1. this Keyword

The this keyword is used in Java to refer to the **current object** of the class. It resolves ambiguity between instance variables and method parameters or is used to invoke other constructors.

Common Uses of this:

1. **Referencing Current Object**:
   - Access instance variables when they are shadowed by method parameters.

```
class Employee {
    String name;

    Employee(String name) {
        this.name = name; // Refers to the instance variable
    }
}
```

2. **Invoke Current Class Methods**:

```
 class Example {

void methodOne() {
    System.out.println("Method One");
    this.methodTwo(); // Calls another method in the same class
}

 void methodTwo() {
    System.out.println("Method Two");
  }
 }
```

3. **Invoke Current Class Constructor (Constructor Chaining)**:

```
class Student {

    String name;
    int age;
    Student(String name) {
        this(name, 18); // Calls another constructor in the same
class
    }

    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

2. `static` Keyword

Java `static` Keyword: In-Depth Explanation

The `static` keyword in Java is used to indicate that a member (variable, method, block, or nested class) belongs to the class rather than to any specific object. Static members are shared across all instances of the class, making them class-level rather than instance-level.

Key Features of the `static` Keyword

1. **Class-Level Scope**:
   Static members belong to the class and are shared among all objects.
2. **Memory Management**:
   Static members are stored in the **Method Area** of JVM memory, reducing redundancy.
3. **Access Without Object**:
   Static members can be accessed using the class name directly.
4. **Shared State**:
   Changes made to static members reflect across all objects of the class.

Where Can `static` Be Used?

1. Static Variables

- **Definition**: Variables declared as `static` are shared among all instances of the class. They maintain a single copy regardless of the number of objects created.

- **Initialization**: Static variables can be initialized directly, in a **static block**, or during declaration.

Example:

```
class Counter {

    static int count = 0; // Static variable

    Counter() {
        count++;
    }

    void displayCount() {
        System.out.println("Count: " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        Counter obj1 = new Counter();
        Counter obj2 = new Counter();
        obj1.displayCount(); // Output: Count: 2
        obj2.displayCount(); // Output: Count: 2
    }
}
```

---

2. Static Methods

- **Definition**: Methods declared as `static` belong to the class rather than any instance.
- **Access Restrictions**:
    - Can access **static variables** and **call other static methods**.
    - Cannot directly access instance variables or methods without creating an object.

Example:

```
class MathUtility {
    static int square(int x) {
        return x * x;
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(MathUtility.square(5)); // Output: 25
    }
}
```

---

3. Static Block

- **Definition**: A block of code that runs once when the class is loaded into memory.
- **Usage**: Used to initialize static variables or perform setup tasks.

Example:

```
class Example {
    static int value;
    static {
        value = 10;
        System.out.println("Static block executed!");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Value: " + Example.value);
    }
}
```

Output:

```
Static block executed!

Value: 10
```

4. Static Nested Class

- **Definition**: A nested class declared as `static` does not require an instance of the outer class to be instantiated.
- **Usage**: Typically used for grouping utility functions or logically related functionality.

Example:

```
class Outer {
    static class Inner {
        void display() {
            System.out.println("Static nested class");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner(); // No instance of
Outer required
        obj.display(); // Output: Static nested class
    }
}
```

Advantages of the `static` Keyword

1. **Memory Efficiency**:
   Shared static members save memory as they are not duplicated for every object.
2. **Global Access**:
   Static members can be accessed globally using the class name.
3. **Utility Functions**:
   Static methods are ideal for utility functions (e.g., `Math.sqrt()`).
4. **Initialization**:
   Static blocks allow initialization of static variables at class loading time.

Limitations of the `static` Keyword

1. **No Polymorphism**:
   Static methods cannot be overridden because they are bound to the class, not objects.
2. **Limited Access**:
   Static methods cannot directly access instance variables or methods.
3. **Thread-Safety**:
   Shared static variables may cause issues in multithreaded environments if not handled carefully.

Best Practices for Using `static`

Use for Constants:
Define constant values as `public static final`.

```
public static final double PI = 3.14159;
```

1. **Avoid Overuse**:
   Do not use `static` unnecessarily, as it can make the code harder to understand and test.
2. **Utility Classes**:
   Use static methods in utility classes, such as a `MathUtility` or `DateUtility`.
3. **Thread Safety**:
   Ensure thread-safety when modifying static variables in multithreaded applications.

Practical Examples

Example 1: Bank Interest Calculation

```
class Bank {
    static double interestRate = 3.5; // Shared by all accounts

    static double calculateInterest(double principal, int years) {
        return (principal * years * interestRate) / 100;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Interest: " +
Bank.calculateInterest(10000, 2)); // Output: Interest: 700.0
    }
}
```

Example 2: Student Count Tracker

```
class Student {
    static int totalStudents = 0; // Tracks total students
    String name;
    Student(String name) {
        this.name = name;
        totalStudents++;
    }
    static void displayTotalStudents() {
        System.out.println("Total Students: " + totalStudents);
    }
}
public class Main {
    public static void main(String[] args) {
        new Student("Alice");
        new Student("Bob");
        Student.displayTotalStudents(); // Output: Total Students: 2
    }
}
```

Example 3: Utility Functions

```
class MathUtility {
    static int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Factorial of 5: " +
MathUtility.factorial(5)); // Output: Factorial of 5: 120
    }
}
```

---

3. `final` Keyword

The `final` keyword is used to declare constants, prevent inheritance, and restrict overriding or reassignment.

Uses of `final`:

1. **Final Variables**:
    - Once assigned, the value cannot be changed.

```
class Example {
    final int MAX_VALUE = 100;

    void display() {
        System.out.println(MAX_VALUE);
    }
}
```

2. **Final Methods**:
   - Prevent a method from being overridden in a subclass.

```
class Parent {
    final void display() {
        System.out.println("Final Method");
    }
}

class Child extends Parent {
    // Cannot override display()
}
```

3. **Final Classes**:
   - Prevent a class from being extended.

```
final class Constants {
    static final double PI = 3.14159;
}

// Cannot extend Constants
```

---

4. instanceof Operator

The `instanceof` operator is used to test whether an object is an instance of a specific class or subclass.

Syntax:

```
object instanceof ClassName
```

Sample Program:

```
class Parent {}
class Child extends Parent {}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Child();

        if (obj instanceof Child) {
            System.out.println("obj is an instance of Child");
        }
        if (obj instanceof Parent) {
            System.out.println("obj is also an instance of Parent");
        }
    }
}
```

Output:

```
obj is an instance of Child
obj is also an instance of Parent
```

Use Cases:

1. **Type Checking in Polymorphism**:
    - Ensure the object is of a specific type before casting.
2. **Avoid `ClassCastException`**:
    - Safeguards against invalid type casting.

Comparison Table

Feature	this	final	instanceof	static
Purpose	Refers to the current object.	Restricts modification or inheritance.	Checks object type.	Creates shared class-level members.
Scope	Current instance of the class.	Variables, methods, or classes.	Objects and types.	Variables, methods, blocks, or classes.
Common Usage	Resolving name conflicts, constructor chaining.	Constants, prevent overriding.	Polymorphism type checks.	Utility methods, shared properties.