

Interfaces in Java

In Java, Interfaces provide a way to define a **contract** that a class must follow in order to implement a certain set of behaviors. The interface defines the methods that a class must implement, but does not provide any implementation details for those methods. It simply specifies the method signatures, return types, and parameter types.



Okay I understand what is an Interface ? But what are the advantages with it & why should we define interfaces ?
It's a great & valid question. Let's discuss...

Suppose you aim to generate various classes that portray IronMan, SpiderMan & CaptainAmerica - renowned superheroes from the Marvel Universe. To accomplish this task, you are collaborating with your peers, and the following is the result of your efforts.



IronMan

```
public class IronMan {  
  
    public void usePower() {  
  
    }  
  
    public String killVillain() {  
  
    }  
  
}
```



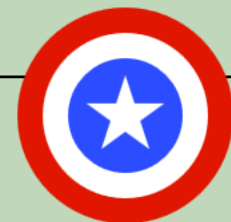
SpiderMan

```
public class SpiderMan {  
  
    public int applyPower() {  
  
    }  
  
    public void stopVillain(char x) {  
  
    }  
  
}
```

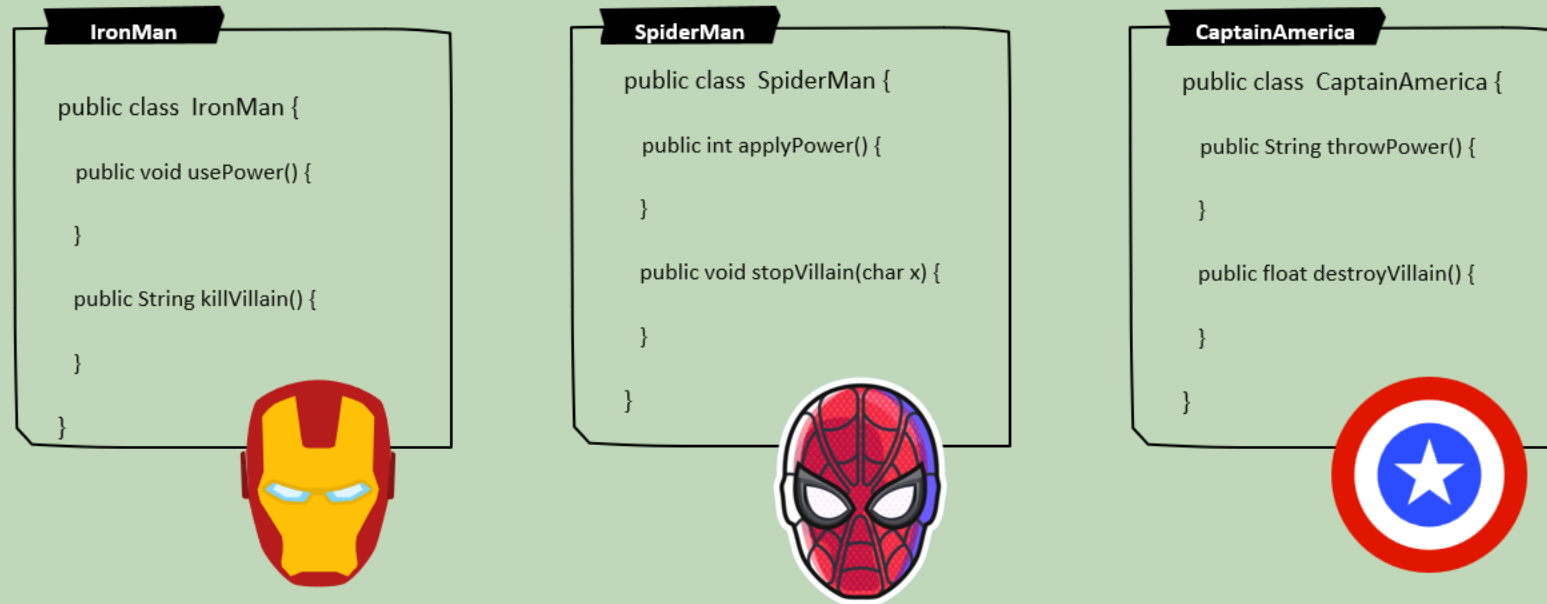


CaptainAmerica

```
public class CaptainAmerica {  
  
    public String throwPower() {  
  
    }  
  
    public float destroyVillain() {  
  
    }  
  
}
```



Problems with out Interfaces in Java



Each developer has written the classes, methods inside them based on their understanding. Though all these heroes have common functionality, the method names, their return types, accepting parameters are different. So there is no consistency



If you have to go for a war with Thanos, you may want to call all your super heroes and invoke their powers, issue commands to kill the villain. But with the above implementation, it is not a simple task as the method names, input & output are different.



To solve all these challenges, we have **interfaces** in Java. With the help of interfaces we can write maintainable, reusable and modular code. Interfaces provide many benefits like abstraction, flexibility, polymorphism, multiple inheritance, and separation of concerns

How to **create** an interface ?

Below is the syntax to declare an interface,

```
[access modifier] interface <interface-name> {  
    // Other interface code  
}
```

To create an interface in Java, we need to use the **interface** keyword followed by the name of the interface. An interface is a reference type, similar to a class, that can **contain static fields, abstract methods, default methods, private methods and static methods**. Method bodies exist only for default, private and static methods. Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces.

Here is an example of an interface called SuperHero that defines two abstract methods which every marvel super hero should have:

```
public interface SuperHero {  
    public static String final UNIVERSE_NAME = "MARVEL";  
    public String usePower();  
    public String stopVillain(char c);  
}
```

How to **create** an interface ?



An interface declaration is implicitly abstract. An interface declaration is considered abstract by default, regardless of whether it is explicitly marked with the "abstract" keyword. For example, below both interface declarations are same,

```
public abstract interface SuperHero {  
    // The interface body is empty  
}
```

```
public interface SuperHero {  
    // The interface body is empty  
}
```

Using the keyword abstract in interfaces declarations is obsolete, and it should not be used.



Classes and interfaces differ significantly. Classes can contain instance fields, while interfaces cannot. Furthermore, while you can create an object by instantiating a class, you cannot do so with an interface.



The body of an interface may consist of abstract, default, private and static methods. Abstract methods are identified by a semicolon instead of braces and lack an implementation. Default methods use the "default" modifier, while static methods use the "static" keyword. **The interface's abstract, default, and static methods are public by default, so the "public" modifier can be omitted.**

implementing a interface in Java



Below is the syntax for a class that implements an interface,:

```
[modifiers] class <class-Name> implements <comma-separated-list-of-interfaces> {  
    // Class body  
}
```



The class that implements an interface must override all abstract methods declared in the interface. Otherwise, the class must be declared as abstract class.



When a class implements interfaces, it can include additional methods that are not inherited from the implemented interfaces. These additional methods can share the same name but have different numbers and/or types of parameters compared to those declared in the implemented interfaces.



An interface can implement other interfaces and add more abstract methods. Below is the syntax,

```
[modifiers] interface <interface-name> extends <comma-separated-list-of-interfaces> {  
    // Class body  
}
```


implementing a interface in Java

To use an interface, you write a class that **implements** the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface.

IronMan

```
public class IronMan implements SuperHero {  
  
    public String usePower() {  
        //implementation code  
    }  
  
    public String stopVillain(char c) {  
        //implementation code  
    }  
  
}
```



SpiderMan

```
public class SpiderMan implements SuperHero {  
  
    public String usePower() {  
        //implementation code  
    }  
  
    public String stopVillain(char c) {  
        //implementation code  
    }  
  
}
```



CaptainAmerica

```
public class CaptainAmerica implements SuperHero {  
  
    public String usePower() {  
        //implementation code  
    }  
  
    public String stopVillain(char c) {  
        //implementation code  
    }  
  
}
```



Like demonstrated above, each class implements an interface and provides its own implementation code, while adhering to the interface's specified contract. Offcourse every hero will have their own style of using power & stopping the villian

Constant Field Declarations in **interface**



Like we discussed before, an interface can contain constant declarations. All constant values defined in an interface are implicitly **public, static, and final**. If needed you can omit these modifiers. For example, both of the below constant declarations are considered same,

```
public interface SuperHero {  
  
    public static final String UNIVERSE_NAME = "MARVEL";  
  
}
```

```
public interface SuperHero {  
  
    String UNIVERSE_NAME = "MARVEL";  
  
}
```



We can access the fields in an interface using **<interface-name>.<field-name>**. For example, we can use **SuperHero.UNIVERSE_NAME** to access the above constant.



Fields within an interface are inherently final, irrespective of whether the "final" keyword is explicitly stated in their declaration. Consequently, it is imperative to initialize a field at the point of declaration. Initialization can be achieved using a compile-time or runtime constant expression. Given that a final field, also known as a constant field, is assigned a value only once, the value of an interface's field cannot be modified except within its declaration.

```
String UNIVERSE_NAME; // Compilation fails as constant value is not initialized
```

Constant Field Declarations in **interface**



In interfaces, it's a common practice to name fields using all uppercase letters to signify that they are constants. However, Java doesn't enforce specific naming rules for interface fields as long as they adhere to standard identifier naming conventions.



Interface fields are always public, but their accessibility from outside the declaring package depends on the interface's scope. For instance, if an interface has package-level scope, its fields won't be accessible outside the package, even though the fields themselves are public.

Method Declarations in **interface**



A developer can create following four types of methods in an interface:

- ✓ Abstract methods
- ✓ Default methods (From Java 8)
- ✓ Static methods (From Java 8)
- ✓ Private methods (From Java 9)

Before Java 8, interfaces could only contain abstract methods. But from Java 8, the declaration of static, default methods and from Java 9 declaration of private methods within interfaces is allowed. If a method lacks any of these modifiers, it is considered abstract.

Before Java 8, we can implement abstraction like shown below percentages,

- **Abstract class (0 to 100%)**
- **Interface (100%)**

From Java 8, we can implement abstraction like shown below percentages,

- **Abstract class (0 to 100%)**
- **Interface (0 to 100%)**

abstract method declaration in interface



When you declare an interface in Java, the primary goal is to outline an abstract blueprint or concept. This is achieved by specifying zero or more abstract methods within the interface. **All method declarations inside an interface are inherently abstract and public, unless specified otherwise with static or default keywords.** Similar to abstract methods in a class, those in an interface lack an implementation. Instead of braces, the body of an abstract method is denoted by a simple semicolon.



In an interface, adding the abstract and public keywords when declaring a abstract method is unnecessary, even though the compiler allows it. This redundancy exists because, by default, a method in an interface is both abstract and public. For example, both of the below examples, will be considered as same

```
public interface Person {  
  
    public abstract void walk();  
    public abstract void sleep();  
  
}
```

```
public interface Person {  
  
    void walk();  
    void sleep();  
  
}
```



In an interface, abstract method declarations can include parameters, a return type, and a throws clause.



Classes implementing an interface inherit its abstract methods. These methods are meant to be overridden by the implementing classes to provide specific implementations. **It's important to note that an abstract method in an interface cannot be declared final because the final keyword implies that a method cannot be overridden.** However, the implementing class has the flexibility to declare the overridden method as final, signaling that its subclasses are not allowed to override it.

default methods in Interface

From Java 8

Think like, all of a sudden the marvel company decided to introduce a new specification/abstract method in SuperHero interface. But all the SuperHero implementation classes are not happy with this. Because it will create compilation errors complaining to implement the new abstract method. To overcome this problem, Java 8 introduced **default** methods in Java

SuperHero

```
public interface SuperHero {  
  
    public String usePower();  
  
    public String stopVillain(char c);  
  
    public String trackLiveLocation();  
  
}
```



In the realm of Java, numerous interfaces from libraries have been widely employed by countless users worldwide. Faced with the challenge of enhancing these interfaces without causing disruptions to existing code, Java designers sought solutions. The chosen remedy came in the form of default methods, allowing the addition of new methods to existing interfaces.

IronMan

```
public class IronMan implements SuperHero {  
  
    public String usePower() {  
        //implementation code  
    }  
  
    public String stopVillain(char c) {  
        //implementation code  
    }  
  
}
```



SpiderMan

```
public class SpiderMan implements SuperHero {  
  
    public String usePower() {  
        //implementation code  
    }  
  
    public String stopVillain(char c) {  
        //implementation code  
    }  
  
}
```



CaptainAmerica

```
public class CaptainAmerica implements SuperHero {  
  
    public String usePower() {  
        //implementation code  
    }  
  
    public String stopVillain(char c) {  
        //implementation code  
    }  
  
}
```





When Java team decided to provide a lot of features that support lambda, functional programming by updating the collections framework, they got a problem. If they add new abstract methods inside the interfaces, all the classes implementing them have to be updated as per the requirements, and this will affect all the teams using Java.

To overcome this and provide backward compatibility even after adding new features inside interfaces, the Java team allowed concrete method implementations inside interfaces which are called as **default methods**.



Prior to Java 8, when a new abstract method was added to an interface, all the classes that implemented that interface would be required to implement the new method, which could be a major hassle if there were a large number of classes implementing the interface.

With the introduction of default methods, we can now add new methods to an interface with a default implementation that can be used by all classes that implement the interface. This allows us to add new features to an interface without requiring any changes to the classes that implement it.

Default methods can also be used to provide a default behavior for methods that are not necessarily required to be implemented by all classes that implement the interface. This can make it easier to create more flexible and extensible interfaces.



Just like regular interface methods, default methods are also implicitly public. Unlike regular interface methods, default methods will be declared with **default** keyword at the beginning of the method signature along with the implementation code.

A default method cannot be declared abstract or static. It must provide an implementation.

default methods in Interface

Here is an example of a default method in an interface:

```
public interface SuperHero {  
  
    public String usePower();  
    public String stopVillain(char c);  
  
    public default String trackLiveLocation() {  
        String liveLocation = "USA";  
        System.out.println("I am in " + liveLocation);  
        return liveLocation;  
    }  
}
```

From Java 9

If there are several default methods in an interface, it is possible to create **private methods** accessible only by the default methods of the interface. They can be used to contain common functionality, instead of repeating it in every default method.

The private methods cannot be accessed from outside the interface.



Interface default methods are by default available to all the implementation classes. Based on their requirement, implementation class can use these default methods with default behavior or can override them.

We can't write default methods inside a class. Even in the scenarios where we are overriding the default keyword should not be used inside the class methods

The most typical use of default methods in interfaces is to incrementally provide additional features/enhancements to a given type without breaking down the implementing classes



static methods in Interface



From Java 8 just like how we can write default methods inside interfaces, we can also write **static** methods inside them to define any utility functionality. They are implicitly public

```
public interface SuperHero {  
  
    public String usePower();  
    public String stopVillain(char c);  
  
    public static String commonCharacteristics() {  
        return "Superhuman abilities,Willingness to sacrifice";  
    }  
  
}
```



Static methods of interfaces can be invoked using the syntax **<interface-name>.<static-method>**

For instance, a static method named m1() within an interface I must be invoked using the syntax I.m1(). Using the unqualified name m1() to call the method is only possible within the interface's body or when the method is imported using a static import statement.



Since static methods are allowed from Java 8, we can write a **main method** inside an interface and execute it as well.

static methods in Interface



Overriding concept is not applicable for the static methods of Interface. Based on a implementation class requirement we can define exactly same method in the implementation class, it's valid but not be considered as overriding

```
public interface A {  
    public static void sayHello() {  
        System.out.println("Hi, This is a static method inside Interface");  
    }  
}  
  
public class B implements A {  
    public static void sayHello() {  
        System.out.println("Hi, This is a static method inside class");  
    }  
}
```

Multiple Inheritance using interfaces



A class has the ability to adopt multiple interfaces. To indicate this, you list all the interfaces that a class implements after the "implements" keyword in the class declaration. Each interface name is separated by a comma. When a class implements multiple interfaces, it commits to provide implementation logic for all the abstract methods found in each of those interfaces.

Consider below classes implementing couples of interfaces which will result into multiple inheritance,

IronMan

```
public class IronMan implements SuperHero, Person {  
  
    // class body  
  
}
```



SpiderMan

```
public class SpiderMan implements SuperHero, Person {  
  
    // class body  
  
}
```



CaptainAmerica

```
public class CaptainAmerica implements SuperHero, Person {  
  
    // class body  
  
}
```



Similarly an interface can inherit other multiple interfaces like shown below,

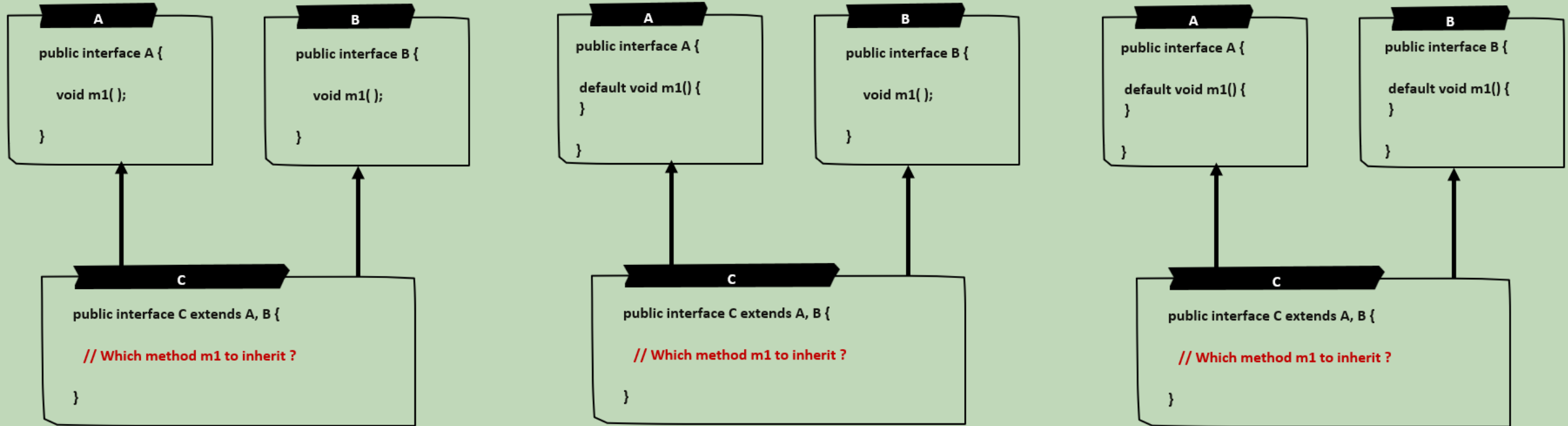
```
public interface SuperHero extends Person, Hero {  
    // interface body  
}
```



There is no restriction on the maximum number of interfaces that a class can implement or interface can extend. When a class implements an interface, its objects acquire an additional type. If a class implements multiple interfaces, its objects gain as many new types as the number of interfaces that have been implemented.

Scenario 1 during multiple inheritance using interfaces

Let me ask a tricky question around multiple inheritance,



When faced with a conflict arising from a combination of abstract-default or default-default methods in an interface, the compiler encounters ambiguity about which method to inherit. To resolve such conflicts, various approaches can be employed.

Scenario 1 during multiple inheritance using interfaces

To address the conflict arising from a combination of abstract-default or default-default methods in an interface, you have a few options:

1. Override the conflicting abstract methods with same abstract method or ignore it.
2. Override the conflicting method with a default method and provide a new implementation.
3. Override the conflicting method with a default method and call one of the methods of the superinterfaces.

```
public interface C extends A, B {  
  
    @Override  
    void m1();  
  
}
```

1. Override the conflicting abstract methods with same abstract method or ignore it

```
public interface C extends A, B {  
  
    @Override  
    default void m1() {  
        // custom logic  
    }  
  
}
```

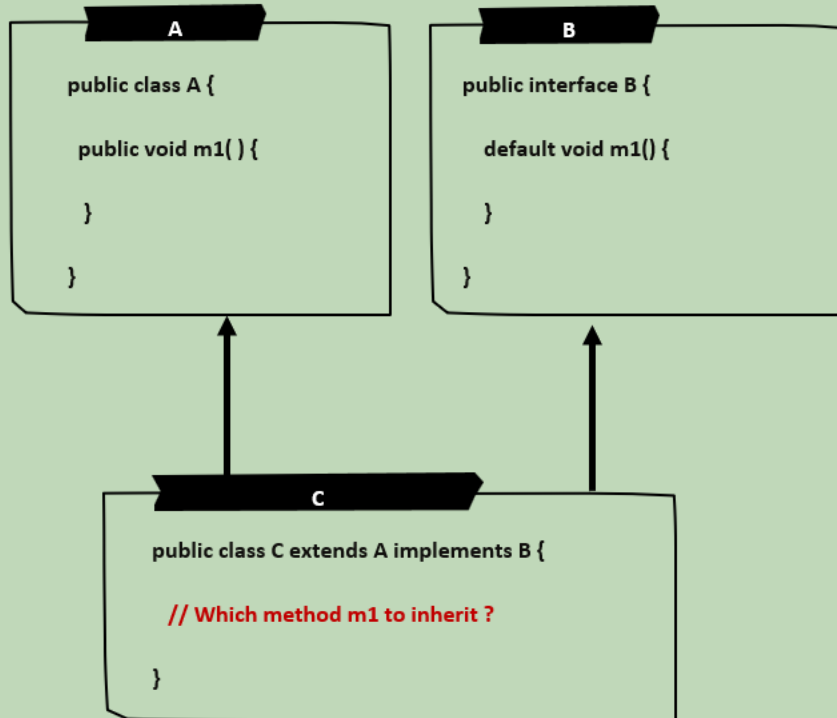
2. Override the conflicting method with a default method and provide a new implementation.

```
public interface C extends A, B {  
  
    @Override  
    default void m1() {  
        // A.super.m1();  
        // B.super.m1();  
    }  
  
}
```

3. Override the conflicting method with a default method and call one of the methods of the superinterfaces.

Scenario 2 - Inheriting conflicting implementations

Before Java 8, inheriting multiple implementations (non-abstract methods) from different supertypes was not possible. The introduction of default methods enabled a class to inherit conflicting implementations from its superclass and superinterfaces. When a class inherits a method with the same signature from multiple paths (superclass and superinterfaces), Java follows various rules to resolve the conflict.

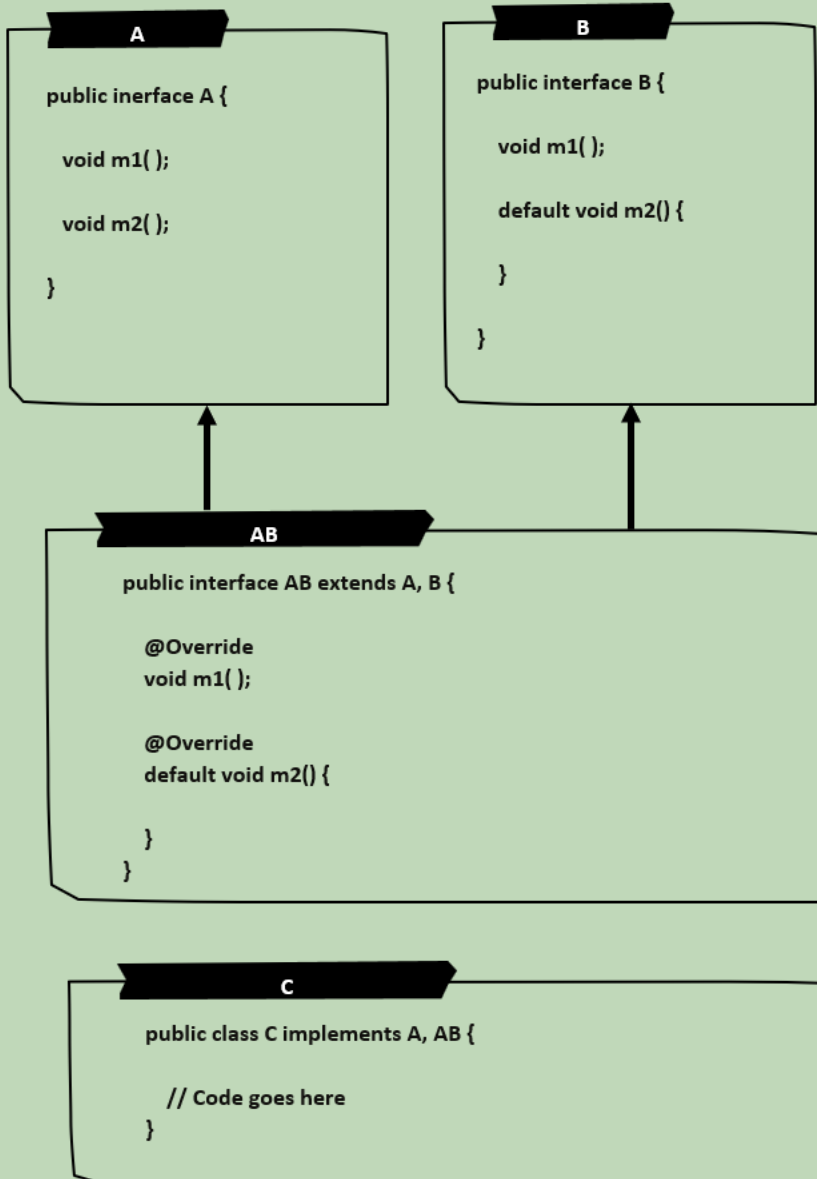


Solution

The Superclass Always Wins

If a class inherits or declares a method, the methods with the same signature in the superinterfaces will be ignored.

Scenario 3 - Inheriting conflicting implementations



Solution

Most Specific Superinterface wins

Which `m1()` method is inherited by the **C** class? Is it going to be `A.m1()` or `AB.m1()`? Since both of them are abstract methods, there will be no conflict. However, the `AB.m1()` method is most specific in this case, as it overrides the `A.m1()` method.

Which `m2()` method is inherited by the **C** class? Is it going to be `A.m2()` or `AB.m2()`? Since `A.m2()` has been overridden by the `AB.m2()`, the class **C** is going to inherit the `AB.m2()`.

If the scenario 2 and 3, if a class fails to resolve the conflicting method's inheritance, it must override the method and decide how it wants to handle the method. It may implement the method in a completely new way, or it may choose to call one of the methods in the superinterfaces.

Interface defines a **new type**




When a class implements an interface, it establishes a **supertype-subtype relationship**. The class becomes a **subtype** of all the interfaces it implements, and, conversely, all interfaces become **supertypes** of the class.

An interface serves as a blueprint for a new reference type. You can utilize an interface type in various contexts where a reference type is applicable. For instance, you can use an interface type to declare a variable (whether it's an instance, static, or local variable) or to specify a parameter type in a method. Additionally, you can employ it as a return type for a method and in other similar scenarios.

Consider below classes implementing couples of interfaces,


IronMan

```
public class IronMan implements SuperHero, Person {  
  
    // class body  
  
}
```




SpiderMan

```
public class SpiderMan implements SuperHero, Person {  
  
    // class body  
  
}
```



CaptainAmerica

```
public class CaptainAmerica implements SuperHero, Person {  
  
    // class body  
  
}
```



Person person = new Person(); // Fails. Can't create objects of interfaces

SuperHero hero = new SuperHero(); // Fails. Can't create objects of interfaces

IronMan ironMan = new IronMan(); // Success. Same for other classes as well

IronMan ironMan = new SuperHero(); // Fails. Can't create object of an interface. Same for Person

SuperHero sHero = new IronMan(); // Success. Object created for IronMan and it is referred using SuperHero obj reference

Person person = new SpiderMan(); // Success. Object created for SpiderMan and it is referred using Person obj reference

Interface defines a **new type**



Below kind of assignments is always allowed. This is very similar to upcasting in the case of SuperClass & SubClass. Why this is always allowed ?
Because IronMan is a SuperHero or Spider is a Person

```
SuperHero sHero = new IronMan(); // Success. Object created for IronMan and it is referred using SuperHero obj reference
```

```
Person person = new SpiderMan(); // Success. Object created for SpiderMan and it is referred using Person obj reference
```



Below kind of assignments is always not allowed. Why this is not allowed ? Because not every Super Hero is a IronMan or every Person is a SpiderMan

```
IronMan ironMan = new SuperHero(); // Fails. Can't create object of an interface
```

```
SpiderMan spiderMan = new Person(); // Fails. Can't create object of an interface
```



```
SpiderMan spiderMan = new SpiderMan();  
Person person = new IronMan();  
spiderMan = person; // Not Allowed. Person is not always a SpiderMan  
person = spiderMan; // Allowed. SpiderMan is always a Person
```

Interface defines a **new type**



Suppose you know for sure that a variable of the Person type contains reference to a IronMan or any other class object which implements Person interface. In such scenarios, we can assign the Person type variable to a variable of the IronMan type by using a typecast, as shown below,

```
Person person = new IronMan();  
IronMan ironman;  
ironMan = person; // Not allowed  
ironMan = (IronMan) person; // Allowed due to developer assurance with casting
```



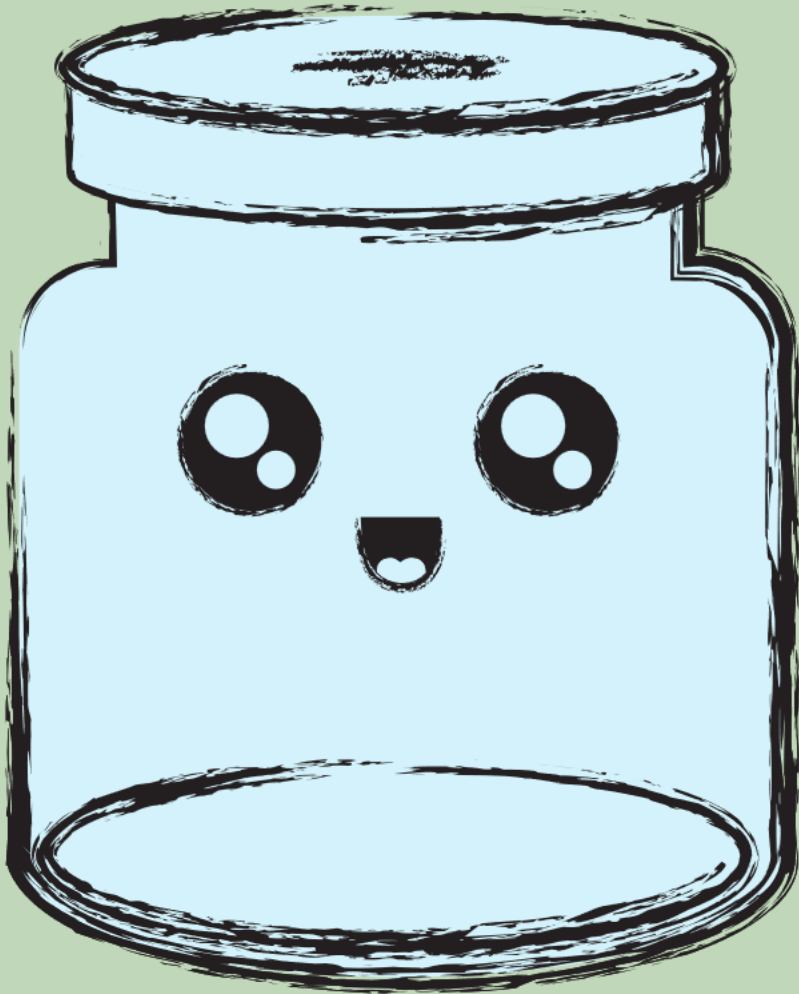
Failure to case at runtime will result in ClassCastException. To avoid this runtime surprises, you can use instanceof operator like shown below,

```
if (person instanceof IronMan) {  
    ironMan = (IronMan) person;  
}
```



We can use pattern-matching instanceof syntax as well which is introduced in Java 16:

```
if (person instanceof IronMan ironMan) {  
    // ironMan which is a IronMan type can be used here in the logic  
}
```



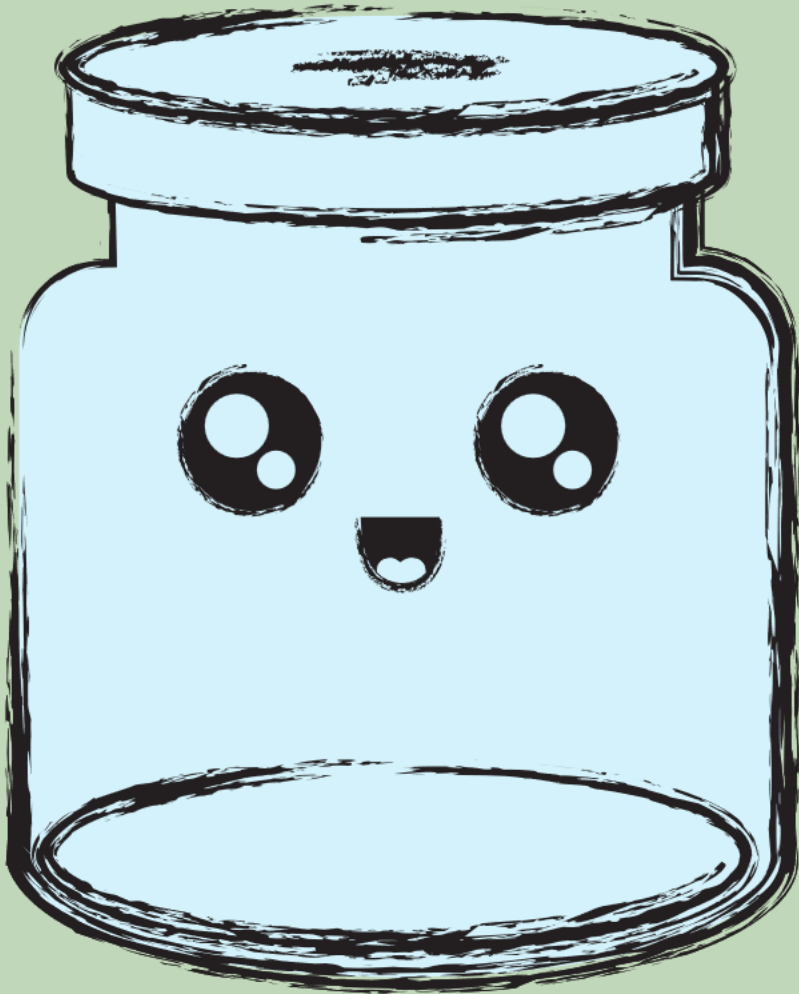
In Java, a **marker interface** is an interface **that doesn't contain any methods and empty**. Marker interfaces are also sometimes referred to as **"tagging interfaces"**.

Marker interfaces are typically used to provide metadata about a class, indicating some special characteristic or behavior of the class. In Java, there are marker interfaces, like **java.lang.Cloneable** and **java.io.Serializable**.

For example, if a class implements the Cloneable interface, it signals that the objects of that class can be cloned. To enable cloning, you must override the clone() method of the Object class in your class. Although you override the clone() method, your class objects cannot be cloned unless your class also implements the Cloneable marker interface.

Implementing Cloneable gives a specific meaning to the class, indicating that its objects can be cloned. When the clone() method is called, Java checks if the object's class implements Cloneable. If not, it throws an exception at runtime.

Similarly, the **Serializable** interface in Java is a marker interface that indicates that an object can be serialized (i.e., converted into a stream of bytes that can be saved to disk or transmitted over a network).

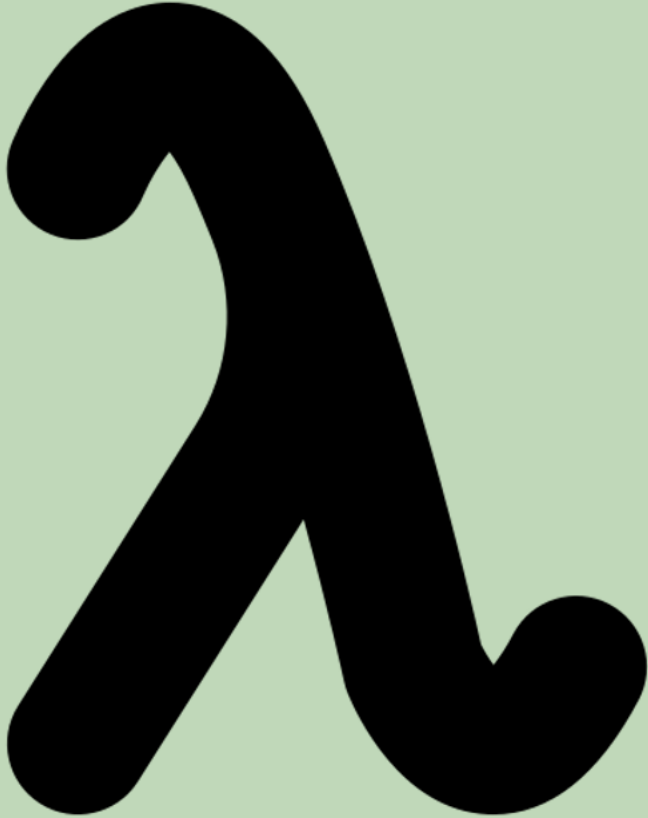


To use a marker interface, you simply define a new interface with no methods, and then have your class implement it. For example:

```
public interface MyMarkerInterface {}  
  
public class MyClass implements MyMarkerInterface {  
    // Class implementation here  
}
```

Note that marker interfaces are not used as frequently in Java as they once were, as alternative mechanisms like annotations have become more popular.

From Java 8



An interface with only one abstract method is commonly referred to as a functional interface. You can recall this concept using the acronym SAM (Single Abstract Method). It's important to note that static and default methods are not considered when determining if an interface qualifies as a functional interface.

With the introduction of Java version 8, the concept of functional interfaces was brought forth, allowing implementation through method references and lambda expressions.

It serves as a foundation for lambda expressions, which are a concise way to represent a single-method interface. The `@FunctionalInterface` annotation is often used to indicate that an interface is intended to be a functional interface. While not strictly required, it's a good practice to use the `@FunctionalInterface` annotation to clearly indicate the intent of the interface.

Java provides several built-in functional interfaces in the `java.util.function` package, such as `Consumer`, `Supplier`, `Predicate`, and `Function`. These interfaces cover common use cases for functional programming.

Example of a Functional Interface:

```
@FunctionalInterface
interface MyFunctionalInterface {
    void myMethod(); // Single abstract method
}
```


Class Vs Abstract Class Vs Interface

criteria	class	abstract class	interface
Inheritance	supports single inheritance	supports single inheritance	supports multiple inheritance
Implementation	Can be instantiated	Cannot be instantiated	Cannot be instantiated
Constructor	Can have constructors	Can have constructors	Constructors not allowed
Methods and Fields	Can have concrete, static methods & fields	Can have abstract ,concrete, static methods & fields	Can have constants, method signatures, default, static & private methods
Purpose	Used to create objects and define behavior	Used to provide a base for derived classes & define common behavior	Used to define behavior and contract
Access Modifiers	Can have public, protected, and private access modifiers	Can have public, protected, and private access modifiers	Methods & constants are by default public and no access modifiers are allowed
Relationship to Subclasses	Serves as a blueprint for derived classes	Serves as a base for derived classes and provides some implementation details	Defines a contract for implementing classes

Abstract classes and interfaces share some similarities, but they also have some important differences.



An abstract class can have a constructor, while an interface cannot. Abstract classes cannot be instantiated directly, just like interfaces. An abstract class can have a state, while an interface cannot.



Abstract classes can contain both abstract and concrete methods, as well as non-static and non-final fields. Additionally, abstract classes can define public, protected, and private methods, whereas all methods declared in an interface are automatically public. In contrast, interfaces only allow public, static, and final fields.



A class can extend only one abstract or concrete class, but it can implement multiple interfaces. This makes interfaces more flexible in terms of allowing a class to implement multiple behaviors from different sources.



To summarize, while both abstract classes and interfaces provide a way to define common behavior among classes, they differ in terms of their capabilities and intended usage. Abstract classes allow for more flexibility in defining fields and methods, while interfaces allow for more flexibility in defining multiple behaviors across classes.

If you are unsure whether to use abstract classes or interfaces, consider the following guidelines:



Abstract classes may be suitable if you want to share code among closely related classes, if the classes that extend your abstract class have common methods or fields, or if you need to declare non-static or non-final fields.

On the other hand, interfaces may be more appropriate if you expect unrelated classes to implement your interface, if you want to specify the behavior of a specific data type, but are not concerned with who implements the behavior, or if you want to take advantage of multiple inheritance of type.