
```

```
    @Override
    METHOD draw()
        SUPER.draw() // Call the draw() method of the Shape class
        PRINT "Drawing a circle with radius " + radius + "."
    END METHOD

END CLASS

MAIN
    myCircle = NEW Circle(5.0)
    myCircle.draw()
END MAIN
```

Code

Java

```java
// Parent class (Superclass)
class Shape {
    public void displayInfo() {
        System.out.println("This is a generic shape.");
    }

    public void calculateArea() {
        System.out.println("Calculating area for a generic shape (undefined).");
    }
}

// Child class (Subclass) inheriting from Shape
class Circle extends Shape {
    double radius;

    public Circle(double radius) {
        this.radius = radius;
        System.out.println("Circle object created with radius: " + radius);
    }
    // Overriding the displayInfo method
    @Override
    public void displayInfo() {
        // Call the displayInfo method of the superclass using super.
        super.displayInfo();
        System.out.println("Specifically, this is a Circle.");
    }

    // Overriding the calculateArea method and using super's method (if it was useful)
    @Override
    public void calculateArea() {
        // In this case, super.calculateArea() might not be useful,
        // but it demonstrates the ability to call it.
        // super.calculateArea(); // Uncomment if you want to also run parent's implementation
        System.out.println("Area of Circle: " + (Math.PI * radius * radius));
    }

    public void roll() {
        System.out.println("Circle is rolling.");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("--- Demonstrating super.method() to call Base Class Method ---");

        // Create an object of the subclass
        Circle myCircle = new Circle(7.5);

        System.out.println("\n--- Calling overridden methods on myCircle ---");
        myCircle.displayInfo();    // Calls Circle's overridden displayInfo, which calls Shape's
        myCircle.calculateArea();  // Calls Circle's overridden calculateArea
        myCircle.roll();           // Calls Circle's specific method
    }
}
```

Output

```
--- Demonstrating super.method() to call Base Class Method ---
Circle object created with radius: 7.5

--- Calling overridden methods on myCircle ---
This is a generic shape.
Specifically, this is a Circle.
Area of Circle: 176.7145867644586
Circle is rolling.
```

---

9. Implement Runtime Polymorphism (Dynamic Dispatch)

Aim

The aim is to implement **runtime polymorphism** in Java, specifically through **dynamic method dispatch**. This concept allows a program to determine which overridden method to call at runtime, based on the actual type of the object, rather than the type of the reference variable.

Algorithm

1. Define a **base class** (superclass), e.g., Animal, with a method that will be overridden, e.g., makeSound().
2. Define multiple **derived classes** (subclasses), e.g., Dog, Cat, Cow, each extending Animal.
3. In each derived class, **override** the makeSound() method to provide a specific sound for that animal (e.g., "Woof!", "Meow!", "Moo!").
4. In the main method, create an array or ArrayList of the **base class type** (Animal[] or List<Animal>).
5. Populate this array/list with objects of the **derived classes** (e.g., new Dog(), new Cat()). This is known as **upcasting**.

6. Iterate through the array/list. For each element, call the makeSound() method.
7. Observe that even though the reference type is Animal, the specific overridden method from the actual object's class (e.g., Dog's makeSound()) is executed. This decision is made at **runtime**, hence "runtime polymorphism" or "dynamic method dispatch."

Pseudocode

```
CLASS Animal
  METHOD makeSound()
    PRINT "Animal makes a sound."
  END METHOD

END CLASS

CLASS Dog EXTENDS Animal
  @Override
  METHOD makeSound()
    PRINT "Dog barks: Woof!"
  END METHOD

END CLASS

CLASS Cat EXTENDS Animal
  @Override
  METHOD makeSound()
    PRINT "Cat meows: Meow!"
  END METHOD

END CLASS

CLASS Cow EXTENDS Animal
  @Override
  METHOD makeSound()
    PRINT "Cow moos: Moo!"
  END METHOD
END CLASS

MAIN
  animals = ARRAY of Animal (size 3)
  animals[0] = NEW Dog()
  animals[1] = NEW Cat()
  animals[2] = NEW Cow()

  FOR EACH animal IN animals
    animal.makeSound() // Dynamic method dispatch happens here
  END FOR
END MAIN
```

Code

Java

```java
// Base class (Superclass)
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a generic sound.");
    }

    public void eat() {
        System.out.println("Animal eats food.");
    }
}

// Derived class 1
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks: Woof! Woof!");
    }
}

// Derived class 2
```

```java
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows: Meow!");
    }
}

// Derived class 3
class Cow extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cow moos: Moo!");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("--- Demonstrating Runtime Polymorphism (Dynamic Dispatch) ---");

        // Create references of the base class type (Animal)
        // but instantiate objects of derived classes.
        // This is called Upcasting.
        Animal myDog = new Dog();
        Animal myCat = new Cat();
        Animal myCow = new Cow();
        Animal genericAnimal = new Animal();

        System.out.println("\n--- Calling makeSound() on different Animal types ---");

        // When makeSound() is called, Java determines which specific
        // overridden method to execute at RUNTIME based on the actual
        // object type (Dog, Cat, Cow, or Animal itself), not the reference type.
        myDog.makeSound();          // Calls Dog's makeSound()
        myCat.makeSound();          // Calls Cat's makeSound()
        myCow.makeSound();          // Calls Cow's makeSound()
        genericAnimal.makeSound();  // Calls Animal's makeSound()

        System.out.println("\n--- Using an Array of Base Class References ---");
        // This is a common way to see dynamic dispatch in action
        Animal[] farmAnimals = new Animal[4];
        farmAnimals[0] = new Dog();
        farmAnimals[1] = new Cat();
        farmAnimals[2] = new Cow();
        farmAnimals[3] = new Animal(); // Can also add a generic Animal

        for (Animal animal : farmAnimals) {
            animal.makeSound(); // Each call dynamically resolves to the correct overridden method
            animal.eat();       // Calls the common inherited method
        }
    }
}
```

Output

```
--- Demonstrating Runtime Polymorphism (Dynamic Dispatch) ---

--- Calling makeSound() on different Animal types ---
Dog barks: Woof! Woof!
Cat meows: Meow!
Cow moos: Moo!
Animal makes a generic sound.

--- Using an Array of Base Class References ---
Dog barks: Woof! Woof!
Animal eats food.
Cat meows: Meow!
Animal eats food.
Cow moos: Moo!
Animal eats food.
Animal makes a generic sound.
Animal eats food.
```