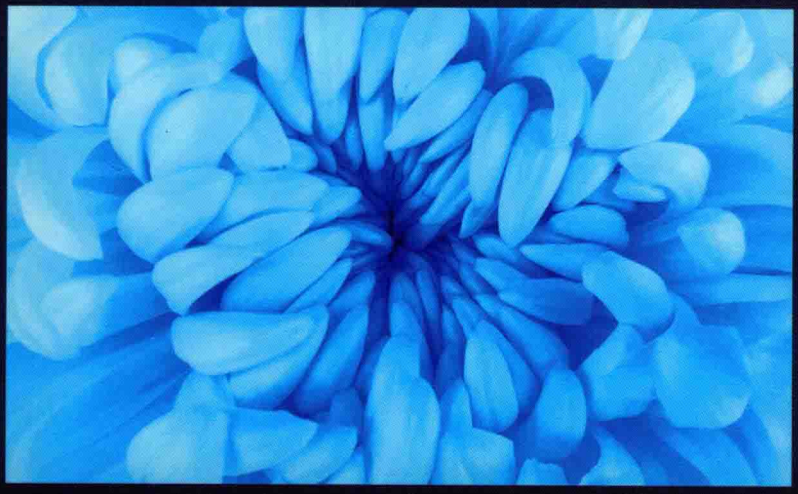
Pearson

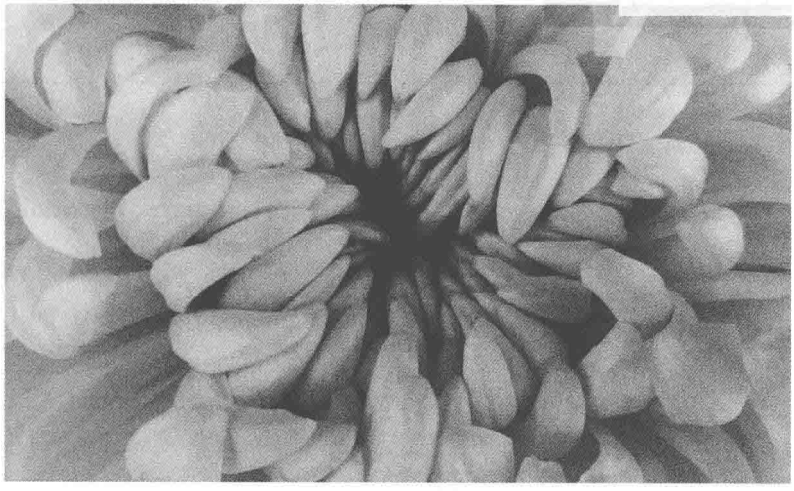


Java核心技术
卷 I：基础知识 下
（第 11 版 · 英文版）

[美] 凯 · S. 霍斯特曼（Cay S. Horstmann） 著

Core Java
Volume I—Fundamentals, Eleventh Edition

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS



Java核心技术
卷 I：基础知识 下
（第 11 版 · 英文版）

[美] 凯 · S. 霍斯特曼（Cay S. Horstmann） 著

Core Java
Volume I—Fundamentals, Eleventh Edition

人 民 邮 电 出 版 社
北 京

目录

CHAPTER 8

Generic Programming

In this chapter

Generic classes and methods have type parameters. This allows them to describe precisely what should happen when they are instantiated with specific types. Prior to generic classes, programmers had to use the `Object` for writing code that works with multiple types. This was both cumbersome and unsafe.

With the introduction of generics, Java has an expressive type system that allows designers to describe in detail how types of variables and methods should vary. In straightforward situations, you will find it simple to implement generic code. In more advanced cases, it can get quite complex—for implemetors. The goal is to provide classes and methods that other programmers can use without surprises.

The introduction of generics in Java 5 constitutes the most significant change in the Java programming language since its initial release. A major design goal was to be backwards compatible with earlier releases. As a result, Java generics have some uncomfortable limitations. You will learn about the benefits and challenges of generic programming in this chapter.

8.1 Why Generic Programming?

Generic programming means writing code that can be reused for objects of many different types. For example, you don't want to program separate classes to collect `String` and `File` objects. And you don't have to—the single class `ArrayList` collects objects of any class. This is one example of generic programming.

Actually, Java had an `ArrayList` class before it had generic classes. Let us investigate how the mechanism for generic programming has evolved, and what that means for users and implementors.

8.1.1 The Advantage of Type Parameters

Before generic classes were added to Java, generic programming was achieved with *inheritance*. The `ArrayList` class simply maintained an array of `Object` references:

```
public class ArrayList // before generic classes
{
   private Object[] elementData;
   . . .
   public Object get(int i) { . . . }
   public void add(Object o) { . . . }
}
```

This approach has two problems. A cast is necessary whenever you retrieve a value:

```
ArrayList files = new ArrayList();
. . .
String filename = (String) files.get(0);
```

Moreover, there is no error checking. You can add values of any class:

```
files.add(new File(". . ."));
```

This call compiles and runs without error. Elsewhere, casting the result of `get` to a `String` will cause an error.

Generics offer a better solution: *type parameters*. The ArrayList class now has a type parameter that indicates the element type:

```
var files = new ArrayList<String>();
```

This makes your code easier to read. You can tell right away that this particular array list contains String objects.

NOTE: If you declare a variable with an explicit type instead of var, you can omit the type parameter in the constructor by using the "diamond" syntax:

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

The omitted type is inferred from the type of the variable.

Java 9 expands the use of the diamond syntax to situations where it was previously not accepted. For example, you can now use diamonds with anonymous subclasses:

```
ArrayList<String> passwords = new ArrayList<>() // diamond OK in Java 9
   {
      public String get(int n) { return super.get(n).replaceAll(".", "*"); }
   };
```

The compiler can make good use of the type information too. No cast is required for calling get. The compiler knows that the return type is String, not Object:

```
String filename = files.get(0);
```

The compiler also knows that the add method of an ArrayList<String> has a parameter of type String. That is a lot safer than having an Object parameter. Now the compiler can check that you don't insert objects of the wrong type. For example, the statement

```
files.add(new File(". . ."); // can only add String objects to an ArrayList<String>
```

will not compile. A compiler error is much better than a class cast exception at runtime.

This is the appeal of type parameters: They make your programs easier to read and safer.

8.1.2 Who Wants to Be a Generic Programmer?

It is easy to use a generic class such as ArrayList. Most Java programmers will simply use types such as ArrayList<String> as if they had been built into the

language, just like `String[]` arrays. (Of course, array lists are better than arrays because they can expand automatically.)

However, it is not so easy to implement a generic class. The programmers who use your code will want to plug in all sorts of classes for your type parameters. They will expect everything to work without onerous restrictions and confusing error messages. Your job as a generic programmer, therefore, is to anticipate all the potential future uses of your class.

How hard can this get? Here is a typical issue that the designers of the standard class library had to grapple with. The `ArrayList` class has a method `addAll` to add all elements of another collection. A programmer may want to add all elements from an `ArrayList<Manager>` to an `ArrayList<Employee>`. But, of course, doing it the other way round should not be legal. How do you allow one call and disallow the other? The Java language designers invented an ingenious new concept, the *wildcard type*, to solve this problem. Wildcard types are rather abstract, but they allow a library builder to make methods as flexible as possible.

Generic programming falls into three skill levels. At a basic level, you just use generic classes—typically, collections such as `ArrayList`—without thinking how and why they work. Most application programmers will want to stay at that level until something goes wrong. You may, however, encounter a confusing error message when mixing different generic classes, or when interfacing with legacy code that knows nothing about type parameters; at that point, you'll need to learn enough about Java generics to solve problems systematically rather than through random tinkering. Finally, of course, you may want to implement your own generic classes and methods.

Application programmers probably won't write lots of generic code. The JDK developers have already done the heavy lifting and supplied type parameters for all the collection classes. As a rule of thumb, only code that traditionally involved lots of casts from very general types (such as `Object` or the `Comparable` interface) will benefit from using type parameters.

In this chapter, we will show you everything you need to know to implement your own generic code. However, we expect that most readers will use this knowledge primarily for help with troubleshooting and to satisfy their curiosity about the inner workings of the parameterized collection classes.

8.2 Defining a Simple Generic Class

A *generic class* is a class with one or more type variables. In this chapter, we will use a simple `Pair` class as an example. This class allows us to focus on

generics without being distracted by data storage details. Here is the code for the generic Pair class:

```
public class Pair<T>
{
   private T first;
   private T second;

   public Pair() { first = null; second = null; }
   public Pair(T first, T second) { this.first = first; this.second = second; }

   public T getFirst() { return first; }
   public T getSecond() { return second; }

   public void setFirst(T newValue) { first = newValue; }
   public void setSecond(T newValue) { second = newValue; }
}
```

The Pair class introduces a type variable T, enclosed in angle brackets < >, after the class name. A generic class can have more than one type variable. For example, we could have defined the Pair class with separate types for the first and second field:

```
public class Pair<T, U> { . . . }
```

The type variables are used throughout the class definition to specify method return types and the types of fields and local variables. For example:

```
private T first; // uses the type variable
```

NOTE: It is common practice to use uppercase letters for type variables, and to keep them short. The Java library uses the variable E for the element type of a collection, K and V for key and value types of a table, and T (and the neighboring letters U and S, if necessary) for "any type at all."

You *instantiate* the generic type by substituting types for the type variables, such as

```
Pair<String>
```

You can think of the result as an ordinary class with constructors

```
Pair<String>()
Pair<String>(String, String)
```

and methods

```
String getFirst()
String getSecond()
```

```
void setFirst(String)
void setSecond(String)
```

In other words, the generic class acts as a factory for ordinary classes.

The program in Listing 8.1 puts the `Pair` class to work. The static `minmax` method traverses an array and simultaneously computes the minimum and maximum values. It uses a `Pair` object to return both results. Recall that the `compareTo` method compares two strings, returning `0` if the strings are identical, a negative integer if the first string comes before the second in dictionary order, and a positive integer otherwise.

C++ NOTE: Superficially, generic classes in Java are similar to template classes in C++. The only obvious difference is that Java has no special `template` keyword. However, as you will see throughout this chapter, there are substantial differences between these two mechanisms.

Listing 8.1 `pair1/PairTest1.java`

```
1  package pair1;
2
3  /**
4   * @version 1.01 2012-01-26
5   * @author Cay Horstmann
6   */
7  public class PairTest1
8  {
9     public static void main(String[] args)
10    {
11       String[] words = { "Mary", "had", "a", "little", "lamb" };
12       Pair<String> mm = ArrayAlg.minmax(words);
13       System.out.println("min = " + mm.getFirst());
14       System.out.println("max = " + mm.getSecond());
15    }
16 }
17
18 class ArrayAlg
19 {
20    /**
21     * Gets the minimum and maximum of an array of strings.
22     * @param a an array of strings
23     * @return a pair with the min and max values, or null if a is null or empty
24     */
25    public static Pair<String> minmax(String[] a)
26    {
27       if (a == null || a.length == 0) return null;
28       String min = a[0];
```

```
29        String max = a[0];
30        for (int i = 1; i < a.length; i++)
31        {
32           if (min.compareTo(a[i]) > 0) min = a[i];
33           if (max.compareTo(a[i]) < 0) max = a[i];
34        }
35        return new Pair<>(min, max);
36     }
37  }
```

8.3 Generic Methods

In the preceding section, you have seen how to define a generic class. You can also define a single method with type parameters.

```
class ArrayAlg
{
   public static <T> T getMiddle(T... a)
   {
      return a[a.length / 2];
   }
}
```

This method is defined inside an ordinary class, not inside a generic class. However, it is a generic method, as you can see from the angle brackets and the type variable. Note that the type variables are inserted after the modifiers (`public static`, in our case) and before the return type.

You can define generic methods both inside ordinary classes and inside generic classes.

When you call a generic method, you can place the actual types, enclosed in angle brackets, before the method name:

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");
```

In this case (and indeed in most cases), you can omit the `<String>` type parameter from the method call. The compiler has enough information to infer the method that you want. It matches the type of the arguments against the generic type `T...` and deduces that `T` must be `String`. That is, you can simply call

```
String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

In almost all cases, type inference for generic methods works smoothly. Occasionally, the compiler gets it wrong, and you'll need to decipher an error report. Consider this example:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

The error message complains, in cryptic terms that vary from one compiler version to another, that there are two ways of interpreting this code, both equally valid. In a nutshell, the compiler autoboxed the parameters into a `Double` and two `Integer` objects, and then it tried to find a common supertype of these classes. It actually found two: `Number` and the `Comparable` interface, which is itself a generic type. In this case, the remedy is to write all parameters as `double` values.

TIP: Peter von der Ahé recommends this trick if you want to see which type the compiler infers for a generic method call: Purposefully introduce an error and study the resulting error message. For example, consider the call `ArrayAlg.getMiddle("Hello", 0, null)`. Assign the result to a `JButton`, which can't possibly be right. You will get an error report:

```
found:
java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends
java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>
```

In plain English, you can assign the result to `Object`, `Serializable`, or `Comparable`.

C++ NOTE: In C++, you place the type parameters after the method name. That can lead to nasty parsing ambiguities. For example, `g(f<a,b>(c))` can mean "call `g` with the result of `f<a,b>(c)`", or "call `g` with the two `boolean` values `f<a` and `b>(c)`".

8.4 Bounds for Type Variables

Sometimes, a class or a method needs to place restrictions on type variables. Here is a typical example. We want to compute the smallest element of an array:

```
class ArrayAlg
{
   public static <T> T min(T[] a) // almost correct
   {
      if (a == null || a.length == 0) return null;
      T smallest = a[0];
      for (int i = 1; i < a.length; i++)
         if (smallest.compareTo(a[i]) > 0) smallest = a[i];
      return smallest;
   }
}
```