# JOI - Java Objects by Interface

Heiko von Drachenfels, Oliver Haase, and Robert Walter

HTWG Konstanz - University of Applied Sciences, Constance, Germany
drachenfels|haase|rwalter@htwg-konstanz.de

## 1 Introduction

Object-orientation has long been equated with inheritance. Nowadays, however, it is widely aknowledged that inheritance can lead to undesirable dependencies between super and subclasses. Due to these dependencies, seemingly correct changes to a base class can lead to malfunctioning subclasses. This effect is known as the *fragile base class problem.* As a consequence, programmers are advised to avoid inheritance and replace it with object composition and delegation.

Another technique to reduce dependencies between classes is to consequently separate implementations (classes) from interfaces. Ideally, a class, *c*, that uses an object, *o*, should not know the actual implementation of *o*, but only the interface(s) that *o* provides. If *c* is consequently programmed against interfaces rather than implementations, replacing the actual implementations of the objects that *c* uses becomes trivial. Exchanging one class (implementation) without affecting other classes is a key requirement of truly modular software components, whose benefit becomes especially obvious during testing.

Both techniques, the avoidance of inheritance and the strict separation of interface and implementation can readily be applied in object-oriented languages such as Java, C++, and C#. The problem, however, is that these techniques are (to some extent) supported, but they are not enforced. In general, a good programming language should not only *allow for* a good programming style (which is possible with most languages, given the proper amount of self discipline), but it should *enforce* it.

In this paper, we present Joi (Java Objects by Interface), a Java extension that enforces the two programming techniques described above. Joi is a programming language that builds upon Java in that it uses Java syntax to a large extent and adds a few own keywords. A Joi program consists of (genuine Java) *interfaces* and so-called *components* which provide implementations for the interfaces. Components cannot be directly instantiated, but objects can only be gotten through factory methods and appear as instances of interface types.

Making objects only available through (Java) interfaces has several implications that lead to a cleaner object-oriented approach. For one, because interfaces cannot contain member variables, a component's member variables cannot be accessed from the outside and thus are automatically private. This not only enforces a better programming style, but also makes the need for explicit access right modifiers obsolete. Secondly, because an interface cannot contain static methods, all functionality has to be modeled as instance methods. Since in a purely object-oriented approach, static methods and static variables are modeled as members of singleton objects, Joi directly supports the singleton pattern, by taking care that the factory methods of a singleton component always return the only existing instance.

A Joi component cannot only use other joi components, but it can also instantiate and use plain vanilla Java objects. The other way around, a Java object can use a Joi component's factory method to get access to a Joi object and subsequently use it. That is, Joi and Java can be mixed together and inter-work, allowing for an evolutionary migration from existing Java code to newly developed Joi code.

## 2 Language Definition

On the level of method definitions, Joi uses Java syntax (and semantics). The differences between Joi and Java lie in the following aspects:

– A Joi component must provide (implement) at least one interface, because Joi objects are only seen through the interface types they provide. A Joi component not providing any interface would be unusable.
– There are no access right modifiers in Joi, because the access rights are defined through the interfaces a component provides: All instance variables and constructors are private because they cannot be included in an interface, all instance methods are public, and the special main method is implicitly made public.
– There are no static class members, they only exception being the main method.
– For each combination of provided interface and defined constructor, Joi generates a (static) factory method with the same parameters as the constructor that return an instance of the interface type.
  If, e.g., a component `MyComponent` provides the interface `MyInterface` and defines a constructor `MyComponent(String name, int[] numbers)`, then Joi automatically generates a factory method
  `MyInterface getMyInterface(String name, int[] numbers)`.
  These factory methods are the only means to get access to a Joi object.
– Joi has built-in support for the singleton pattern. If a component is defined as `singleton`, the factory methods always return the same singleton object. This support is particularly useful, because static variables and methods need to be modeled as singleton variables and methods.

The formal syntax definition of Joi in EBNF (Extendend Backus-Naur Form) is presented and explained in the following section.

### 2.1 EBNF for Joi

To keep the EBNF simple, we only show the parts that differ from Java and presume the Java elements to be well known. More specifically, this comprises the following non-terminals:

– `<PACKAGE_DIRECTIVE>`: The package the component belongs to.
– `<IMPORT>`: A Java style import statement.
– `<MAIN>`: The main method in Java syntax, except that the signature comes without modifiers (`public`, `static`) and without return value (`void`).
– `<INTERFACE_NAME>`: A type name in Java syntax.
– `<INSTANCE_METHOD>`, `<INSTANCE_VARIABLE>`, `<CONSTRUCTOR>`. Same as in Java, except without access right modifiers.

Based on the above presumptions, a simplified Joi-EBNF is shown in the following listing 1.1. As usual, the notation [ s ] stands for an optional occurrence of s, while { s } denotes arbitrary many (including zero) repetitions of s.

**Listing 1.1.** Joi - EBNF

```
CompileUnit =  [ <PACKAGE_DIRECTIVE> ] { <IMPORT> } Component ;


Component   =  [ "singleton" ] "component" <ID> "provides" Interfaces
               "{"
                  { Member }
                  [ <MAIN> ]
               "}" ;


Interfaces  =  <INTERFACE_NAME> [ "," Interfaces ] ;


Member      =  <INSTANCE_METHOD> | <INSTANCE_VARIABLE> | <CONSTRUCTOR> ;
```

A compile unit can start with a package declaration, followed by either none or an arbitrary amount of package imports. A component optionally starts with the keyword *singleton* and has to provide at least one interface. The members of a component are instance methods, instance variables and constructors, in arbitrary order. If the class defines a main method, it has to go last in the component definition.

## 2.2 Simple Example

In this section, we illustrate the use of Joi with a simple example. As each Joi component must provide at least one interface, we start with an interface definition as shown in listing 1.2. Please note that the interface is defined in plain vanilla Java.

**Listing 1.2.** Interface `WelcomeIF`

```java
package article_examples;

public interface WelcomeIF {
   void warmWelcome(String name);
}
```

As can be seen, the interface `WelcomeIF` contains only one method `warmWelcome` with a parameter `name` of type `String` and no return value.

Listing 1.3 shows a simple Joi component `JoiWelcome` which provides the java interface `WelcomeIF` by implementing the method `warmWelcome`.

**Listing 1.3.** Joi component `JoiWelcome`

```
package article_examples;

component JoiWelcome provides WelcomeIF {
   JoiWelcome() {}

   void warmWelcome(String name) {
      System.out.println("Pleased to meet you, " + name +
                         ". I hope you will enjoi it here!");
   }

   main(String[] args)  {
      if(args.length != 1) {
         System.out.println("Usage: java JoiWelcome <YourName>");
         System.exit(1);
      }

      WelcomeIF wif = JoiWelcome.getWelcomeIF();
      wif.warmWelcome(args[0]);
   }
}
```

The component `JoiWelcome` contains a main method that shows the use of the generated factory method `getWelcomeIF` to get an instance of type `WelcomeIF`. At this instance `wif`, only the methods defined in `WelcomeIF` can be called, in our case the method `warmWelcome`.

## 3 Implementation

A Joi component `MyComponent` is defined in a file named `MyComponent.joi`. In our current implementation, the Joi compiler `joic` converts the Joi source file into a Java source file containing a Java class with the same name, `MyComponent`, which hides the implementation details in a nested class. After running `joic`, the java source file can be compiled into byte code and executed by a Java virtual machine, as usual. Listing 1.4 shows the result of applying `joic` to the Joi component `JoiWelcome` in listing 1.3.

**Listing 1.4.** Generated Java class `JoiWelcome`

```
package article_examples;

public final class JoiWelcome {
   private JoiWelcome() { }

   public static WelcomeIF getWelcomeIF () {
      return (WelcomeIF) new Implementation();
   }

   private static final class Implementation implements WelcomeIF {
      private Implementation () { }
      public void warmWelcome (String name) {
         System.out.println("Pleased to meet you " + name +
                            ". I hope you will enjoi it here!");
      }
   }

   public static void main (String[] args) {
      if(args.length != 1) {
         System.out.println("Usage: java JoiWelcome <YourName>");
```

```
            System.exit(1);
        }

        WelcomeIF wif = JoiWelcome.getWelcomeIF();
        wif.warmWelcome(args[0]);
    }
}
```

As can be seen, the interface `WelcomeIF` is implemented not by the outer class `JoiWelcome`, but by the nested class `Implementation`. The generated factory method `getWelcomeIF` instantiates an object of the nested class `Implementation` and returns it as an instance of the interface type `WelcomeIF`. Please note that the only public method of the Java class `JoiWelcome` is the factory method. Thus, from a user perspective `JoiWelcome` is an utility class whose only use is to generate instances of type `WelcomeIF`.

The use of the factory pattern minimizes the dependency between implementation provider and user. The user must code against an interface rather than against an implementation. The actual implementation can be exchanged by simply replacing one factory with another one that generates objects of the same interface type.

Finally, it should be noted that Joi components could also directly be transformed into byte code, without the intermediate step of Java source code. We opted for transformation on the source code level, because it was easier to realize and because the generated Java source code nicely illustrates the ideas and mechanisms behind Joi.