### ****Super Keyword****

In Java, the super keyword is a reference variable used to refer to the immediate parent class object. It is commonly used in the context of inheritance, allowing access to:

1. Parent class methods
2. Parent class constructors
3. Parent class instance variables

### ****Key Points****:

* super can be used to access methods and variables from the parent class that have been overridden or shadowed in the subclass.
* super can be used to invoke the parent class constructor.
* It is always used in the context of inheritance where there is a parent-child relationship between classes.

### ****Uses of**** super ****Keyword****

1. **Accessing Parent Class Methods**:
   * You can use super to call a method of the parent class when it has been overridden by a subclass.
2. **Accessing Parent Class Constructor**:
   * The super() constructor is used to invoke a parent class constructor. If no constructor is explicitly called, the default constructor of the parent class is called automatically.
3. **Accessing Parent Class Instance Variables**:
   * The super keyword can also be used to access instance variables of the parent class when they are shadowed (i.e., have the same name) in the subclass.

### ****Detailed Explanation with Examples****

#### 1. ****Accessing Parent Class Methods****:

When a subclass overrides a method from the parent class, we can use super to call the method in the parent class.

**Example:**

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

void sound() {

// Overriding the sound method

System.out.println("Dog barks");

}

void callSuperSound() {

// Calling the sound method of the parent class using super

super.sound();

}

}

public class Test {

public static void main(String[] args) {

Dog dog = new Dog();

dog.sound(); // Calls Dog's overridden method

dog.callSuperSound(); // Calls Animal's method using super

}

}

**Output:**

Dog barks

Animal makes a sound

**Explanation:**

* The sound() method is overridden in the Dog class.
* The callSuperSound() method in the Dog class uses super.sound() to invoke the sound() method of the parent Animal class, bypassing the overridden method.

#### 2. ****Accessing Parent Class Constructor****:

You can call a parent class constructor using super() to initialize the parent class part of an object.

**Example:**

class Animal {

Animal() {

System.out.println("Animal constructor");

}

}

class Dog extends Animal {

Dog() {

// Calling the parent class constructor using super()

super(); // Calls Animal constructor

System.out.println("Dog constructor");

}

}

public class Test {

public static void main(String[] args) {

Dog dog = new Dog(); // Creates a Dog object and calls both Animal and Dog constructors

}

}

**Output:**

Animal constructor

Dog constructor

**Explanation:**

* The Dog class has a constructor that calls the parent class (Animal) constructor using super(). This ensures that the parent class is properly initialized before the subclass-specific code executes.

#### 3. ****Accessing Parent Class Instance Variables****:

If a subclass has a field with the same name as a field in its parent class (shadowing), super can be used to refer to the parent class field.

**Example:**

class Animal {

String name = "Animal";

}

class Dog extends Animal {

String name = "Dog";

void printNames() {

System.out.println("Subclass name: " + name); // Refers to Dog's name

System.out.println("Superclass name: " + super.name); // Refers to Animal's name

}

}

public class Test {

public static void main(String[] args) {

Dog dog = new Dog();

dog.printNames();

}

}

**Output:**

Subclass name: Dog

Superclass name: Animal

**Explanation:**

* In this example, the Dog class shadows the name variable from the Animal class.
* The super.name refers to the name variable in the parent class (Animal), while name without super refers to the field in the Dog class.

### ****Key Points to Remember****:

* **Calling Parent Class Methods**: You can use super to call overridden methods from the parent class.
* **Calling Parent Class Constructors**: Use super() to invoke the parent class constructor explicitly. If no constructor is called, the default constructor of the parent is used.
* **Accessing Parent Class Fields**: If a subclass has fields that shadow the parent class fields, you can use super to refer to the parent class's field.

### ****Constructor Rules with**** super():

* The super() call must be the **first statement** in the constructor if it's used.
* If no explicit call to super() is made in the subclass constructor, the default constructor of the parent class is called automatically.

#### ****Summary****:

* The super keyword is an essential tool in Java's inheritance mechanism.
* It allows a subclass to refer to and manipulate its parent class’s methods, constructors, and fields, especially when they are overridden or shadowed.
* It helps ensure that a subclass can properly initialize and interact with the state and behavior of the parent class.

### ****Final Keyword****

The final keyword in Java is used to define constants, prevent method overriding, and prevent inheritance. It can be applied to variables, methods, and classes. When applied, it provides immutability or restricts modification to certain aspects of the class or object.

### ****Key Uses of the**** final ****Keyword:****

1. **Final Variables (Constant Variables)**:
   * When applied to a variable, the final keyword makes the variable constant, meaning its value cannot be changed once assigned.
2. **Final Methods**:
   * When applied to a method, the final keyword prevents the method from being overridden by any subclass.
3. **Final Classes**:
   * When applied to a class, the final keyword prevents the class from being subclassed or extended.
4. **Final Parameters**:
   * When applied to method parameters, the final keyword ensures that the parameter value cannot be changed inside the method.

### ****Detailed Explanation with Examples****

#### 1. ****Final Variables (Constants)****:

When a variable is declared as final, its value cannot be reassigned once it is initialized.

**Example:**

class Circle {

final double PI = 3.14159; // Declaring a constant

void printPi() {

System.out.println("The value of PI is: " + PI);

}

void changePi() {

// PI = 3.14; // Error: cannot assign a value to a final variable

}

}

public class Test {

public static void main(String[] args) {

Circle circle = new Circle();

circle.printPi();

// circle.PI = 3.14; // Error: cannot modify final variable PI

}

}

**Output:**

The value of PI is: 3.14159

**Explanation:**

* The variable PI is declared as final, so its value cannot be changed once it's assigned.

#### 2. ****Final Methods****:

A final method cannot be overridden by any subclass. This is useful when you want to ensure that a method's behavior is not altered by any subclass.

**Example:**

class Animal {

final void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

// Trying to override the final method will result in a compile-time error

// void sound() {

// System.out.println("Dog barks");

// }

}

public class Test {

public static void main(String[] args) {

Animal animal = new Animal();

animal.sound();

Dog dog = new Dog();

dog.sound(); // This will call the Animal's final method

}

}

**Output:**

Animal makes a sound

Animal makes a sound

**Explanation:**

* The sound() method in the Animal class is marked as final, so it cannot be overridden in the Dog subclass. Any attempt to override it would result in a compile-time error.

#### 3. ****Final Classes****:

A final class cannot be subclassed. This is useful when you want to ensure that the class is not extended, for example, to create immutable objects.

**Example:**

final class Vehicle {

void move() {

System.out.println("Vehicle is moving");

}

}

// The following code will result in a compile-time error:

// class Car extends Vehicle { // Error: Cannot inherit from final 'Vehicle'

// void move() {

// System.out.println("Car is moving");

// }

// }

public class Test {

public static void main(String[] args) {

Vehicle vehicle = new Vehicle();

vehicle.move();

}

}

**Output:**

Vehicle is moving

**Explanation:**

* The Vehicle class is declared final, so it cannot be extended by any other class like Car. Any attempt to subclass a final class will lead to a compile-time error.

#### 4. ****Final Parameters****:

When the final keyword is applied to a method parameter, it ensures that the parameter value cannot be changed inside the method.

**Example:**

class MathOperation {

void multiply(final int number) {

// number = number \* 2; // Error: cannot assign a value to final variable number

System.out.println("The number is: " + number);

}

}

public class Test {

public static void main(String[] args) {

MathOperation math = new MathOperation();

math.multiply(5);

}

}

**Output:**

The number is: 5

**Explanation:**

* The parameter number in the multiply() method is declared final. Therefore, you cannot modify the value of number within the method.

### ****Final with Inheritance****:

* **Final Methods**: A method declared as final cannot be overridden in any subclass, even if the subclass is a direct child.
* **Final Classes**: A class declared as final cannot have any subclasses. This can be useful to prevent any subclass from modifying the behavior of the class.

### ****Key Points to Remember****:

1. **Final Variables**: Once initialized, the value of a final variable cannot be changed. They are typically used to define constants.
2. **Final Methods**: A final method cannot be overridden in any subclass, which ensures that the method's behavior remains consistent.
3. **Final Classes**: A final class cannot be extended, which ensures that the class's behavior is preserved without modification.
4. **Final Parameters**: A final parameter ensures that the value of the parameter cannot be changed within the method.

### ****Summary****:

The final keyword is a powerful tool in Java that allows you to define constants, prevent method overriding, prevent inheritance, and ensure the immutability of method parameters. It helps ensure that the design of your classes and methods remains secure and unchangeable where required.