



LEARN PROGRAMMING IN JAVA™

Learning OO Java™ as a Beginner – Interfaces

COURSE AGENDA

- What is an Object
 - The Basics
 - Operators
 - Control Flow
 - Implementation Hiding
 - Reuse
 - Interfaces
- Collections
 - Functional Programming
 - Stream
 - Exceptions
 - Generics
 - Arrays



INSTRUCTOR – HUGO SCAVINO

- 30 Years of IT Experience
- Using Java since the beta
- Taught Java and OOP in USA, UK, and France to Fortune 500
- Senior Software Architect with
 - DTCC
 - Penske
 - HSBC
 - Government Agencies



<https://www.linkedin.com/in/hugoscavino/>

REQUIRED SOFTWARE

- JDK™
 - JDK 8 or Newer (Open Source, Amazon, or Oracle)
 - The beginner course focuses on core language constructs; feel free to use 11, 17, 21, etc.
- IDE
 - IntelliJ Community Edition (Free) or Enterprise (Paid)
 - While you can use any IDE with Java, the course and labs are explicitly made with IntelliJ in mind



RECOMMENDED TEXTS



- Java 5 Book
 - [Thinking in Java – 4th Edition - Free – PDF - GitHub](#)
 - [Thinking in Java – 4th Edition – Hard Cover - Amazon](#)
- Recommended Java 8 Book
 - [Bruce Eckel on Java 8](#)
 - Contains references to newer Java syntax
- #1 Best Seller in Beginner's Guide to Java Programming
 - [Head First Java: A Brain-Friendly Guide 3rd Edition](#)



ONLINE VIDEO TUTORIAL

- Learning Programming In Java
 - [YouTube Tutorials](#)
- Lab Exercises
 - <http://www.learnprogramminginjava.com/>





TOPIC DESCRIPTION

Interfaces is a reference type, like a class, that is used to define a contract for classes that implement it. An interface specifies a set of methods that a class must implement, but it does not provide the implementation of these methods.

Instead, the implementing class is responsible for providing the actual code.

Interfaces are widely used in Java to achieve abstraction and to support multiple inheritance, as Java classes cannot inherit from more than one superclass.

TOPIC DESCRIPTION

Method Declarations Only: contain method declarations (method signatures) without a body. By default, methods in an interface are abstract and public.

Constants: You can declare variables in an interface, but they are implicitly public, static, and final. These are effectively constants.

Multiple Implementation: A class can implement multiple interfaces, which helps overcome the limitation of single inheritance in Java.

Default and Static Methods: interfaces can have default methods (with a default implementation) and static methods. Allows adding new functionality to interfaces without breaking existing implementations.

Functional Interfaces: Functional interfaces (interfaces with exactly one abstract method) were introduced, enabling the use of lambda expressions. Include Runnable, Callable, and Comparator.

INTERFACES EXAMPLE

Interface

```
public interface Speakable { 2 usages 2 implementations

    // No need to be public
    boolean isVeryLoud = true; no usages

    // No need to add public nor abstract
    void speak(); no usages 2 implementations new *
}
```

Implementation

```
public class Dog extends Mammal implements Speakable {

    // Note the annotation for @Override
    @Override no usages new *
    public void speak() {
        System.out.println("Woof");
    }

    public class Person implements Speakable{ no usages

        @Override no usages new *
        public void speak() {
            System.out.println("Hello World!");
        }
    }
}
```

Interfaces are essential particularly when designing frameworks and large systems

Interfaces provide a way to specify what a class should do without dictating how it should do it.

Common interface amongst unrelated classes

BENEFITS OF INTERFACES

Benefits of Using Interfaces

Decoupling: Interfaces provide a way to define methods that unrelated classes can implement, allowing flexibility and loose coupling.

Multiple Inheritance: Java does not support multiple inheritance with classes, but interfaces allow a class to inherit behaviors from multiple sources.

Polymorphism: Interfaces allow for polymorphic behavior, meaning you can write code that works on objects of classes implementing the same interface, even if they are not related by inheritance.

WHAT ARE ABSTRACT CLASSES

Is a class that cannot be instantiated directly and is used as a base for other classes to inherit from.

Abstract classes are designed to be inherited by other classes, which then provide the implementation for any abstract methods (methods without a body) defined in the abstract class.

Useful for creating a common template **with some shared code** while leaving certain methods to be defined by subclasses.

ABSTRACT CLASS KEY POINTS

- **Cannot Be Instantiated:** Java cannot create an object directly from an abstract class. They can only be subclassed.
- **Abstract Methods:** Abstract classes can contain abstract methods (methods without a body), which must be implemented by any concrete (non-abstract) subclass.
- **Concrete Methods:** Unlike interfaces, abstract classes can also contain concrete methods (methods with an implementation). This is useful for providing some common behavior that all subclasses can inherit without needing to redefine.
- **Constructors:** Abstract classes can have constructors, which are used when a subclass instantiates and initializes inherited fields.

ABSTRACT CLASS KEY POINTS

- **Fields and Properties:** Abstract classes can define fields (variables), and these fields can have any access modifier (private, protected, public), **unlike in interfaces where fields are always public, static, and final.**
- **When to Use Abstract Classes:** When a base class that should not be instantiated on its own, but instead provides a common foundation for multiple subclasses.
- If you need to define some common code across related classes but leave certain methods for the subclasses to implement, use an abstract class.

WHAT ARE THE DIFFERENCES?

Feature	Abstract Class	Interface
Instantiation	Cannot be instantiated	Cannot be instantiated
Method Implementation	Can have both abstract and concrete methods	Methods are abstract by default
Fields	Can have instance variables	Only constants (public, static, final)
Inheritance	A class can inherit only one abstract class	A class can implement multiple interfaces
Use Case	When classes are closely related and share common code	When unrelated classes need to implement the same functionality

EXAMPLES OF BOTH

```
public interface Speakable { 2 usages 2 implementations

    // No need to be public
    boolean isVeryLoud = true; no usages

    // No need to add public nor abstract
    void speak(); no usages 2 implementations new *
}
```

```
public abstract class Mammal { no usages 1 inheritor new *

    // Fields
    public int NUM_LIMBS = 4; no usages

    // There is no implementation
    abstract void giveMilk(); no usages 1 implementation new *
}
```

```
public class Dog extends Mammal implements Speakable { r

    // Note the annotation for @Override
    @Override no usages new *
    public void speak() {
        System.out.println("Woof");
    }

    @Override no usages new *
    void giveMilk() {
        System.out.println("Mother's give milk");
    }
}
```

DEFAULT METHODS - INTERFACE

- Default methods in Java interfaces are methods with a body, introduced in Java 8 to allow interfaces to have implemented methods.
- This was a significant change to Java's design, as previously, interfaces could only contain abstract method declarations.
- Default methods enable developers to add new methods to interfaces without breaking existing implementations of the interface, supporting backward compatibility.
- Huh?

DEFAULT METHODS – KEY POINTS

- **Method Implementation:** Default methods have an implementation (method body) within the interface. They use the default keyword in their declaration.
- **Backward Compatibility:** They were introduced to allow new methods in interfaces without forcing all implementing classes to provide an implementation, which would otherwise break backward compatibility.
- **Access in Implementing Classes:** Classes implementing the interface can either use the default implementation or override the default method with their own implementation.
- **Difference from Abstract Classes:** Unlike abstract classes, interfaces with default methods still cannot have instance fields, and they are primarily used to add behavior rather than state.

DEFAULT METHODS – EXAMPLE

```
public interface Speakable { 2 usages 2 implementations new *  
  
    // No need to be public  
    boolean isVeryLoud = true; no usages  
  
    int volumeLevel = 100; no usages  
  
    static void calculateDecibels(){ no usages new *  
        System.out.println("Calculate the volume in decibels");  
    }  
  
    // No need to add public nor abstract  
    void speak(); no usages 2 implementations new *  
  
    // note the `default` key word and implementation!  
    default void mute(){ no usages new *  
        System.out.println("mute the speaker!");  
    }  
}
```

Method Implementation:

Default methods have an implementation (method body) within the interface. They use the default keyword in their declaration.

Those implementing the interface can override the behavior.



STATIC METHODS – INTERFACES

- **Static methods** in Java interfaces are methods with a body that belong to the interface itself rather than to any instance of the implementing class.
- Introduced in Java 8, static methods in interfaces provide utility functions related to the interface, like static methods in regular classes.


STATIC METHODS – KEY POINTS

Key Points About Static Methods in Interfaces

- **Interface-Level Utility:** Static methods are typically used for utility or helper functions that pertain to the interface's behavior but do not depend on any instance of the implementing class.
- **Direct Access:** Unlike default methods, **static methods cannot be inherited or overridden by implementing classes**. They are called on the interface itself rather than on instances of the implementing classes.
- **Accessing Static Methods:** They are accessed using `InterfaceName.methodName()` syntax, like accessing static methods in classes.

STATIC METHODS – EXAMPLE

```
public interface Speakable { 2 usages 2 implementations new *  
  
    // No need to be public  
    boolean isVeryLoud = true; no usages  
  
    int volumeLevel = 100; no usages  
  
    static void calculateDecibels(){ no usages new *  
        System.out.println("Calculate the volume in decibels");  
    }
```

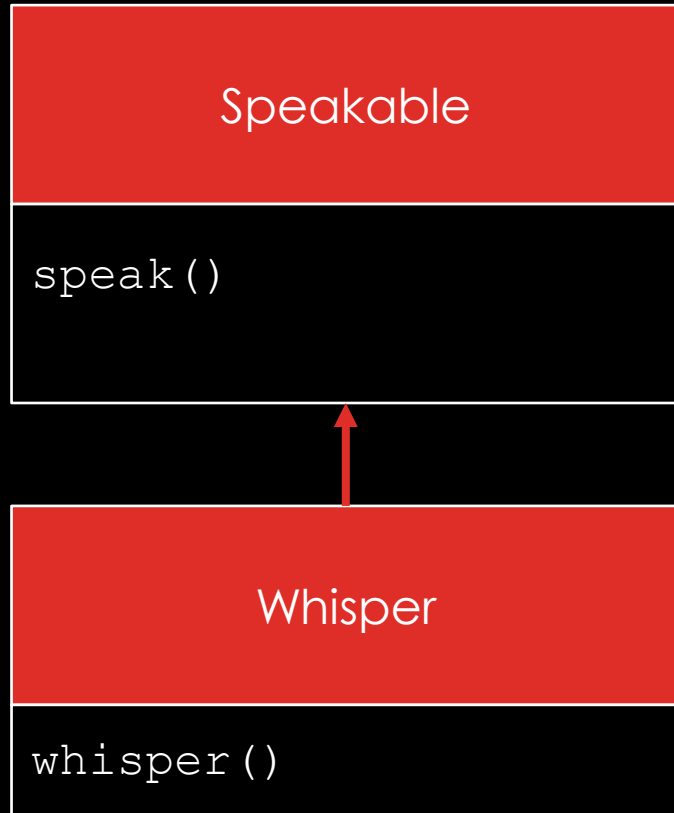


- **Interface-Level Utility:** Static methods are typically used for utility or helper functions that pertain to the interface's behavior but do not depend on any instance of the implementing class.

WHAT ARE THE DIFFERENCES?

Feature	Static Method	Default Method
Access	Accessed via the interface name	Accessed through instances
Overriding	Cannot be overridden by implementing classes	Can be overridden by implementing classes
Purpose	Utility or helper methods related to the interface	Default behavior for implementing classes

WHAT IS INHERITANCE?



Interfaces are usually few methods

The `Whisper` Interface consists of all the functions for a `Speakable`.

An interface can inherit from another interface using the `extends` keyword, like how a class inherits from a superclass.

This allows one interface to build upon another, adding more abstract methods or default methods, and forming a hierarchy of interfaces.

INHERITANCE FEATURES

Method Inheritance: When one interface extends another, it inherits all the methods from the parent interface. This includes abstract methods, default methods, and static methods.

Multiple Inheritance: An interface can extend multiple interfaces, allowing it to combine functionality from multiple sources.

Adding New Methods: The child interface can add new abstract methods or default methods in addition to those inherited from the parent interface(s).

No Implementation Requirement: When a class implements an interface that extends another interface, it must implement all abstract methods from the entire hierarchy unless they are default methods or already implemented.

INHERITANCE EXAMPLE

```
public interface Whisper extends Speakable {  
  
    void whisper();  
}
```

Benefits of Interface Inheritance

Code Reusability: Allows common behavior definitions to be shared across related interfaces.

Type Polymorphism: Enables treating classes implementing extended interfaces as instances of all parent interfaces, enhancing flexibility and abstraction.

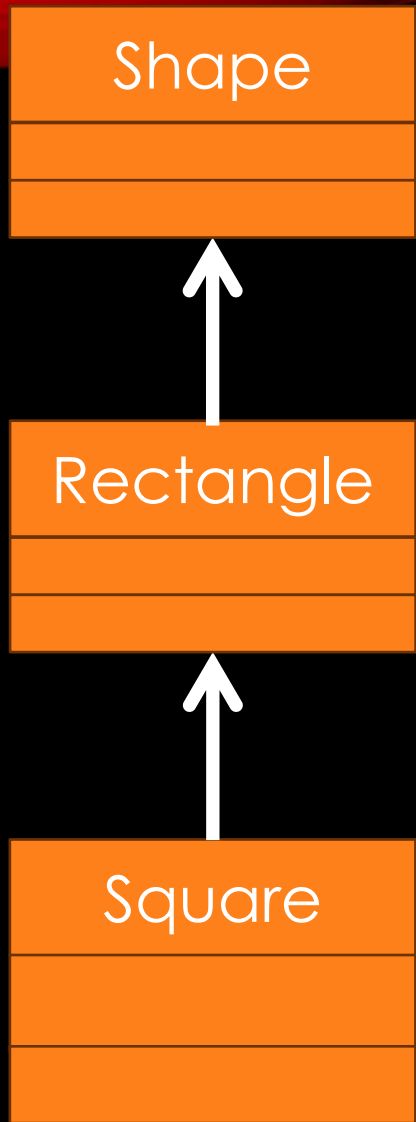
Modularity: Enables designing modular, reusable APIs where functionality can be progressively added through interface hierarchies.

TOPIC SUMMARY – INTERFACES ALWAYS?

Preference? Do we always prefer and create interfaces to submit to a popular design pattern? Do we always create an interface when designing classes. Why? Is it premature to start designing interfaces?

```
public class Dog extends Mammal implements Speakable {  
  
    // Note the annotation for @Override  
    @Override no usages new *  
    public void speak() {  
        System.out.println("Woof");  
    }  
  
    @Override no usages new *  
    void giveMilk() {  
        System.out.println("Mother's give milk");  
    }  
}
```





TOPIC SUMMARY - INHERITANCE

Inheritance: a fundamental object-oriented programming (OOP) principle allowing a new class (a subclass) to inherit properties and behaviors (methods) from an existing class (called a superclass or base class).

Referred to as an "is-a" relationship, where the subclass is a more specific version of the superclass. E.g. A square "is-a" rectangle, which in turn "is-a" Shape



TOPIC SUMMARY - UPCASTING

```
public class ShapeManager { new *  
    public void draw(Rectangle rectangle) {  
  
Rectangle r1 = new Rectangle( width: 4, height: 4);  
Square s1 = new Square( length: 10, height: 2);  
  
Rectangle[] rectangles = {r1,s1};  
sm.draw(rectangles);
```

- The ShapeManager class has a draw method that takes in an array of Rectangle classes, which is a base class
- We can create an array of Rectangles
 - One Rectangle
 - One Square
- The draw method only needs what's in the Rectangle interface and a Square will suffice because a Square is a Rectangle.

FINAL FIELDS, METHODS, ARGUMENTS, CLASSES

final field is a variable declared with the **final** keyword, which makes it a constant after it's initialized. Once a final field is assigned a value, that value cannot be changed, commonly used to create constants and ensure immutability.

If a **final method argument** means the method will not change the reference to the object, but the contents and fields may change.

final Methods prevent Modification. Final methods ensure that the method's functionality remains unchanged in any subclasses

final classes are declared with the final keyword. They cannot be subclassed. Declaring a class as final is useful when you want to prevent subclasses.



ONLINE VIDEOS

- Learning Programming In Java
 - [YouTube Tutorials](#)
- Lab Exercises
 - <http://www.learnprogramminginjava.com/>

