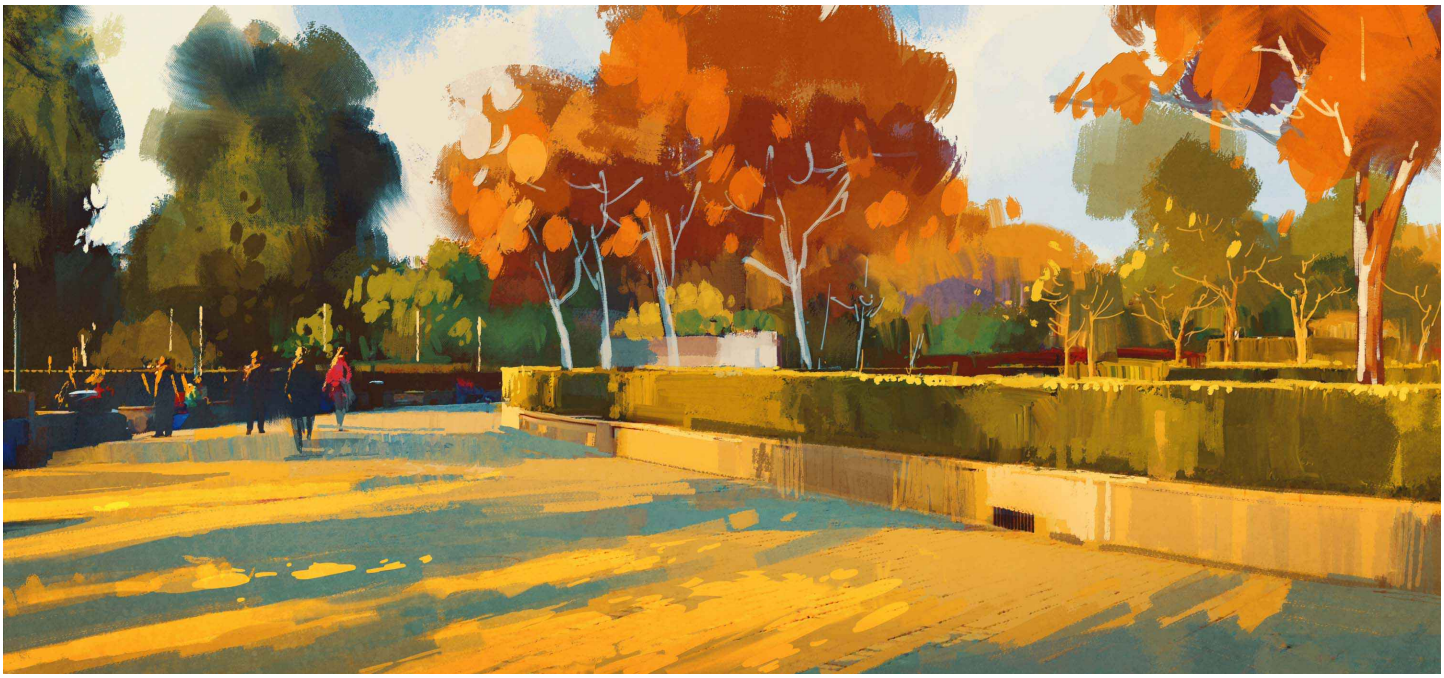


16-Abstraction

Author: Vincent Lau

Learning Objectives

> Understand the benefits of interfaces
>
> List the features of interfaces
>
> Define and implement an interface
>
> Use interfaces to create flexible code

Introduction

- In software engineering, a group of programmers **(method creators)** agree to a **contract** regarding how their applications interact. Another group of programmers **(method callers)** can write their code without any knowledge of how the other group's code is written. ***Interfaces* are such contracts**.

- In Java, an `interface` is a reference type, similar to a class. An interface may contain:
  - Constants (static final)
  - Method signatures (no implemention)
  - Default & Static methods (after Java 8)
  - Nested types (e.g. Enum, in the later chapter)
- **No Instance Variable**
- Same as abstract class, Interfaces **cannot be instantiated** - they can only be *implemented* by classes or *extended* by other interfaces.

Act as Contract & 100% Abstraction

- Objects define their interaction with the outside world through the methods they expose.
- Methods form the object's *interface* with the outside world. For example, the common buttons (play, fast forward/rewind, increase/decrease volume) on a television remote control form an interface for users to interact with.
- *The interface* **reflects 100% abstraction** in Java (Before Java 8).
- *Interfaces* form a `contract` between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile. This is the `rule` when we are using interface for design technically.
- While developers enjoy the benefit of **hiding the implementation details** behind the interface, **they also bear the responsibility to keep the interface stable** so that users can rely upon it safely.
- In programming, implementing an interface allows a class to become more formal about the behavior it promises to provide.

```
1 public interface Vehicle {
2     // NO instance variable
3     // behaviors to describe Vehicle
4     // a contract to define all behaviors
5     void start();
6     void stop();
7     void accelerate();
8     void brake();
9 }
```

- To implement this interface, you'd use the ***implements*** keyword in the class declaration.

- **Car & Bike are the implementations of the interface of Vehicle**. Both of them have the same behaviors, which are written in the contract Vehicle.
- Car can be with method(s) that Bike does not have.

```
public class Car implements Vehicle {
    void start () { //code here…
    }
    void stop () { //code here…
    }
    void accelerate () { //code here…
    }
    void brake () { //code here…
    }
}

public class Bike implements Vehicle {
    void start () { //code here…
    }
    void stop () { //code here…
    }
    void accelerate () { //code here…
    }
    void brake () { //code here…
    }
}
```

After Java 8

After Java 8, although *default* method with implementation is allowed in Interface, the initial objective of *default* method is NOT to violate 100% abstraction

The primary reasons for introducing default methods in interfaces were:

1. **Backward Compatibility**: Existing interfaces could be enhanced with new methods using default methods without breaking the implementation classes. Classes that implemented these interfaces would automatically inherit the default method's implementation without any need for modification.
2. **Code Reusability**: Default methods allowed for the reuse of common code across multiple classes that implement the same interface. This reduced code duplication and improved maintainability.

3. **Gradual Migration**: Default methods helped with the gradual migration of code from **abstract classes to interfaces**. With default methods, some of the functionality previously provided by abstract classes could be moved to interfaces.

Defining an Interface

```java
public interface FarmAnimal extends Animal, FarmItem, SomeOtherInterface {
    String FARM_ANIMAL_NAME_PREFIX = "Old McDonald"; // Constant, implicitly

    void makeSound();

    default void sleep() { // after Java 8
        System.out.println("Zzzzzzzzzzz");
    }

    static boolean isNameValid(String name) { // after Java 8
        return name.contains(FARM_ANIMAL_NAME_PREFIX);
    }
}

class FinalClass implements FarmAnimal {
    // all the methods in Animal, FarmItem, SomeOtherInterface
    // makeSound() {...}
    // sleep() {...} (Override)

    // main()
    // FinalClass finalclass = new FinalClass();
    // finalClass.sleep()
    // finalClass.makeSound()
    // FarmAnimal.isNameValid()
}
```

Interface Body

- The ***public*** access modifier that comes before the *interface* keyword indicates the interface can be used by any classes in any package.
- If you do not specify, then the default access level is ***package-private***, which means it is accessible only to classes defined in the same package.

Methods - Abstract, Default & Static

- The interface body can contain:

- ***Abstract*** methods (e.g. *makeSound*) have **no braces or method body, only a semicolon**.
- ***Default*** methods (e.g. *sleep)*
  - Default method is for its implementation class to "inherit". For instance/object to use.
- ***Static*** methods (e.g. *isNameValid)*
  - Static method is a tool which belongs to Interface itself ONLY. Static methods can not be inherited by implementation class.

```java
public interface MyInterface {
    static void staticMethod() {
        System.out.println("Static method in interface");
    }
}

public class MyClass implements MyInterface {
    public static void main(String[] args) {
        // MyClass.staticMethod(); // This is invalid
        MyInterface.staticMethod(); // Correct way to call the static method
    }
}
```

Implicit modifiers of Interface

- All members (fields and methods) in an interface are implicitly *public* (which is the whole *purpose* of an interface).
- Fields are **implicitly** *static* and *final,* also known as Constants.
- Method bodies exist only for **default** methods & **static** methods.
- Static methods are similar to default methods but the only difference is that you **can’ t override static methods**.
- Methods in an interface that are not declared as default or static are *implicitly abstract*, so the ***abstract*** **modifier is optional**.
- Default & static has to be explicitly defined.

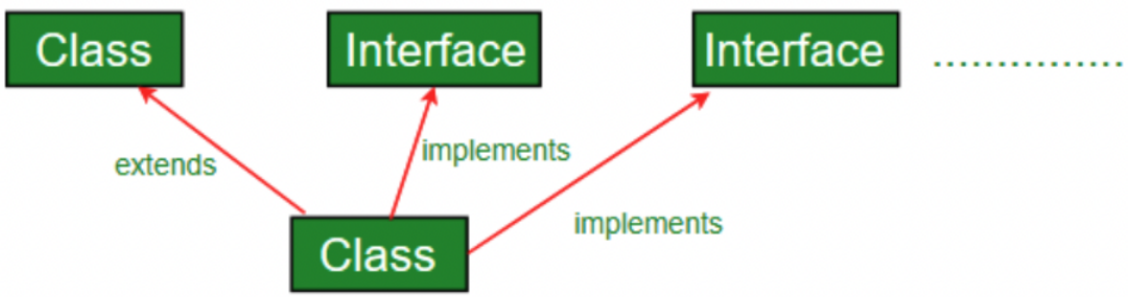
```java
public interface InterfaceWithAbstractMethods {
    public static final int someValue = 10;

    // Implicitly abstract, if it is not default or static
    // equals to "void doSomething();"
    public abstract void doSomething();
```

```
7
8       public default void doSomethingElse() {
9           System.out.println("do something else");
10      }
11
12      public static void doSomethingStatic() {
13          System.out.println("do something else");
14      }
15  }
```

As a result, interface should be defined in this way:

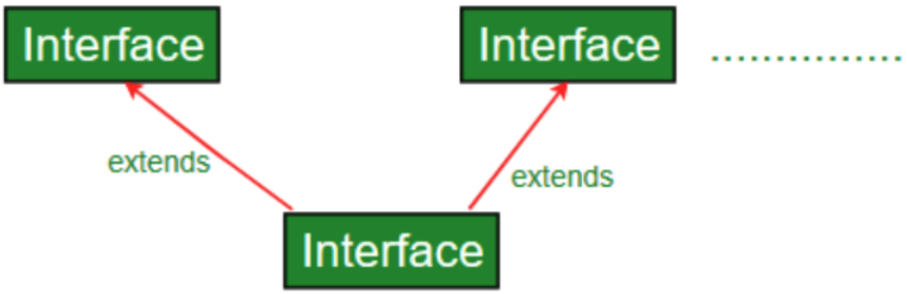
```
1  public interface MyInterface {
2
3      int VOLUME = 5; // Constants, implicitly static final
4
5      //default method
6      default void newMethod() { // implicitly public
7          System.out.println("Hello, this is default method");
8      }
9      // Abstract method
10     void existingMethod(String msg); // implicitly public & abstract
11     // static method
12     static void sayLouder(String msg) { // implicitly public
13         System.out.println(msg);
14     }
15 }
```

Implementing an Interface



- A class implements an interface by using the ***implements*** keyword.
- Your class can implement more than one interface, so the *implements* keyword is followed by a comma-separated list of the interfaces implemented by the class.

Extending an Interface



- An interface extends an interface by using the ***extends*** keyword.
- An interface can extend more than one interface, so the ***extends*** keyword is followed by a comma-separated list of the interfaces ***extends*** by the class

Why doesn't Java allow multiple inheritance, but multiple interfaces?

- In Java, a class is only allowed to extend one direct superclass, whereas a class is allowed to implement multiple interfaces.
- The reason is simple - imagine if a class is allowed to extend more than one class, it will have the potential to **inherit fields or methods with the same name from different superclasses**. This will cause **name conflicts** and **ambiguity,** also known as the ***Diamond Problem.***

```
1  public class ClassA {
2      public void doSomething() {
3          System.out.println("[ClassA] do something from class A...");
4      }
5  }
6
7  public class ClassB {
8      public void doSomething() {
9          System.out.println("[ClassB] do something from class B...");
10     }
11 }
12
13 /*
14 If a class is allowed to extend both ClassA and ClassB,
15 it will inherit the *doSomething* method from both classes
16 and leave the compiler confused which method to use at runtime.
17 */
```

- The reason why the same problem does not occur when a class implements multiple interfaces is because a class **is forced to provide its own implementation** when name conflicts or ambiguity occurs, otherwise the compiler will not let the class compile successfully [For default method in interface].

```java
public interface InterfaceA {
    // after Java 8
    default void doSomethingElse() {
        System.out.println("[InterfaceA] do something else...");
    }
}

public interface InterfaceB {
    // after Java 8
    default void doSomethingElse() {
        System.out.println("[InterfaceB] do something else...");
    }
}

// ClassC is forced to provide its own implementation
public class ClassC implements InterfaceA, InterfaceB{
    @Override
    public void doSomethingElse() {
        System.out.println("[ClassC] do something else...");
    }
}

/*
ClassC will not compile unless it provides its own implementation of doSomethingElse,
hence the diamond problem does not happen when implementing multiple interfaces.
*/
```

Precedence - Interface & Inheritance Methods

- A subclass inherits **default** and **abstract** methods from interfaces just like how it inherits instance methods from an abstract class. When superclasses or upstream interfaces provide multiple default methods with the same signature, the Java compiler will follow two principles to resolve the name conflict:
    a. **Instance** methods are **preferred over interface default** methods.

b. Methods that are already overridden by closer ancestors **take precedence**.

```
public interface Flyer {
    default public String speak() {
        return "I am able to fly.";
    }
}

public interface Runner {
    default public String speak() {
        return "I am able to run";
    }
}

// Instance methods are preferred over interface default methods
public class Horse {
    public String speak() {
        return "I am able to speak.";
    }
}

public class Unicorn extends Horse implements Flyer, Runner {
    public static void main(String... args) {
        Unicorn unicorn = new Unicorn();
        System.out.println(unicorn.speak());
    }
}

// Output: I am able to speak
```

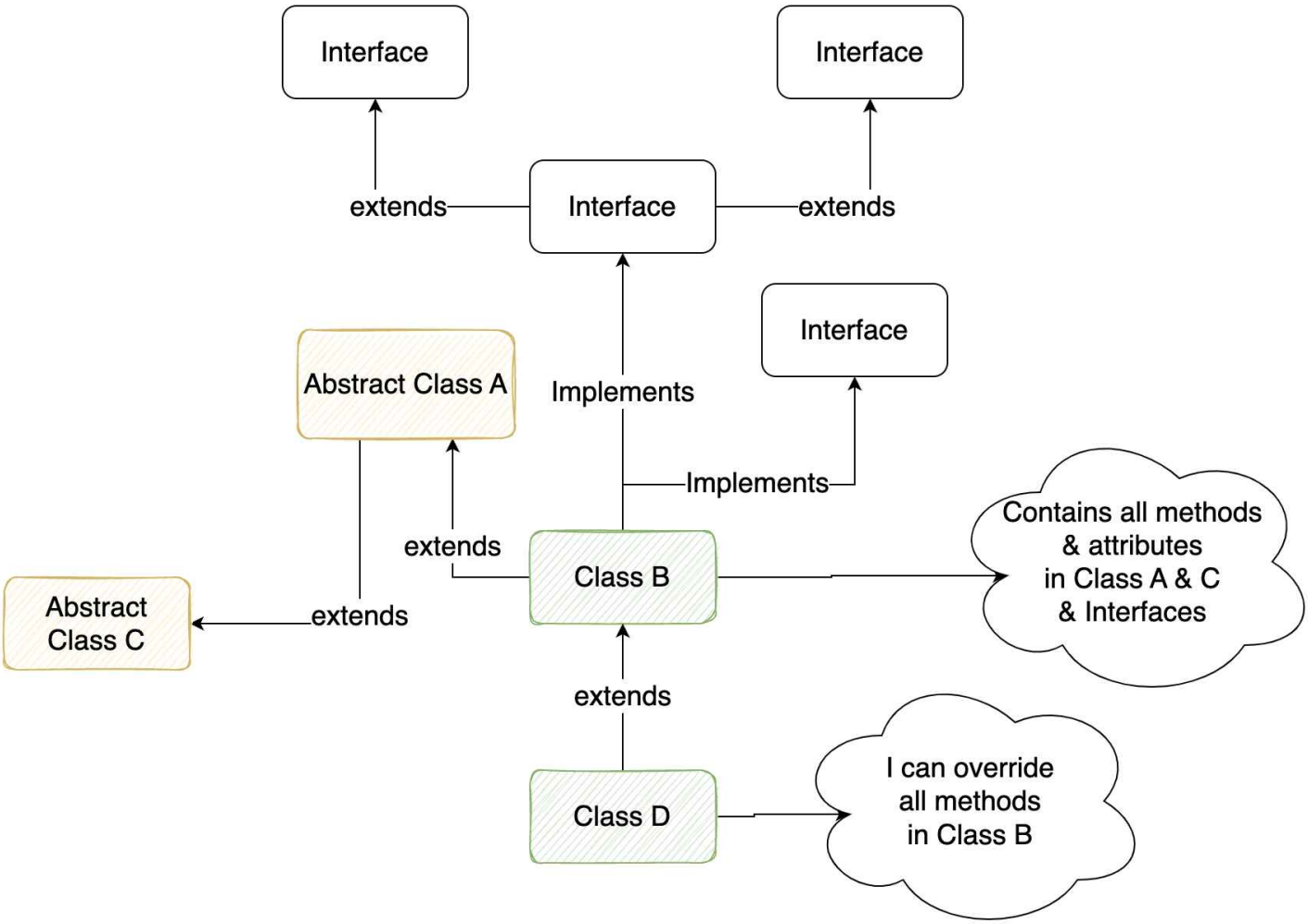
Interface Examples

```
// Interface FarmAnimal extend more than one interface
public interface Animal {
    void move();
}

public interface SustainLife {
    void eat();
    void breath();
}

public interface FarmAnimal extends Animal, SustainLife {
```

```
12      String FARM_ANIMAL_NAME_PREFIX = "Old McDonald"; // implicitly static final
13
14      void makeSound();
15
16      default void sleep() {
17          System.out.println("Zzzzzzzzzzz");
18      }
19
20      static boolean isNameValid(String name) {
21          return name.contains(FARM_ANIMAL_NAME_PREFIX);
22      }
23  }
24
25  // By convention, the implements clause follows the extends clause, if there
    is one.
26  public class Pig extends SomeClass implements FarmAnimal {
27      private final String name;
28      // This class Pig has sleep method & the static method isNameValid()
29      public Pig(String name) {
30          if (!FarmAnimal.isNameValid(name)) {
31              throw new IllegalArgumentException("Invalid name");
32          }
33          this.name = name;
34      }
35
36      @Override
37      public void makeSound() {
38          System.out.println("Oink Oink");
39      }
40      // need implement move(), eat(), breath() also
41  }
```

Example Relationship - Inheritance & Interface

Abstract Class/ Interface/ Class



Questions

- Can abstract methods in an interface have method body? Why or why not?
- Can an interface have empty body?
- What is the difference between abstract methods, default methods and static methods in an interface?
- Why does Java not allow multiple inheritance?
- What are the differences between abstract classes and interfaces?