MICHAEL KÖLLING

Generics: The Hard Parts

Wildcards, subtyping, and type erasure in generics

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

In short, I wanted to write a general `printList` method such as this:

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

And I wanted 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 wanted to call the method above like this:

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

This does not work in Java. The reason is that `List<Student>` is not considered a subtype of `List<Person>` even though `Student` is a subtype of `Person`.

What's the Problem?

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.

The problem becomes apparent when you consider that the `printList` method could also 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 `Faculty` is a subtype of `Person`, adding this 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 a clear error and should not be allowed to happen.

The only 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 my general `printList` method?

Wildcards to the Rescue

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

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

Note the question mark in place of the element type of

PHOTOGRAPH BY JOHN BLYTHE

the list. The question mark is the wildcard symbol, and it denotes a type called *unknown*. My 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 my `printList` method with both `List<Student>` and `List<Faculty>` as parameters:

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

Every list type is considered to be 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 I avoid 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 my problems. You can see this if you think about what I can do with my list elements now. What if my `Person` superclass had a method `printAddressDetails` that I want to use as part of my `printList` method:

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

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

Not knowing much about the element type can still be OK 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 (maybe 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 my 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*.

Bounded Wildcards

Generic parameters can have bounds, which restrict what kind of actual types can be used for them. Consider this next version of my `printList` method:

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

This definition now allows lists of `Person` or subtypes of `Person` (and only these) as parameters, just as I intended. Because I am using a wildcard, I am still not allowed to add to the list, but I 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 my problem.

Other Bounded Types

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 last issue of *Java Magazine*, 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 my implementation of the PersonList class, because I have a guarantee that any concrete instantiation of T will have these methods.

Generic Methods

This is a good time to introduce another generic feature: generic methods. In the previous examples, the generic type parameter was introduced in the class header when we declared 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 limit) {
    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 can use the compareTo method, but this works only when T is a subtype of Comparable. I can enforce this by changing my method as follows:

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

Here, I have declared that I only accept 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, for example:

```
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 we were to draw 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 my 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 use `Person` or the types above it in the hierarchy. This is less often used than upper bounds but can be helpful in some situations.

Implementation

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.

Type Erasure

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.

To understand generics at first, it is 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 ultimately false. It is useful, because it is easy to understand, and it gives a good approximation of how generics behave. It is important to know, however, how things really work, because sometimes that makes a noticeable difference.

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 than templates in C++, for example, where this expansion actually happens. In Java, the generated code will just 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 why it does not work: the runtime system has no idea whether a type is `List<Person>`, because it does not keep this information around. (All it knows about is `List<Object>` but nothing more specific.) So it cannot perform this check and give you the answer. You will see an error saying `illegal generic type for instanceof`.

The same problem shows up when 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, it will actually evaluate to true. As far as the runtime system is concerned, the class of both objects is `ArrayList`.

Generic Classes and Static Attributes

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

If you are interested in the details of Java and type safety, you might like this little bit of Java trivia: 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.

The problem is the same problem I discussed earlier in this article: If `B` is a subtype of `A`, is then `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 we 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.ArrayStoreException: A
```

The last line in this example represents a type error: I am trying to insert an `A` object into an array of `B`. The problem is that 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 it designed 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 all of the concepts together, the class and method definitions can become quite tricky to read even for experienced programmers. Have a 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. Do not let this discourage you; 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. `</article>`

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.

learn more

The Java Tutorial on generic types

FEATURED JDK ENHANCEMENT PROPOSAL

JEP 282 jlink: The Java Linker

For most readers, the idea of a linker for Java might seem very peculiar indeed. Linker functions, which are part of a build tool associated with native languages, are performed by the JVM in its class-loading mechanism. In particular, these functions are executed in the algorithms for finding JARs that contain referred-to classes and methods and then loading them into the current JVM memory space. [For more information on this process, download a PDF of our article "How the JVM Locates, Loads, and Runs Libraries" by Oleg Šelajev. —*Ed.*]

What JEP 282 proposes is not the traditional linker but, rather, a generic tool that runs where a linker does in the build process—after the compiler but before creation of the executable. The tool would define a plugin interface, by which a variety of tools could be inserted into the build process. The most obvious of these would be an optimizer, especially a whole-program optimizer that could identify opportunities to improve performance and reduce code size that are not visible to the compiler on a class basis. Other plugins suggested in the JEP document could remove debug information, reorder resources so that they can be loaded faster, and even compress generated files.

In theory, many other refinements to generated code could be performed—including those from third parties. Some examples are insertion of instrumentation data, supplementation of debugging data, conversion of bytecodes to other formats, intraclass optimization, and so on. All of this could be done through plugins to the proposed jlink technology.