CODING

Understanding Java generics, Part 2: The hard part

Learn about wildcards, bounded wildcards, subtyping, and type erasure.

by Michael Kölling

May 14, 2021

[*Java Magazine* is pleased to republish this two-part tutorial from Michael Kölling, published in 2016, about generics. You can read the first part here. —*Ed*.]

Welcome back to the discussion of generic types in Java. The previous article introduced generic types, why they are useful, what you can do with them, and how to use them. The introductory part of the topic is quite straightforward, but at the end of that discussion I mentioned a problem: generic collections and subtyping.

To recap: I want to write a general `printList` method such as this.

```
private void printList(List<Person> list)
```

And I want it to print out lists of subtypes of `Person`, such as `List<Student>` or `List<Faculty>`. In other words, given that `Student` is a subtype of `Person`, I want to call the method above like this.

```
List<Student> students = getStudentList();
printList(students);
```

This does not work in Java, because `List<Student>` is *not* considered a subtype of `List<Person>` even though `Student` is a subtype of `Person`.

So, why is `List<Student>` not a subtype of `List<Person>`? If you think only about printing out the list, there seems to be no problem. The `printList` method could call, for instance, a `print` method on all the list's elements (which might be defined in `Person` and redefined appropriately in the subtypes). All seems well. Right?

The underlying problem becomes apparent when you consider that the `printList` method could include functionality to *modify* the list. It could, for example, include the following line:

```
list.add(new Faculty());
```

Because the static type of the `list` variable (the formal parameter to the method) is `List<Person>`, and because `Faculty` is a subtype of `Person`, adding the object causes no type problems. However, if the actual list passed to the `printList` method were a list of students, then I have now added a `Faculty` object to the `Student` list. This is clearly an error and should not be allowed to happen.

A solution is to declare that `List<Student>` is *not* a subtype of `List<Person>` and to prevent student lists from being passed in to the `printList` method. Type safety is preserved, but I am back to square one: How can I now write the general `printList` method?

Wildcards to the rescue

The solution to this problem is to use wildcards. I can write the `printList` method like this.

```
private void printList(List<?> list)
```

Note the question mark in place of the element type of the list. The question mark is the wildcard symbol, and it denotes a type called *unknown*. The parameter is now a *list of unknown type*.

There is an obvious benefit to this construct. I can now do what I intended to do: I can call the `printList` method with either `List<Student>` or `List<Faculty>` as parameters, as follows:

```
List<Student> students = getStudentList();
List<Faculty> professors = getFacultyList();
printList(students);
printList(professors);
```

Every list type is a subtype of the list of this unknown type, so this code now works. The trade-off is that I cannot add to the list

when the element type is unknown, so this avoids the type problem discussed earlier when I tried to add to the list.

What is known about the unknown type?

The wildcard is a good step forward, but it does not solve all the problems. You can see this if you think about what I can do with the list elements now. What if the `Person` superclass had a method `printAddressDetails` that I want to use as part of the `printList` method?

```
private void printList(List<?> list) {
      for (Person p: list) {
            ...
            p.printAddressDetails();
      }
}
```

This code will not work. The advantage of using the unknown type is that you can pass in lists of any type, but you pay because you don't know much about that type. All you know, in fact, is that the type is a subtype of `Object` (because every type is a subtype of `Object`). Therefore, I cannot treat the element type as `Person`.

Not knowing much about the element type can still be acceptable in some cases. I could still use all list operations that do not depend on the element type, such as `size()` and `clear()`. I could also do anything that I can do with the `Object` type, such as using the `toString` method (perhaps implicitly by calling `System.out.println`).

But to call type-specific methods, I need something else. In using the wildcard, I went from saying that the parameter is exactly a *List of Person* to saying that it is a *List of anything*. Instead, I would like to say that it is a *List of any subtype of Person*. I can do this with a *bounded wildcard*.

Generic parameters can have bounds which restrict the kind of actual types that can be used for them. Consider this next version of the `printList` method.

```
private void printList(List<? extends Person>
```

This definition now allows lists of `Person` or subtypes of `Person` (and only these) as parameters. Because I am using a wildcard, I am still not allowed to add to the list, but I now know that all elements are of type `Person` (or its subtypes). I can now treat elements as `Person` objects and call the appropriate methods. This finally solves the problem.

Wildcards are not the only place where bounds can be used and are useful. Type bounds can also be employed in the declaration of generic types and in methods without wildcards.

For example, I can define a generic type `PersonList` that accepts only `Person` and its subtypes as parameters.

```
class PersonList<T extends Person>
```

This is similar to the definition of `ArrayList` that I showed in the previous article, but this time only subtypes of `Person` can be used to instantiate the type.

```
PersonList<Student> students =
        new PersonList<Students>();
PersonList<Faculty> professors =
        new PersonList<Faculty>();
```

In return, all methods from the `Person` type can now be used on objects of type `T` in the implementation of the `PersonList` class, because I have a guarantee that any concrete instantiation of `T` will have these methods.

Generic methods

In the previous examples, the generic type parameter was introduced in the class header when declaring a generic class. It is also possible to have single *generic methods*, without making the whole class generic. In that case, the single method can handle generic types. Generic methods are often combined with bounded generic types. Consider the following example. Here, I attempt to write a method that prints all elements from a list that are smaller than a given limit.

```
public <T> void underLimit(List<T> myList, T
        for (T e : myList) {
              if (e < limit)
                  System.out.println(e);
        }
}
```

The new syntax here is the type parameter `<T>` in the header after the keyword `public` and before the return type. I am assuming that this method is in a class that is not generic, so no type parameter has previously been declared. To use a generic type in the parameter list, I need to declare this type first—that is the effect of writing the type `<T>` in the header.

This code will fail, however, because the less-than operator cannot be applied to any unspecified type `T`. Instead, I could use the `compareTo` method, but this works only when `T` is a subtype

of `Comparable`. I can enforce that restriction by changing the method as follows:

```
public <T extends Comparable<T>> void underLi
        List<T> myList, T limit) {
        for (T e : myList) {
              if (e.compareTo(limit) < 0)
                  System.out.println(e);
         }
}
```

Here, I have declared that I accept only types for type `T` that are subtypes of `Comparable` so that the methods needed are guaranteed to be available.

Upper bounds and lower bounds

So far, I have discussed bounded types only by showing an upper bound to establish a supertype (an upper bound) for the wildcard parameter, as in the following:

```
List<? extends Person>
```

The effect is that only the named type or its subtypes can be used to instantiate the type. In other words, the concrete type at the point of use must extend (or implement) `Person`. If you drew a typical inheritance hierarchy around `Person`, only `Person` or the classes below it in the hierarchy can be used.

I can also restrict the type in the other direction by declaring a lower bound.

```
List<? super Person>
```

By using the `super` keyword for the declaration, I am stating that the type has to be `Person` or a supertype of `Person`. If you picture this in an inheritance hierarchy, you can see `Person` or the types above it in the hierarchy. This construct is used less often than upper bounds, but it can be helpful in some situations.

Type erasure

In addition to knowing how to use generic types, it is also useful to know a little bit about how they are implemented in the Java compiler and the JVM. If you ever talked with anyone about the implementation, it is likely that the term *type erasure* came up at some stage. It is important to know what this means, because it affects not only the efficiency of implementation but also the semantics of your code in certain cases.

At the core of type erasure is the fact that type parameters exist only at compile time; they are completely removed at runtime. They are a construct exclusively used for type checking during compilation to ensure type safety, but they are not carried through into the Java bytecode.

When learning about generics, it's often helpful to think of generic classes as expanded at instantiation time. For example, consider the following type:

```
class List<T> {
      public void add(T elem);
      ...
}
```

If it is then instantiated by using the concrete type `List<String>`, it can be thought of as having every occurrence of `T` in the source text replaced by `String`, so that the parameter type in the `add` method becomes `String`. For `List<Integer>`, each `T` would be replaced by `Integer`, and so on.

This is a useful mental model to start understanding generics… but it is inaccurate. You need to know how generics *really* work, because sometimes how they work makes a noticeable difference.

Java's generic types are *never* expanded into their concrete instantiations: not in source code, not in binary code, not on disk, and not in memory.

This is different from what happens with templates in C++, for example, where this expansion actually occurs.

In Java, the generated code will merely insert `Object` as the type for each unbounded type parameter, or the bounding type for types that have bounds. Thus, `List<String>`, `List<Integer>`, and `List<Person>` are all represented by a single class, `List<Object>`, by the time your program executes. By then, the compiler has made sure that you used the class in a type-safe manner, and type problems have been prevented. You used many types but get only one class.

Discarding type parameter information at runtime has advantages and disadvantages. One of the advantages is that it saves time and space: The class file needs to exist only once for every generic class. It does not need to be stored or compiled multiple times. This is a clear benefit.

On the downside, type erasure makes life harder for tool writers, such as creators of development environments. It is hard, for example, for a debugger to figure out the correct type for an object at runtime if that type is derived from a generic class. No information is kept in the class file about the full type information.

More important for you as a programmer is the fact that type erasure can influence the behavior of your code. The following sections describe examples where it is necessary to understand type erasure to understand the behavior of the Java system.

No instanceof for types with type parameters

The `instanceof` operator cannot be used with parameterized types. Consider the following attempt to use `List<T>`, as defined in the previous section:

```
if (list instanceof List<Person>) {
     List<Person> pl = (List<Person>) list;
}
```

This code looks entirely reasonable, but if you consult the previous section on type erasure, you will see that the runtime system has no idea whether a type is `List<Person>` because it does not keep this information around. (All the runtime knows about is `List<Object>` but nothing more specific.) Therefore, the runtime cannot perform a type-safe check and give you an answer. You will see an error saying `illegal generic type for instanceof`.

A similar problem shows up if you use the `getClass` method.

```
List<Student> sl = new ArrayList<Student>();
List<Faculty> fl = new ArrayList<Faculty>();
if (sl.getClass() == fl.getClass())
     ...
```

At first glance, you might think that the condition in the `if` statement is false, but because of type erasure, the statement will evaluate to true. As far as the runtime system is concerned, the class of both objects is `ArrayList`.

One of the areas where type erasure becomes most visible in source code is when you use static attributes in generic classes. Static methods and static fields are shared between all instantiations of a generic class. The reason is again the same: Only one copy of the generic class actually exists. You have to be aware of this to write correct code. A side effect of this is that it is not possible to declare a static field of a generic parameter type.

```
class MyClass<T> {
      private static T value; // error
      ...
}
```

Because this field is shared between all variants of the type, it cannot refer to the type parameter of specific instantiations.

Java trivia: Arrays and type safety

The implementation of arrays in Java has a hole in its type system. This is one of the rare cases where Java is not statically type-safe.

Consider this question: If `B` is a subtype of `A`, is `List<B>` a subtype of `List<A>`? For lists, the answer is no. Earlier in this article, I explained why this is and how it could go wrong if you were to consider `List<B>` a subtype. However, for arrays (a very similar situation), Java *does* consider the list to be a subtype. This introduces a potential type problem. Consider the following code:

```
A[] aa;
B[] ba = new B[3];
aa = ba; // allowed! B[] is subtype of A[]
aa[0] = new B();
aa[1] = new A(); // java.lang.ArrayStoreExcep
```

The last line in this example represents a type error: I am trying to insert an `A` object into an array of `B`. However, the assignment in the third line is allowed. This problem is picked up only at runtime, not at compile time, breaking Java's static type safety. When designing generic classes, the Java team decided to be more conservative and detect the equivalent problem at compile time.

Conclusion

Generic types are easy to understand in principle and generally quite easy to use. However, when you start writing more sophisticated code—particularly if you're writing libraries—you might run into a whole range of situations where you need to understand the advanced constructs in generics.

When you put the concepts together, the class and method definitions can become tricky to read, even for experienced programmers. Look at the `max` method of class `Collections` in the standard library, for example, or the definition of methods in the `Class` class. You will see that it can take some time to get your head around the combination of all the constructs. Don't be discouraged! These complex constructs are rare, and with the concepts I have discussed here and some practice, you should be able to work out most of it. More importantly, you should be able to write correct and flexible code yourself.

Dig deeper

- Java tutorial on generics

- Generics: How they work and why they are important
- Java SE documentation for generics

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.

Share this Page