

# Java SE & Spring

## Module 1: Java SE

## 07.OOP in Depth



# Inheritance

Inheritance is one of the key features of OOP that allows us to create a new class from an existing class. The new class that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class). The **extends** keyword is used to perform inheritance in Java.

## Example:

```
1  class Animal {
2      // methods and fields
3  }
4
5  // use of extends keyword
6  // to perform inheritance
7  class Dog extends Animal {
8
9      // methods and fields of Animal
10     // methods and fields of Dog
11 }
```

# is-a Relationship

In Java, inheritance is an is-a relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example:

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Sedan** is a **Car**
- **Cat** is an **Animal**

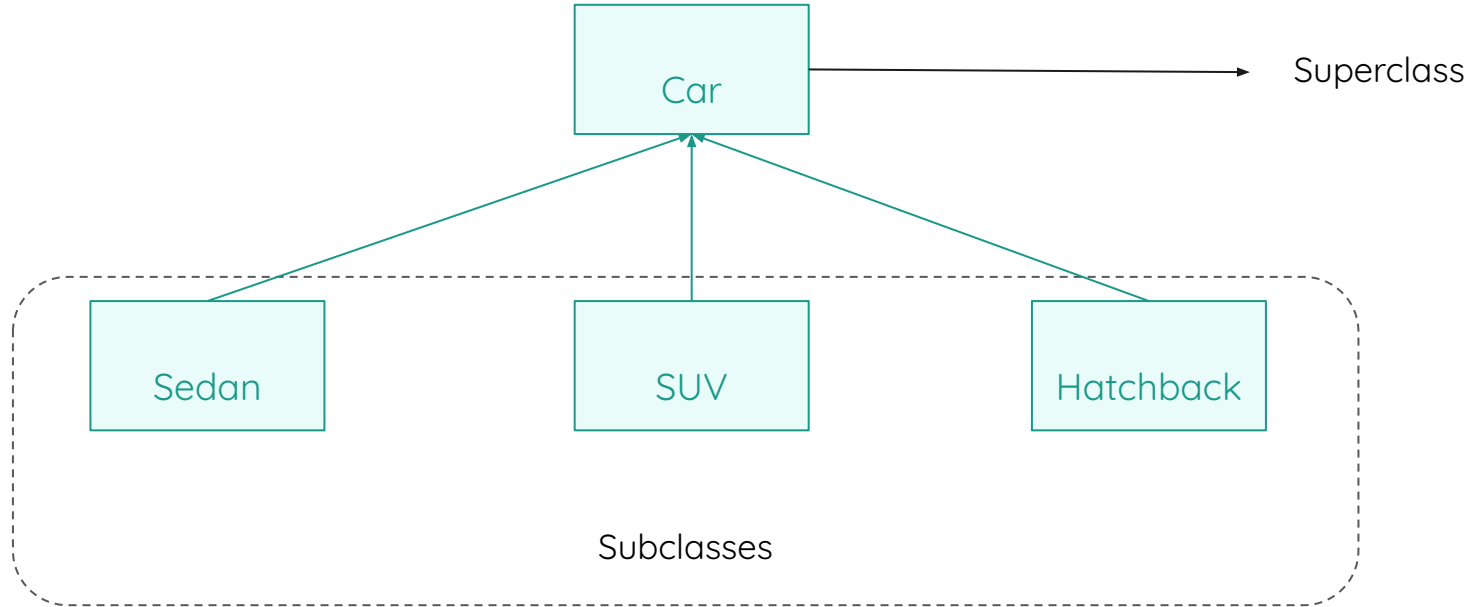
Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on, and **Vehicle** is superclass of the **Car** class, on the other hand, **Car** is subclass of the **Vehicle** class.

---

Further reading: <https://www.programiz.com/java-programming/inheritance>

# is-a Relationship

Example:



Sedan, SUV and Hatchback **is-a** Car.

# Method Overriding

If the same method is defined in both the superclass and the subclass, then the method of the subclass class overrides the method of the superclass. This is known as method **overriding**.

## Example:

```
1  class Car{
2      public void displayInfo() {
3          System.out.println("This is a car.");
4      }
5  }
6
7  class Sedan extends Car{
8      @Override
9      public void displayInfo() {
10         System.out.println("This is a sedan car.");
11     }
12 }
```

# Method Overriding

If we create a `Sedan` object where in previous slide, `Sedan` class's `displayInfo()` method will be executed rather than `Car` class's `displayInfo()` method:

```
1  class Main {  
2      public static void main(String[] args) {  
3          Sedan sedanCar = new Sedan();  
4          sedanCar.displayInfo();  
5      }  
6  }
```

Notice the use of the `@Override` annotation in our example. In Java, annotations which we will talk about it later, are the metadata that we used to provide information to the compiler. Here, the `@Override` annotation specifies the compiler that the method after this annotation overrides the method of the superclass.

# Method Overriding

## Overriding Rules:

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
- We cannot override the method declared as **final** and **static**.
- We should always override abstract methods of the superclass (will be discussed in next slides).



# super keyword

The **super** keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods). These below are the use cases of the super keyword:

1. To call methods of the superclass that is overridden in the subclass.
2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

---

Further reading: <https://www.programiz.com/java-programming/super-keyword>

# super keyword

## To call methods of the superclass that is overridden in the subclass:

We have talked about this use case in previous slides. In the `Car` and `Sedan` example, we have created a `Sedan` object which is a subclass of the `Car`, then called the `displayInfo()` method. As you noticed, the `displayInfo()` of the `Sedan` class have executed rather than `Car` class's method. So, if we want to call the superclass's `displayInfo()` method, how can we achieve this? The answer is **super** keyword.

---

Further reading: <https://www.programiz.com/java-programming/super-keyword>

# super keyword

To call methods of the superclass that is overridden in the subclass:

Example:


```
1  class Car{
2      public void displayInfo() {
3          System.out.println("This is a car.");
4      }
5  }
6
7  class Sedan extends Car{
8      @Override
9      public void displayInfo() {
10         super.displayInfo();
11         System.out.println("This is a sedan car.");
12     }
13 }
```

---

Further reading: <https://www.programiz.com/java-programming/super-keyword>

# super keyword

## Access Attributes of the Superclass:

The superclass and subclass can have attributes with the same name. We use the **super** keyword to access the attribute of the superclass. Example is on the next slide. 

---

Further reading: <https://www.programiz.com/java-programming/super-keyword>

# super keyword

## Access Attributes of the Superclass:

```
1  class Animal {
2      protected String type="animal";
3  }
4
5  class Dog extends Animal {
6      public String type="mammal";
7
8      public void printType() {
9          System.out.println("I am a/an " + type);
10         System.out.println("I am also a/an " + super.type);
11     }
12 }
13
14 class Main {
15     public static void main(String[] args) {
16         Dog dog1 = new Dog();
17         dog1.printType();
18     }
19 }
```

# super keyword

## To access superclass constructor:

As we know, when an object of a class is created, its default constructor is automatically called. To explicitly call the superclass constructor from the subclass constructor, we use `super()`. It's a special form of the **super** keyword. `super()` can be used only inside the subclass constructor and must be the first statement.

Example is on the next slide. 

---

Further reading: <https://www.programiz.com/java-programming/super-keyword>

# super keyword

## To access superclass constructor:

```
1  class Animal {
2      Animal() {
3          System.out.println("I am an animal");
4      }
5  }
6
7  class Dog extends Animal {
8      // default or no-arg constructor of class Dog
9      Dog() {
10         super();
11         System.out.println("I am a dog");
12     }
13 }
14
15 class Main {
16     public static void main(String[] args) {
17         Dog dog1 = new Dog();
18     }
19 }
```

# Abstraction

**Abstraction** is an important concept of object-oriented programming that allows us to hide unnecessary details and only show the needed information. This allows us to manage complexity by omitting or hiding details with a simpler, higher-level idea. Real life examples of the abstraction is below:

- Your car is a great example of abstraction. You can start a car by turning the key or pressing the start button. You don't need to know how the engine is getting started, what all components your car has. The car internal implementation and complex logic is completely hidden from the user.
- We can heat our food in Microwave. We press some buttons to set the timer and type of food. Finally, we get a hot and delicious meal. The microwave internal details are hidden from us. We have been given access to the functionality in a very simple manner.



# Abstraction

There are two types of abstraction:

- Data Abstraction
- Process Abstraction

## 1. Data Abstraction

When the object data is not visible to the outer world, it creates data abstraction. If needed, access to the Objects' data is provided through some methods. eg. getter and setter methods.

## 2. Process Abstraction

We don't need to provide details about all the functions of an object. When we hide the internal implementation of the different functions involved in a user operation, it creates process abstraction.

# Abstraction

## Abstraction in Java

Abstraction in Java is implemented through **interfaces** and **abstract classes**. They are used to create a base implementation or contract for the actual implementation classes.

# Abstract Classes

An abstract class can have an abstract method without body and it can have methods with implementation also. Below, we can summarize the important point of the abstract classes:

- **abstract** keyword is used to create an abstract class in java.
- Abstract class in java can't be instantiated.
- We can use **abstract** keyword to create an abstract method, an abstract method doesn't have body.
- If a class have abstract methods, then the class should also be abstract using abstract keyword, else it will not compile.
- It's not necessary for an abstract class to have abstract method. We can mark a class as abstract even if it doesn't declare any abstract methods.

# Abstract Classes

- If abstract class doesn't have any method implementation, its better to use interface because java doesn't support multiple class inheritance.
- The subclass of abstract class in java must implement all the abstract methods unless the subclass is also an abstract class.
- All the methods in an interface are implicitly abstract unless the interface methods are static or default. Static methods and default methods in interfaces are added in Java 8, for more details read [Java 8 interface changes](#).
- Java Abstract class can implement interfaces without even providing the implementation of interface methods.

# Abstract Classes

Here is an Example:

```
1 public abstract class Car {  
2     public void accelerate(){  
3         System.out.println("Accelerating!");  
4     }  
5     // this is an abstract method, and it cannot have a body.  
6     public abstract void slow();  
7 }
```

If we want to create a subclass of the **Car** abstract class, then we have to implement all abstract methods of the **Car** class.(`slow()` method for our example). So let's create a subclass of the **Car**.

# Abstract Classes

Let's create a subclass of the Car class:

```
1 public class Sedan extends Car{
2     @Override
3     public void accelerate(){
4         System.out.println("Test");
5         super.accelerate();
6     }
7
8     @Override
9     public void slow() {
10        System.out.println("Sedan car is slowing!");
11    }
12 }
```

# Interfaces

Interface in java is one of the core concept. Java Interface is core part of java programming language and used a lot not only in JDK but also java design patterns. Most of the frameworks use java interface heavily. Interface in java provide a way to achieve abstraction. Java interface is also used to define the contract for the subclasses to implement. An interface is a fully abstract class. It includes a group of abstract methods (methods without a body). We use the **interface** keyword to create an interface in Java.

## Example:

```
1 interface OperateCar{
2     public void turn(Direction direction,
3         double radius,
4         double startSpeed,
5         double endSpeed);
6     public void accelerate(double increaseSpeed);
7     public void brake(double decreaseSpeed);
8     //more methods...
9 }
```

# Implementing Interfaces

In our example, OperateCar is an interface, and turn, accelerate and breake methods are abstract methods.

Like abstract classes, we cannot create objects of interfaces. To use an interface, other classes must implement it. We use the **implements** keyword to implement an interface. We will implement the example which is in the previous slide, in our IDE.



# Important Points about Interfaces

- **interface** is the code that is used to create an interface in java.
- We can't instantiate an **interface** in java.
- Interface provides absolute abstraction, in last post we learned about abstract classes in java to provide abstraction but abstract classes can have method implementations but interface can't.
- Interfaces can't have constructors because we can't instantiate them and interfaces can't have a method with body.
- An interface can extend other interfaces, just as a class subclass or extend another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

# Important Points about Interfaces

- The **interface** body can contain abstract methods, **default** methods, and **static** methods. An abstract method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation). Default methods are defined with the **default** modifier, and **static** methods with the **static** keyword.
- All **abstract**, **default**, and **static** methods in an interface are implicitly public, so you can omit the **public** modifier. In addition, an interface can contain constant declarations. All constant values defined in an interface are implicitly public, static, and final. Once again, you can omit these modifiers.

# default Methods in Interface

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

## Why default methods?

Let's take a scenario to understand why default methods are introduced in Java.

Suppose, we need to add a new method in an interface. We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method. If a large number of classes were implementing this interface, we need to track all these classes and make changes to them. This is not only tedious but error-prone as well. To resolve this, Java introduced default methods. Default methods are inherited like ordinary methods.

# private and static Methods in Interface

- The Java 8 also added another feature to include **static** methods inside an interface. Similar to a class, we can access static methods of an interface using its references.
- With the release of Java 9, **private** methods are also supported in interfaces. We cannot create objects of an interface. Hence, **private** methods are used as helper methods that provide support to other methods in interfaces.

# Abstract Classes vs Interfaces

With the introduction of concrete methods (default and static methods) to interfaces from Java 8, the gap between interface and abstract class has been reduced significantly. Now both can have concrete methods as well as abstract methods. But, still there exist some minute differences between them.

## Differences Between Interface And Abstract Class After Java 8:

### 1) Fields

Interface fields are **public**, **static** and **final** by default. Interfaces still don't support non-static and non-final variables. Interfaces can only have public, static and final variables. On the other hand, abstract class can have static as well as non-static and final as well as non-final variables. They also support private and protected variables along with public variables.

---

Further reading: <https://javaconceptsoftheday.com/interface-vs-abstract-class-after-java-8/>

# Abstract Classes vs Interfaces

## 2) Methods

After Java 8, an interface can have **default** and **static** methods along with abstract methods.

Interfaces don't support **final** methods. But, abstract classes support final as well as non-final methods and static as well as non-static methods along with abstract methods. Also note that, only interfaces can have default methods. Abstract classes can't have default methods.

## 3) Constructors

Interfaces can't have constructors. Abstract classes can have any number of constructors.

---

Further reading: <https://javaconceptsoftheday.com/interface-vs-abstract-class-after-java-8/>

# Abstract Classes vs Interfaces

## 4) Multiple Inheritance

A class can extend only one abstract class(because of the diamond problem), but can implement multiple interfaces. Thus, a class can inherit multiple properties from multiple sources only through interfaces, not through abstract classes.

---

Further reading: <https://javaconceptsoftheday.com/interface-vs-abstract-class-after-java-8/>

# Encapsulation

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of fields and methods inside a single class. It prevents outer classes from accessing and changing fields and methods of a class. This also helps to achieve data hiding. In other words, wrapping data and methods within classes in combination with implementation hiding (through access control) is often called encapsulation. The result is a data type with characteristics and behaviors. Encapsulation essentially has both i.e. information hiding and implementation hiding.



# Encapsulation

## Example:

```
1  class Person {  
2  
3      // private field  
4      private int age;  
5  
6      // getter method  
7      public int getAge() {  
8          return age;  
9      }  
10  
11     // setter method  
12     public void setAge(int age) {  
13         this.age = age;  
14     }  
15 }
```

private fields is not accessible from outside of the Person class, so we hide the variable `age`. We can only get its value by using get method.

# Encapsulation vs Abstraction

Abstraction is more about ‘What’ a class can do. Encapsulation is more about ‘How’ to achieve that functionality. For more details you can examine [OOps - Encapsulation vs Abstraction](#) especially “Encapsulation vs Abstraction” section.

# Polymorphism

Polymorphism is an important concept of object-oriented programming. It simply means more than one form. That is, the same entity (method or operator or object) can perform different operations in different scenarios. Polymorphism allows us to create consistent code. We can achieve polymorphism in Java using the following ways:

- Method Overriding
- Method Overloading
- Operator Overloading

---

Further reading: <https://javaconceptsoftheday.com/interface-vs-abstract-class-after-java-8/>

# Polymorphism

## Example:

```
1 class Polygon {
2     public void render() {
3         System.out.println("Rendering Polygon...");
4     }
5 }
6
7 class Square extends Polygon {
8     @Override
9     public void render() {
10        System.out.println("Rendering Square...");
11    }
12 }
13
14 class Circle extends Polygon {
15     @Override
16     public void render() {
17        System.out.println("Rendering Circle...");
18    }
19 }
```

# Polymorphism

## Example(continuing):

```
1  class Main {
2      public static void main(String[] args) {
3
4          // create an object of Square
5          Square s1 = new Square();
6          s1.render();
7
8          // create an object of Circle
9          Circle c1 = new Circle();
10         c1.render();
11     }
12 }
```

In the example above, we have created a superclass: Polygon and two subclasses: Square and Circle. Notice the use of the render() method. Hence, the render() method behaves differently in different classes. Or, we can say render() is polymorphic.



# Practice Time: Let's Code Together!

## Exercise:

Define an Employee interface that has several methods: calculateSalary() , getSalary(). Then define concrete classes which are Employer, Supervisor, BlueCollar, WhiteCollar. Finally create instances of all these concrete classes and get and print out their salaries.

Notes: You can calculate salary by using this formula: (total work days \* dailySalaryAmount \* employeeConstant). Each of this constants in concrete classes would be different from each other.



# Questions ?





## Next: Java Collections Framework