



MICHAEL KÖLLING

# The Evolving Nature of Interfaces

Understanding multiple inheritance in Java

In the “New to Java” series, I try to provide benefit by picking topics that invite a deeper understanding of the conceptual background of a language construct. Often, novice programmers have a working knowledge of a concept—that is, they can use it in many situations, but they lack a deeper understanding of the underlying principles that would lead to writing better code, creating better structures, and making better decisions about when to use a given construct. Java *interfaces* are just such a topic.

In this article, I assume that you have a basic understanding of inheritance. Java interfaces are closely related to inheritance, as are the `extends` and `implements` keywords. So, I will discuss why Java has two different inheritance mechanisms (indicated by these keywords), how abstract classes fit in, and what various tasks interfaces can be used for.

As is so often the case, the story of these features starts with some quite simple and elegant ideas that lead to the definition of concepts in early Java versions, and the story gets more complicated as Java advances to tackle more-intricate, real-world problems. This leads to the introduction of default methods in Java 8, which muddy the waters a bit.

## A Little Background on Inheritance

Inheritance is quite straightforward to understand in principle: a class can be specified as an extension of another class. In such a case, the present class is called a *subclass*, and the class it’s extending is called the *superclass*. Objects of the subclass have all the properties of both the superclass and the subclass. They have all fields defined in either subclass or

superclass and also all methods from both. So far, so good.

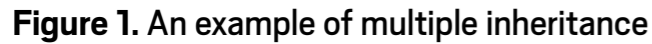
Inheritance is, however, the equivalent of the Swiss Army knife in programming: it can be used to achieve some very diverse goals. I can use inheritance to reuse some code I have written before, I can use it for subtyping and dynamic dispatch, I can use it to separate specification from implementation, I can use it to specify a contract between different parts of a system, and I can use it for a variety of other tasks. These are all important, but very different, ideas. It is necessary to understand these differences to get a good feel for inheritance and interfaces.

## Type Inheritance Versus Code Inheritance

Two main capabilities that inheritance provides are the ability to inherit code and the ability to inherit a type. It is useful to separate these two ideas conceptually, especially because standard Java inheritance mixes them together. In Java, every class I define also defines a type: as soon as I have a class, I can create variables of that type, for example.

When I create a subclass (using the `extends` keyword), the subclass inherits both the code and the type of the superclass. Inherited methods are available to be called (I’ll refer to this as “the code”), and objects of the subclass can be used in places where objects of the superclass are expected (thus, the subclass creates a subtype).

Let’s look at an example. If `Student` is a subclass of `Person`, then objects of class `Student` have the type `Student`, but they also have the type `Person`. A student is a person. Both the code and the type are inherited.



## Multiple Inheritance

ber, and so on). But they are also students: they are enrolled in a course, have a student ID number, and so on. I can model this as multiple inheritance (see **Figure 1**).

but different implementations? For these cases, I need language constructs that specify some solution to the problem of ambiguity and name overloading. However, it gets worse.

A more complicated scenario is known as *diamond inheritance* (see Figure 2). This is where a class (`PhDStudent`) has two superclasses (`Faculty` and `Student`), which in turn have a common superclass (`Person`). The inheritance graph forms a diamond shape.

The answer is: it depends. If the field in question is, say, an ID number, maybe a PhD student should have two: a student ID and a faculty/payroll ID that might be a different number. If the field is, however, the person's family name, then you want only one (the PhD student has only one family name, even though it is inherited from both superclasses).

In short, things can become very messy. Languages that allow full, multiple inheritance need to have rules and constructs to deal with all these situations, and these rules are complicated.

When you think about these problems carefully, you realize that all the problems with multiple inheritance are related to inheriting code: method implementations and fields. Multiple code inheritance is messy, but multiple type inheritance causes no problems. This fact is coupled with another observation: multiple code inheritance is not terribly important, because you can use *delegation* (using a reference to another object) instead, but multiple subtyping is often very useful and not easily replaced in an elegant way.







