Understanding Java generics, Part 1: Principles and fundamentals
Ancient history: Heterogeneous collections
Java and type safety
Generics to the rescue
Abstraction over types
Improved correctness and better readability
Shortening code with the diamond notation
Generics and subtyping
Conclusion
Dig deeper

CODING

Understanding Java generics, Part 1: Principles and fundamentals

Learn how to use generics to increase type safety and code readability.

by Michael Kölling

April 30, 2021

[*Java Magazine* is pleased to republish this tutorial from Michael Kölling, published in 2016, about generics. —*Ed*.]

If you have programmed for a little while in Java, it is likely that you have come across generics, and you have probably used them. (Generics are hard to avoid when using collections, and it is hard to do anything interesting without collections.) If you are coming to Java from C++, you might have encountered the same concept as generics under the name of *parameterized types* or *templates*. (Templates in C++ are not the same as generics in all aspects, but they are closely related.)

Many novice Java programmers use generics without a full understanding of how generics work and what they can do. This gap is what I address in this article.

In Java, the concept of generics is simple and straightforward in principle but tricky in the details. There is much to say about the corner cases, and it is also interesting to investigate how generics are implemented in the Java compiler and the JVM. This knowledge helps you understand and anticipate some of the more surprising behaviors.

This discussion is spread over two parts. In this article, I discuss the principles and fundamental ideas of generic types. I look at the definition and use of generics and provide a basic, overall understanding. In the second article, I will look at the more subtle parts, advanced uses, and implementation. If you read both articles, you will arrive at a good understanding of how generics can help you write better code.

Ancient history: Heterogeneous collections

Before Java 5 was released in 2004, Java did not have generic types. Instead, it had what are called *heterogeneous collections*, such as lists. The following is an example of what a heterogeneous list looked like in those days:

```
List students = new ArrayList();
```

(Knowing this history is important, because you can still write this code in Java today —even though you shouldn't. And you might come across these collections in legacy code you maintain.)

In the example above, I intend to create a list of `Student` objects. I am using subtyping—the declared type of the variable is the interface `List`, a supertype of `ArrayList`, which is the actual object type created. This approach is a good idea because it increases flexibility. I can then add the `Student` objects to the list, as follows:

```
students.add(new Student("Fred"));
```

When the time comes to get the `Student` object out of the list again, the most natural thing to write would be this.

```
Student s = students.get(0);
```

This, however, does not work! In Java, in order to give the `List` type the ability to hold elements of any type, the `add` method was defined to take a parameter of type `Object`, and the `get` method appropriately also returns an `Object` type. This makes the previous statement an attempt to assign an expression of type `Object` to a variable of type `Student`—which is an error.

In the early days of Java, this error was fixed with a cast.

```
Student s = (Student) students.get(0);
```

Writing casts was always a mild annoyance. As the developer, you usually know what types of objects are stored in any particular list. Why can't the compiler keep track of them as well? But the problem goes deeper than mere annoyance. Because the element type of the list is declared to be `Object`, you are allowed to add objects of any type to the same list.

```
students.add(new Integer(42));
students.add("a string");
```

The fact that this is *possible*—that the same list can hold objects of different types—is the reason the list is referred to as heterogeneous: A list can contain mixed element types.

Having lists of different, unrelated element types is rarely useful, but it can easily be done in error. The problem with heterogeneous lists is that this error cannot be detected until runtime. Nothing prevents you from accidentally inserting the wrong element type into the student list. Worse, even if you get the element out of the list and cast it to a `Student`, the code compiles. The error surfaces only at runtime when the cast fails. Then a runtime type error is generated.

The problem with this runtime error is not only that it occurs late (at runtime when the application might already have been delivered to a customer) but also that the source location of the detected error might be far removed from the actual mistake. You are notified about the problem when getting the element *out* of the list, while the actual error was made when putting the element *in*. The root cause, therefore, might well be in a different part of the program entirely.

Java and type safety

Java was always intended to be a type-safe language. This means that type errors should be detected at compile time and runtime type errors should not be possible. This aim was never achieved completely, but it's a goal that the language designers strove for as much as possible. Casting breaks this goal: Every time you use a cast, you punch a hole in type safety. You tell the type checker to look the other way and just trust you. But there is no guarantee that you will get things right.

Many times, when a cast is used, the code can be rewritten: Often better object-oriented techniques can be used to avoid casting and maintain type safety. However, collections presented a different problem. There was no way to use them without casting, and this jarred with the philosophy of Java. That such an important area of programming could not be used in a type-safe way was a real annoyance. Thus in 2004, the Java language team fixed this problem by adding generics to the Java language.

The term for the problem with heterogeneous collections is *type loss*. To construct collections of any type, the element type of all collections was defined to be `Object`. The `add` method, for example, might be defined as follows:

```
public void add(Object element)
```

This works very well to put elements—say, objects of type `Student`—into the collection, but it creates a problem when you want to get them out again. Even though the runtime system will know that the element is of type `Student`, the compiler does not keep track of this. When you later use a `get` method to retrieve the

element, all the compiler knows is that the element type is `Object`. The compiler loses track of the actual element type—thus type loss occurs.

Generics to the rescue

The solution to the type loss problem was to give the compiler enough information about the element type. This was done by adding a `type` parameter to the class or interface definition. Consider an incomplete and simplified definition of an `ArrayList`. Before generics, it might have looked like the following:

```
class ArrayList {
      public void add(Object element);
      public Object get(int index);
}
```

The element type here is `Object`. Now, with the generic parameter, the definition looks as follows:

```
class ArrayList<E> {
      public void add(E element);
      public E get(int index);
}
```

The `E` in the angle brackets is the *type* parameter: Here, you can specify what the element type of the list should be. With generics, you no longer create an `ArrayList` object for the `Student` elements by writing the following:

```
new ArrayList()
```

Instead, you now write this.

```
new ArrayList<Student>()
```

Just as with parameters for methods, you have a formal parameter specification in the definition (the `E`) and an actual parameter at the point of use (`Student`). Unlike in method parameters, the actual parameter is *not a value but a type*.

By creating an `ArrayList <Student>` (which is usually read out loud as “an `ArrayList` of `Student`”), the other mentions of the type parameter `E` in the specification are also replaced with the actual type parameter `Student`. Thus, the parameter type of the `add` method and the return type of the `get` method are now both `Student`. This is very useful because now only `Student` objects can be added as elements. You will retrieve only `Student` objects when you get them out again—no casting is needed, and there are no runtime errors either.

Abstraction over types

It is useful to understand one aspect that changed slightly when generics entered the Java language: the relationship between classes and types. Prior to generics, each class defined a type. For example, if you define a class `Hexagon`, then you automatically get a type called `Hexagon` to use in variable and parameter definitions. There is a very simple one-to-one relationship.

With generic classes, this is different. A generic class does not define a type. Rather, it defines a set of types. For example, the class `ArrayList<E>` defines the types `ArrayList<Student>`, `ArrayList<Integer>`, `ArrayList<String>`, `ArrayList<ArrayList<String>>`, and all other types that can be specified by replacing the type parameter `E` with a concrete type.

In other words, generics introduce an abstraction over types—a powerful new feature.

Improved correctness and better readability

One benefit of using generic classes should now be obvious: improved correctness. Incorrect types of objects can no longer be entered into a list. While erroneous attempts to insert an element could previously be detected only during testing (and

testing can never be complete), they are now detected at compile time, and type correctness is guaranteed. In addition, if there is such an error, it will be reported at the point of the incorrect insertion—not at the point of retrieving the element, which is far removed from the actual error location.

There is, however, a second benefit: readability. By explicitly specifying the element type of collections, you are providing useful information to human readers of your program as well. Explicitly saying what type of element a collection is intended for can make life easier for a maintenance programmer.

Shortening code with the diamond notation

When generics were introduced, a useful shortcut notation—the *diamond notation*—was provided to ensure that the increased code readability does not lead to unnecessary verboseness.

Consider the very common case of declaring a variable and initializing it with a newly created object.

```
ArrayList<String> myList =
       new ArrayList<String>();
```

In some generic types, especially when there is more than one generic parameter, this line can get rather long.

```
HashSet<Integer, String> mySet =
        new HashSet<Integer, String>();
```

And it can get worse if a type parameter itself is generic.

```
HashSet<Integer, ArrayList<String>> mySet =
        new HashSet<Integer, ArrayList<String>>();
```

In each of these examples, the same lengthy generic type is spelled out twice: once on the left for the variable declaration and once on the right for the object creation. In this situation, the Java compiler allows you to omit part of the second mention of the type and instead write the following:

```
HashSet<Integer, String> mySet = new HashSet<>();
```

Here, the generic parameters are omitted from the right side (leaving the angle brackets to form a diamond shape, thus the term *diamond notation*). This is allowed in this situation and means that the generic parameters on the right are the same as those on the left. It just saves some typing and makes the line easier to read. The semantics are the same as they would be had you written out the types in full.

Generics and subtyping

This was the easy part. The use of generics can make code safer and easier to read. Writing a simple generic class is quite straightforward, as is creating objects of generic types. However, the story does not end here. So far, I have ignored some problems that arise with generics, and understanding the mechanisms to solve them gets a little trickier. Let's look at the problem first.

Assume that you have a small inheritance hierarchy. To model people in a university, you have classes `Student` and `Faculty` and a common superclass `Person`. `Student` and `Faculty` are both subclasses of `Person`. So far, so good.

Now you also create types for lists of each of these: `List<Student>`, `List<Faculty>`, and `List<Person>`.

The student and faculty lists are held in the parts of your program that hold and process the `Student` and `Faculty` objects. The `Person` list type can be useful as a formal parameter for a method that you want to use with both faculty and students, for example

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

The idea is that you want to be able to call `printList` with both the student and faculty lists as actual parameters. This should work if the types of these lists are subtypes of `List<Person>`, but are they? In other words, if `Student` is a subtype of `Person`, is then `List<Student>` a subtype of `List<Person>`?

Unfortunately, the correct answer is no.

Imagine that the `printList` method also modifies the list passed as a parameter. Assume that this method inserts an object of type `Faculty` into the list. Because the list is declared in the parameter as `List<Person>`, and `Faculty` is a subtype of `Person`, this is perfectly legal. However, the actual list passed to this method might have been a `List<Student>`. Then, suddenly, a `Faculty` object has been inserted into the student list! This is clearly a problem.

The only way to avoid this problem is to avoid considering lists of subtypes and lists of supertypes to be in a subtype/supertype relationship themselves. In other words, `List<Student>` is *not* a subtype of `List<Person>`.

Conclusion

There are many situations in which you need subtyping with generic types, such as the above attempt to define the generalized `printList` method. You have seen that it does not work with the constructs I have discussed so far, but just saying it can't be done is not good enough—you do need to be able to write such code on occasion.

The solutions entail additional constructs for generics: bounded types and wildcards. These concepts are powerful but have some rather tricky corner cases. I will discuss them in part 2 of this tutorial. Until then, study the generic classes available in the Java library—especially the collections—and get used to the notation discussed in this article. I will go deeper next time!

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.