Java Fundamentals: Methods, Overloading, Overriding, Constructors, and More

This guide provides an in-depth explanation of **Methods**, **Overloading and Overriding**, **Constructors**, **Destructors**, **Classes and Objects**, **Access Modifiers**, and the `this` **keyword** in Java, with code examples, clear explanations, and **syntax** for each concept.

---

1. Methods

In Java, a **method** is a block of code that performs a specific task. It defines the behavior of an object.

Syntax:

```
returnType methodName(parameters) {
    // Method body
}
```

- **returnType**: The type of value the method will return (e.g., `int`, `void` for no return value).
- **methodName**: The name of the method.
- **parameters**: A list of parameters the method accepts (optional).

Example:

```
class Calculator {

    // Method to add two numbers
    public int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 3));  // Output: 8
    }
}
```

---

2. Method Overloading

Method Overloading occurs when multiple methods with the same name exist in a class but with different parameters (either in number or type).

Syntax for Overloading:

```
returnType methodName(parameter1, parameter2, ...) {
    // Method body
}
```

Example of Overloading:

```
class Printer {

    // Method to print an integer
    public void print(int a) {
        System.out.println("Integer: " + a);
    }

    // Method to print a string
    public void print(String s) {
        System.out.println("String: " + s);
    }

    public static void main(String[] args) {
        Printer p = new Printer();
        p.print(100);   // Output: Integer: 100
        p.print("Hello!");  // Output: String: Hello!
    }
}
```

---

3. Method Overriding

Method Overriding happens when a subclass provides its own implementation of a method that is already defined in its superclass. The method signature must match.

Syntax for Overriding:

```
class Superclass {
    returnType methodName() {
        // Superclass method
    }
}

class Subclass extends Superclass {
```

```
    @Override
    returnType methodName() {
        // Subclass method
    }
}
```

Example of Overriding:

```
class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();  // Output: Dog barks
    }
}
```

---

4. Constructors and Destructors

4.1. Constructors

A **constructor** is a special method used to initialize objects when they are created. It has the same name as the class and no return type.

Syntax for Constructor:

```
class ClassName {
    // Constructor
    public ClassName(parameters) {
        // Initialization code
    }
}
```

Example of a Constructor:

```java
class Car {
    String model;
    int year;

    // Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public void display() {
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }

    public static void main(String[] args) {
        Car car = new Car("Tesla", 2023);
        car.display();  // Output: Model: Tesla, Year: 2023
    }
}
```

4.2. Destructors in Java

Java does not have explicit destructors like C++. Instead, Java uses **garbage collection** to automatically reclaim memory when an object is no longer in use. However, the `finalize()` method can be used to perform clean-up tasks before an object is destroyed (though it's rarely used).

Syntax for `finalize()`:

```java
class ClassName {
    @Override
    protected void finalize() throws Throwable {
        // Clean-up code
    }
}
```

Example of `finalize()` (not commonly used):

```java
class Sample {
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Object is being garbage collected");
    }

    public static void main(String[] args) {
        Sample obj = new Sample();
```

```
        obj = null;  // Dereference the object
        System.gc();  // Suggest garbage collection
    }
}
```

---

5. Classes and Objects

5.1. Classes

A **class** is a blueprint for creating objects, defining attributes and behaviors.

5.2. Objects

An **object** is an instance of a class. It is created using the `new` keyword.

Syntax to Create an Object:

```
ClassName objectName = new ClassName(parameters);
```

Example of Classes and Objects:

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

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

    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }

    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
        person.display();  // Output: Name: Alice, Age: 30
    }
}
```

---

6. Access Modifiers

6.1. `public` Modifier

A member (variable, method, class) with `public` access can be accessed from any other class.

Syntax for `public`:

```
public class ClassName {
    public returnType methodName() {
        // Method body
    }
}
```

Example of `public`:

```
class Person {
    public String name;  // Accessible from any class

    public void display() {
        System.out.println("Name: " + name);
    }
}

public class Test {
    public static void main(String[] args) {
        Person person = new Person();
        person.name = "John";  // Accessible due to 'public' modifier
        person.display();  // Output: Name: John
    }
}
```

6.2. `private` Modifier

A member with `private` access is accessible only within the same class.

Syntax for `private`:

```
class ClassName {
    private returnType variableName;

    private void methodName() {
        // Method body
```

```
    }
}
```

Example of `private`:

```
class Person {
    private int age;  // Accessible only within the Person class

    public void setAge(int age) {
        this.age = age;
    }

    public void display() {
        System.out.println("Age: " + age);
    }
}

public class Test {
    public static void main(String[] args) {
        Person person = new Person();
        person.setAge(25);  // Accessing private field through setter method
        person.display();  // Output: Age: 25
    }
}
```

6.3. `protected` Modifier

A member with `protected` access can be accessed within the same package or in subclasses (even if they are in different packages).

Syntax for `protected`:

```
class Superclass {
    protected returnType variableName;
}
```

Example of `protected`:

```
class Animal {
    protected String name;  // Accessible in subclasses

    public void display() {
        System.out.println("Animal name: " + name);
    }
}
```

```
class Dog extends Animal {
    public void setName(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.setName("Buddy");
        dog.display();  // Output: Animal name: Buddy
    }
}
```

6.4. Default (Package-Private) Modifier

If no modifier is specified, the default access level is applied. Members with default access can only be accessed within the same package.

Syntax for Default Modifier:

```
class ClassName {
    returnType methodName() {
        // Method body
    }
}
```

Example of Default Modifier:

```
class Person {
    String name;  // Default access

    public void display() {
        System.out.println("Name: " + name);
    }
}

public class Test {
    public static void main(String[] args) {
        Person person = new Person();
        person.name = "Alice";  // Accessible because both classes are in the same package
        person.display();  // Output: Name: Alice
    }
}
```

---

7. `this` Keyword

The **`this` keyword** is a reference to the current object. It helps to differentiate between instance variables and parameters with the same name, and can also be used to invoke other constructors.

Syntax for `this`:

```
this.variableName;
this.methodName();
```

Example 1: Referring to Instance Variables

```
class Student {
    String name;
    int age;

    // Constructor
    public Student(String name, int age) {
        this.name = name;  // Using 'this' to differentiate
        this.age = age;    // instance variables from parameters
    }

    public void display() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }

    public static void main(String[] args) {
        Student student = new Student("John", 20);
        student.display();  // Output: Name: John, Age: 20
    }
}
```

Example 2: Invoking Another Constructor

```
class Rectangle {
    int length;
    int width;

    // Constructor with parameters
    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
```

```
    }

    // Constructor calling another constructor using 'this'
    public Rectangle(int side) {
        this(side, side);  // Calling the constructor with two parameters
    }

    public void display() {
        System.out.println("Length: " + length);
        System.out.println("Width: " + width);
    }

    public static void main(String[] args) {
        Rectangle square = new Rectangle(5);  // Calls the constructor with one parameter
        square.display();  // Output: Length: 5, Width: 5
    }
}
```

Summary

- **Methods** define behavior in classes.
- **Method Overloading** allows multiple methods with the same name but different parameters.
- **Method Overriding** provides a new implementation of a method in a subclass.
- **Constructors** are special methods to initialize objects; Java has no explicit destructors.
- **Classes and Objects** are fundamental OOP concepts, where classes define blueprints and objects are instances of classes.
- **Access Modifiers** (`public`, `private`, `protected`, default) control visibility and access levels.
- The **`this` keyword** refers to the current object and is used to resolve naming conflicts, invoke constructors, and pass the current object as a parameter.
