Understanding multiple inheritance in Java

by Michael Kölling

In the occasional "New to Java" series, I try to pick 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 often 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 challenge led to the introduction of default methods in Java 8, which muddied the waters a bit.

A Little Background on Inheritance

Inheritance is 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.

The decision to link type inheritance and code inheritance in Java is a
language design choice: it was done because it is often useful, but it is
not the only way a language can be designed. Other programming
languages allow inheriting the code without inheriting the type (such as
C++ private inheritance) or inheriting the type without inheriting the code
(which Java also supports, as I explain shortly).

Multiple Inheritance

The next idea entering the mix is multiple inheritance: a class may have
more than one superclass. Let me give you an example: PhD students at
my university also work as instructors. In that sense, they are like faculty
(they are instructors for a class, have a room number, a payroll number,
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).

Figure 1. An example of multiple inheritance

PhDStudent is a subclass of both Faculty and Student. This way, a
PhD student will have the attributes of both students and faculty.
Conceptually this is straightforward. In practice, however, the language
becomes more complicated if it allows multiple inheritance, because that
introduces new problems: What if both superclasses have fields with the
same name? What if they have methods with the same signature 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.

Diamond Inheritance

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.

Figure 2. An example of diamond inheritance

Now, consider this question: If there is a field in the top-level superclass (Person, in this case), should the class at the bottom (PhDStudent)
have one copy of this field or two? It inherits this field twice, after all, once
via each of its inheritance branches.
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).

The Java designers arrived at a pragmatic solution: allow only single inheritance for code, but allow multiple inheritance for types.

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.

**Type Inheritance to the Rescue**

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.

That is why the Java designers arrived at a pragmatic solution: allow only single inheritance for code, but allow multiple inheritance for types.

**Interfaces**

To make it possible to have different rules for types and code, Java needs to be able to specify types without specifying code. That is what a Java interface does.

Interfaces specify a Java type (the type name and the signatures of its methods) without specifying any implementation. No fields and no method bodies are specified. Interfaces can contain constants. You can leave out the modifiers (public static final for constants and public for methods)—they are implicitly assumed.

This arrangement provides me with two types of inheritance in Java: I can inherit a class (using extends), in which I inherit both the type and the code, or I can inherit a type only (using implements) by inheriting from an interface. And I can now have different rules concerning multiple inheritance: Java permits multiple inheritance for types (interfaces) but only single inheritance for classes (which contain code).

**Benefits of Multiple Inheritance for Types**

The benefits of allowing the inheritance of multiple types—essentially of being able to declare that one object can be viewed as having a different type at different times—are quite easy to see. Suppose you are writing a traffic simulation, and in it you have objects of class Car. Apart from cars, there are other kinds of active objects in your simulation, such as pedestrians, trucks, traffic lights, and so on. You may then have a central collection in your program—say, a List—that holds all the actors:

```java
private List<actor> actors;
</actor>
```

Actor, in this case, could be an interface with an act method:

```java
public interface Actor
{
    void act();
}
```
Your *Car* class can then implement this interface:

```java
class Car implements Actor {
    public void act() {
        ...
    }
}
```

Note that, because *Car* inherits only the type, including the signature of the *act* method, but no code, it must itself supply the code to implement the type (the implementation of the *act* method) before you can create objects from it.

So far, this is just single inheritance and could have been achieved by inheriting a class. But imagine now that there is also a list of all objects to be drawn on screen (which is not the same as the list of actors, because some actors are not drawn, and some drawn objects are not actors):

```java
private List<drawable> drawables;
</drawable>
```

You might also want to save a simulation to permanent storage at some point, and the objects to be saved might, again, be a different list. To be saved, they need to be of type *Serializable*:

```java
private List<serializable> objectsToSave;
</serializable>
```

In this case, if the *Car* objects are part of all three lists (they act, they are drawn, and they should be saved), the class *Car* can be defined to implement all three interfaces:

```java
class Car implements Actor, Drawable, Serializable
```

Situations like this are common, and allowing multiple supertypes enables you to view a single object (the car, in this case) from different perspectives, focusing on different aspects to group them with other similar objects or to treat them according to a certain subset of their possible behaviors.

Java’s GUI event-processing model is built around the same idea: event handling is achieved via event listeners—interfaces (such as *ActionListener*) that often just implement a single method—so that objects that implement it can be viewed as being of a listener type when necessary.

**Abstract Classes**

I should say a few words about *abstract classes*, because it is common to wonder how they relate to interfaces. Abstract classes sit halfway between classes and interfaces: they define a type and can contain code (as classes do), but they can also have *abstract methods*—methods that are specified only, but not implemented. You can think of them as partially implemented classes with some gaps in them (code that is missing and needs to be filled in by subclasses).

In my example above, the *Actor* interface could be an abstract class instead. The *act* method itself might be abstract (because it is different in each specific actor and there is no reasonable default), but maybe it contains some other code that is common to all actors.
In this case, I can write `Actor` as an abstract class, and the inheritance declaration of my `Car` class would look like this:

```java
class Car extends Actor implements Drawable, Serializable {
    // code
}
```

If I want several of my interfaces to contain code, turning them all into abstract classes does not work. As I stated before, Java allows only single inheritance for classes (that means only one class can be listed after the `extends` keyword). Multiple inheritance is for interfaces only.

There is a way out, though: default methods, which were introduced in Java 8. I’ll get to them shortly.

**Empty Interfaces**

Sometimes you come across interfaces that are empty—they define only the interface name and no methods. `Serializable`, mentioned previously, is such an interface. `Cloneable` is another. These interfaces are known as marker interfaces. They mark certain classes as possessing a specific property, and their purpose is more closely related to providing metadata than to implementing a type or defining a contract between parts of a program. Java, since version 5, has had annotations, which are a better way of providing metadata. There is little reason today to use marker interfaces in Java. If you are tempted, look instead at using annotations.

**A New Dawn with Java 8**

So far, I have purposely ignored some new features that were introduced with Java 8. This is because Java 8 adds functionality that contradicts some of the earlier design decisions of the language (such as “only single inheritance for code”), which makes explaining the relationship of some constructs quite difficult. Arguing the difference between and justification for the existence of interfaces and abstract classes, for instance, becomes quite tricky. As I will show in a moment, interfaces in Java 8 have been extended so that they become more similar to abstract classes, but with some subtle differences.

In my explanation of the issues, I have taken you down the historical path—explaining the pre-Java 8 situation first and now adding the newer Java 8 features. I did this on purpose, because understanding the justification for the combination of features as they are today is possible only in light of this history.

If the Java team were to design Java from scratch now, and if breaking backward compatibility were not a problem, they would not design it in the same way. The Java language is, however, not foremost a theoretical exercise, but a system for practical use. And in the real world, you must find ways to evolve and extend your language without breaking everything that has been done before. Default methods and static methods in interfaces are two mechanisms that made progress possible in Java 8.

**Evolving Interfaces**

One problem in developing Java 8 was how to evolve interfaces. Java 8 added lambdas and several other features to the Java language that made it desirable to adapt some of the existing interfaces in the Java library. But how do you evolve an interface without breaking all the existing code that uses this interface?

Imagine you have an interface `MagicWand` in your existing library:

```java
public interface MagicWand {
    void doMagic();
}
```
This interface has already been used and implemented by many classes in many projects. But you now come up with some really great new functionality, and you would like to add a really useful new method:

```java
public interface Magic Wand
{
    void doMagic();
    void doAdvancedMagic();
}
```

If you do that, then all classes that previously implemented this interface break, because they are required to provide an implementation for this new method. So, at first glance, it seems you are stuck: either you break existing user code (which you don’t want to do) or you’re doomed to stick with your old libraries without a chance to improve them easily. (In reality, there are some other approaches that you could try, such as extending interfaces in subinterfaces, but these have their own problems, which I do not discuss here.) Java 8 came up with a clever trick to get the best of both worlds: the ability to add to existing interfaces without breaking existing code. This is done using default methods and static methods, which I discuss now.

**Default Methods**

Default methods are methods in interfaces that have a method body—the default implementation. They are defined by using the `default` modifier at the beginning of the method signature, and they have a full method body:

```java
public interface Magic Wand
{
    void doMagic();
    default void doAdvancedMagic()
    {
        ... // some code here
    }
}
```

Classes that implement this interface now have the chance to provide their own implementation for this method (by overriding it), or they can completely ignore this method, in which case they receive the default implementation from the interface. Old code continues to work, while new code can use this new functionality.

**Static Methods**

Interfaces can now also contain static methods with implementations. These are defined by using the usual `static` modifier at the beginning of the method signature. As always, when writing interfaces, the `public` modifier may be left out, because all methods and all constants in interfaces are always public.

**So, What About the Diamond Problem?**

As you can see, abstract classes and interfaces have become quite similar now. Both can contain abstract methods and methods with implementations, although the syntax is different. There still are some differences (for instance, abstract classes can have instance fields, whereas interfaces cannot), but these differences support a central point: since the release of Java 8, you have multiple inheritance (via interfaces) that can contain code!

At the beginning of this article I pointed out how the Java designers treaded very carefully to avoid multiple code inheritance because of
possible problems, mostly related to inheriting multiple times and to name clashes. So what is the situation now?

As usual, the Java designers devised the following sensible and practical rules to deal with these problems:

- Inheriting multiple abstract methods with the same name is not a problem—they are viewed as the same method.
- Diamond inheritance of fields—one of the difficult problems—is avoided, because interfaces are not allowed to contain fields that are not constants.
- Inheriting static methods and constants (which are also static by definition) is not a problem, because they are prefixed by the interface name when they are used, so their names do not clash.
- Inheriting from different interfaces multiple default methods with the same signature and different implementations is a problem. But here Java chooses a much more pragmatic solution than some other languages: instead of defining a new language construct to deal with this, the compiler just reports an error. In other words, it’s your problem. Java just tells you, “Don’t do this.”

**Conclusion**

Interfaces are a powerful feature in Java. They are useful in many situations, including for defining contracts between different parts of the program, defining types for dynamic dispatch, separating the definition of a type from its implementation, and allowing for multiple inheritance in Java. They are very often useful in your code; you should make sure you understand their behavior well.

The new interface features in Java 8, such as default methods, are most useful when you write libraries; they are less likely to be used in application code. However, the Java libraries now make extensive use of them, so make sure you know what they do. Careful use of interfaces can significantly improve the quality of your code.

This article was originally published in the January/February 2018 issue of Java Magazine.

[An earlier version ran in the September/October 2016 issue of Java Magazine. — Ed.]

**Michael Kölling**

Michael Kölling is a Java Champion and a professor at the University of Kent, England. He has published two Java textbooks and numerous papers on object orientation and computing education topics, and he is the lead developer of BlueJ and Greenfoot, two educational programming environments. Kölling is also a Distinguished Educator of the ACM.