**National University of Computer & Emerging Sciences, Karachi**

**Computer Science Department**

**Spring 2022, Lab Manual - 01**

|  |  |
| --- | --- |
| **Course Code: SL3001** | **Course : Software Development and construction** |
| **Instructor(s) :** | **Miss Nida Munawar, Miss Abeeha Sattar** |

**Lab # 02**

# Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## Usage of Java super Keyword

A screenshot of a phone

Description automatically generated with low confidence

## super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

class Animal{

String color="white";

}

class Dog extends Animal{

String color="black";

void printColor(){

System.out.println(color);//prints color of Dog class

System.out.println(super.color);//prints color of Animal class

}

}

class TestSuper1{

public static void main(String args[]){

Dog d=new Dog();

d.printColor();

}}

Text

Description automatically generated

## 2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void eat(){System.out.println("eating bread...");}

void bark(){System.out.println("barking...");}

void work(){

super.eat(); // it could be second statement inside method no error

bark();

}

}

class TestSuper2{

public static void main(String args[]){

Dog d=new Dog();

d.work();

}}

Text

Description automatically generated

## 3) super is used to invoke parent class constructor.

#### Note: super() is added in each class constructor automatically by compiler if there is no super() or this().

**Diagram

Description automatically generated**

**example of super keyword where super() is provided by the compiler implicitly.**

class Animal{

Animal(){System.out.println("animal is created");}

}

class Dog extends Animal{

Dog(){

System.out.println("dog is created");

}

}

class TestSuper2{

public static void main(String args[]){

Dog d=new Dog();

}}

**Text

Description automatically generated**

**example of super keyword where super() is provided explicitly**

**class Animal{**

**Animal(){System.out.println("animal is created");}**

**}**

**class Dog extends Animal{**

**Dog(){**

**super();**

**System.out.println("dog is created");**

**}**

**}**

**class TestSuper2{**

**public static void main(String args[]){**

**Dog d=new Dog();**

**}}**

**Text

Description automatically generated**

**Real usage of super**

**1:**

**class** Parentclass

{

//no-arg constructor

Parentclass(){

System.***out***.println("no-arg constructor of parent class");

}

//arg or parameterized constructor

Parentclass(String str){

System.***out***.println("parameterized constructor of parent class");

}

}

**class** Subclass **extends** Parentclass

{

Subclass(){

**super**("Hahaha");

System.***out***.println("Constructor of child class");

}

**void** display(){

System.***out***.println("Hello");

}

**public** **static** **void** main(String args[]){

Subclass obj= **new** Subclass();

obj.display();

}

}

**2.**

**class** Person{

**int** id;

String name;

Person(**int** id,String name){

**this**.id=id;

**this**.name=name;

}

}

**class** Emp **extends** Person{

**float** salary;

Emp(**int** id,String name,**float** salary){

**super**(id,name);//reusing parent constructor

**this**.salary=salary;

}

**void** display(){System.***out***.println(id+" "+name+" "+salary);}

}

**class** TestSuper2{

**public** **static** **void** main(String[] args){

Emp e1=**new** Emp(1,"Nida",45000f);

e1.display();

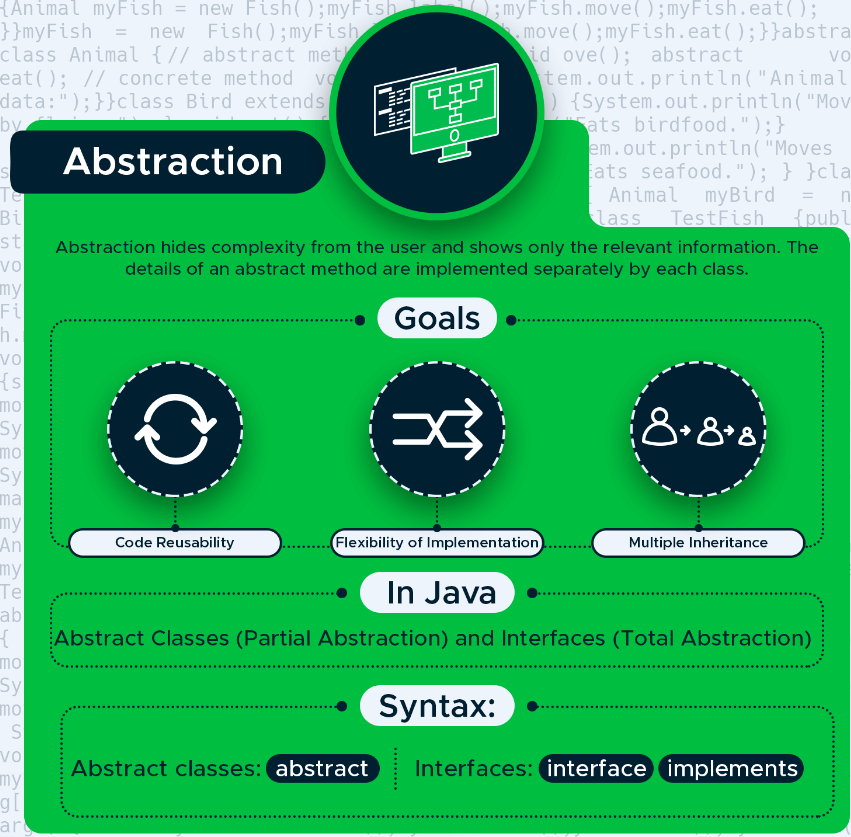
}}

**Text

Description automatically generated**

**Abstraction:**

Abstraction aims to hide complexity from the users and show them only the relevant information. For example, if you want to drive a car, you don’t need to know about its internal workings. The same is true of Java classes. You can hide internal implementation details by using abstract classes or interfaces. On the abstract level, you only need to define the method signatures (name and parameter list) and let each class implement them in their own way.



With abstraction, you can hide the internal workings of an object and only show the features the user needs to know about. Java provides two ways to implement abstraction: abstract classes and interfaces. With abstract classes, you can achieve partial abstraction, while interfaces make total (100%) abstraction possible.

**Abstract classes**

An abstract class is a superclass (parent class) that cannot be instantiated. You need to instantiate one of its child classes if you want to create a new object. Abstract classes can have both abstract and concrete methods. Abstract methods contain only the method signature, while concrete methods declare a method body as well. Abstract classes are defined with the abstract keyword.

In the example below, you can see an abstract class called Animal with two abstract and one concrete method.

abstract class Animal {

// abstract methods

abstract void move();

abstract void eat();

// concrete method

void label() {

System.out.println("Animal's data:");

}

}

Extend the Animal abstract class with two child classes: Bird and Fish. Both of them set up their own functionality for the move() and eat() abstract methods.

class Bird extends Animal {

void move() {

System.out.println("Moves by flying.");

}

void eat() {

System.out.println("Eats birdfood.");

}

}

class Fish extends Animal {

void move() {

System.out.println("Moves by swimming.");

}

void eat() {

System.out.println("Eats seafood.");

}

}

Now, test it with the TestBird and TestFish classes. Both call the one concrete (label()) and the two abstract (move() and eat()) methods.

class TestBird {

public static void main(String[] args) {

Animal myBird = new Bird();

//you can create reference of abstract class

myBird.label();

myBird.move();

myBird.eat();

}

}

class TestFish {

public static void main(String[] args) {

Animal myFish = new Fish();

myFish.label();

myFish.move();

myFish.eat();

}

}

#### Interfaces

An interface is a 100% abstract class. It can have only static, final, and public fields and abstract methods. It’s frequently referred to as a blueprint of a class as well. Java interfaces allow us to implement multiple inheritance in our code, as a class can implement any number of interfaces. Classes can access an interface using the implements keyword.

#### the Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

Diagram

Description automatically generated

Diagram

Description automatically generated

In the example, define two interfaces, Animal and Bird. Animal has two abstract methods, while Bird has two static fields and an abstract method.

interface Animal {

public void eat();

public void sound();

}

interface Bird {

int numberOfLegs = 2;

String outerCovering = "feather";

public void fly();

}

The class Eagle implements both interfaces. It defines its own functionality for the three abstract methods. The eat() and sound() methods come from the Animal class, while fly() comes from Bird.

class Eagle implements Animal, Bird {

public void eat() {

System.out.println("Eats reptiles and amphibians.");

}

public void sound() {

System.out.println("Has a high-pitched whistling sound.");

}

public void fly() {

System.out.println("Flies up to 10,000 feet.");

}

}

In the TestEagle test class, instantiate a new Eagle object (called myEagle) and print out all the fields and methods to the console.

As static fields don’t belong to a specific object but to a whole class, you need to access them from the Bird interface instead of the myEagle object.

class TestEagle {

public static void main(String[] args) {

Eagle myEagle = new Eagle();

myEagle.eat();

myEagle.sound();

myEagle.fly();

System.out.println("Number of legs: " + Bird.numberOfLegs);

System.out.println("Outer covering: " + Bird.outerCovering);

}

}

## Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

Diagram

Description automatically generated

## Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of [class](https://www.javatpoint.com/object-and-class-in-java) because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

interface Printable{

void print();

}

interface Showable{

void print();

}

class TestInterface3 implements Printable, Showable{

public void print(){System.out.println("Hello");}

public static void main(String args[]){

TestInterface3 obj = new TestInterface3();

obj.print();

}

}

Output:

Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

## Interface inheritance

A class implements an interface, but one interface extends another interface.

interface Printable{

void print();

}

interface Showable extends Printable{

void show();

}

class TestInterface4 implements Showable{

public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}

public static void main(String args[]){

TestInterface4 obj = new TestInterface4();

obj.print();

obj.show();

}

}

## Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

interface Drawable{

void draw();

default void msg(){System.out.println("default method");}

}

class Rectangle implements Drawable{

public void draw(){System.out.println("drawing rectangle");}

}

class TestInterfaceDefault{

public static void main(String args[]){

Drawable d=new Rectangle();

d.draw();

d.msg();

}}

drawing rectangle

default method

## Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

interface Drawable{

void draw();

static int cube(int x){return x\*x\*x;}

}

class Rectangle implements Drawable{

public void draw(){System.out.println("drawing rectangle");}

}

class TestInterfaceStatic{

public static void main(String args[]){

Drawable d=new Rectangle();

d.draw();

System.out.println(Drawable.cube(3));

}}

**An Example of Diamond Problem with Default Methods**

Graphical user interface, text, application, email

Description automatically generated

**Solution: to overcome with this error override the duplicated method in InterfaceExample**

**An Example of Diamond Problem with Default Methods**

**interface** MyInterface1{

**int** ***num*** = 100;

**public** **default** **void** display() {

System.***out***.println("display method of MyInterface1");

}

}

**interface** MyInterface2{

**int** ***num*** = 1000;

**public** **default** **void** display() {

System.***out***.println("display method of MyInterface2");

}

}

**public** **class** InterfaceExample **implements** MyInterface1, MyInterface2{

**public** **void** display() {

MyInterface1.**super**.display();

//or,

MyInterface2.**super**.display();

}

**public** **static** **void** main(String args[]) {

InterfaceExample obj = **new** InterfaceExample();

obj.display();

System.***out***.println(MyInterface1.***num***);//fields are by default static and we can not use super with static

}

}

Text

Description automatically generated

**An Example of Diamond Problem with static Methods**

**package** secpro;

**interface** MyInterface1{

**int** ***num*** = 100;

**public** **static** **void** display() {

System.***out***.println("display method of MyInterface1");

}

}

**interface** MyInterface2{

**int** ***num*** = 1000;

**public** **static** **void** display() {

System.***out***.println("display method of MyInterface2");

}

}

**public** **class** InterfaceExample **implements** MyInterface1, MyInterface2{

**public** **void** display() {

MyInterface1.*display*();//we can not use super with static

//or,

MyInterface2.*display*();

}

**public** **static** **void** main(String args[]) {

InterfaceExample obj = **new** InterfaceExample();

obj.display();

System.***out***.println(MyInterface1.***num***);//fields are by default static and we can not use super with static

}

}

Text

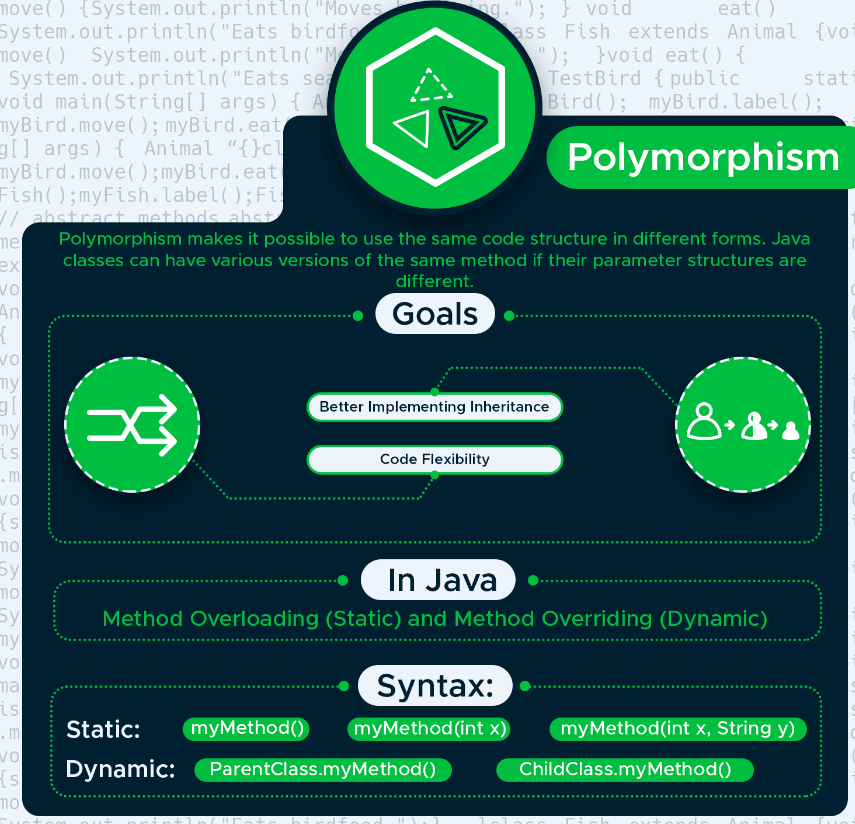
Description automatically generated

# Why can't we use the "super" keyword is in a static method in java?

# Why can't we use the "this" keyword is in a static method in java?

# Why can't we use the "super" and “this” keyword in a constructor in java?

### Polymorphism



Polymorphism makes it possible to use the same entity in different forms. In Java, this means that you can declare several methods with the same name until they are different in certain characteristics. Java provides us with two ways to implement polymorphism: method overloading and method overriding.

**Static polymorphism**

Method overloading means that you can have several methods with the same name within a class. However, the number, names, or types of their parameters need to be different.

For example, the Bird() class below has three fly() methods. The first one doesn’t have any parameters, the second one has one parameter (height), and the third one has two parameters (name and height).

class Bird {

public void fly() {

System.out.println("The bird is flying.");

}

public void fly(int height) {

System.out.println("The bird is flying " + height + " feet high.");

}

public void fly(String name, int height) {

System.out.println("The " + name + " is flying " + height + " feet high.");

}

}

The test class instantiates a new Bird object and calls the fly() method three times. Firstly, without parameters, secondly, with one integer parameter for height, and thirdly, with two parameters for name and height.

class TestBird {

public static void main(String[] args) {

Bird myBird = new Bird();

myBird.fly();

myBird.fly(10000);

myBird.fly("eagle", 10000);

}

}

**Dynamic polymorphism**

## Dynamic Polymorphism or DMD

**Dynamic polymorphism** is a process or mechanism in which a call to an overridden method is to resolve at runtime rather than compile-time. It is also known as [**runtime polymorphism**](https://www.javatpoint.com/runtime-polymorphism-in-java) or **dynamic method dispatch**. We can achieve dynamic polymorphism by using the [**method overriding**](https://www.javatpoint.com/method-overriding-in-java).

In this process, an overridden method is called through a reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

**package** secpro;

//parent class

**class** Fruits

{

**public** **void** color()

{

System.***out***.println("Parent class method is invoked.");

}

}

//derived or child class that extends the parent class

**class** Mango **extends** Fruits

{

//overrides the color() method of the parent class

**public** **void** color()

{

System.***out***.println("The child class method is invoked.");

}

}

**public** **class** DynamicPolymorphismExample

{

**public** **static** **void** main(String args[])

{

//assigning a child class object to a parent class reference

Fruits fruits = **new** Mango();

//invoking the method

fruits.color();

}

}

A picture containing text

Description automatically generated