



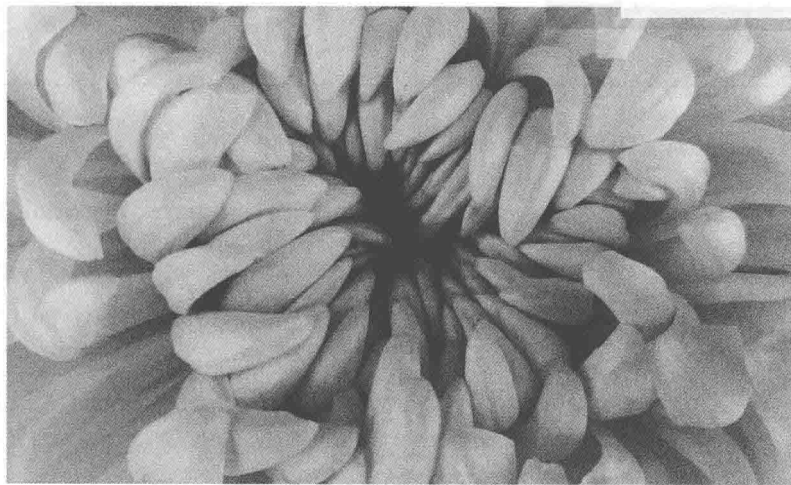
Java核心技术

卷I：基础知识

（第11版·英文版）

[美] 凯·S. 霍斯特曼 (Cay S. Horstmann) 著

Core Java
Volume I—Fundamentals, Eleventh Edition



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 / 泛型编程	431
8.1 Why Generic Programming? / 为什么要使用泛型编程	432
8.1.1 The Advantage of Type Parameters / 类型参数的好处	432
8.1.2 Who Wants to Be a Generic Programmer? / 哪些人想成为泛型程序员	433
8.2 Defining a Simple Generic Class / 定义简单的泛型类	434
8.3 Generic Methods / 泛型方法	437
8.4 Bounds for Type Variables / 类型变量的绑定	438
8.5 Generic Code and the Virtual Machine / 泛型代码与虚拟机	441
8.5.1 Type Erasure / 类型擦除	441
8.5.2 Translating Generic Expressions / 翻译泛型表达式	442
8.5.3 Translating Generic Methods / 翻译泛型方法	443
8.5.4 Calling Legacy Code / 调用遗留代码	445
8.6 Restrictions and Limitations / 约束与局限性	447
8.6.1 Type Parameters Cannot Be Instantiated with Primitive Types / 类型参数不能用基本类型来实例化	447
8.6.2 Runtime Type Inquiry Only Works with Raw Types / 运行时类型查询只适用于原始类型	447
8.6.3 You Cannot Create Arrays of Parameterized Types / 不能创建参数化类型的数组	448
8.6.4 Varargs Warnings / 注意变长参数情况	448
8.6.5 You Cannot Instantiate Type Variables / 不能实例化类型变量	450
8.6.6 You Cannot Construct a Generic Array / 不能构造泛型数组	451
8.6.7 Type Variables Are Not Valid in Static Contexts of Generic Classes / 类型变量在泛型类的静态上下文中无效	452

8.6.8	You Cannot Throw or Catch Instances of a Generic Class / 不能抛出或捕获泛型类的实例	453
8.6.9	You Can Defeat Checked Exception Checking / 可以打破“检查型异常必须检查”的规则	454
8.6.10	Beware of Clashes after Erasure / 意类型擦除后的冲突	455
8.7	Inheritance Rules for Generic Types / 泛型类型的继承规则	457
8.8	Wildcard Types / 通配符类型	459
8.8.1	The Wildcard Concept / 通配符的概念	459
8.8.2	Supertype Bounds for Wildcards / 通配符的超类型限定	461
8.8.3	Unbounded Wildcards / 无限定通配符	464
8.8.4	Wildcard Capture / 通配符捕获	465
8.9	Reflection and Generics / 反射与泛型	467
8.9.1	The Generic Class Class / 泛型的 Class 类	467
8.9.2	Using Class<T> Parameters for Type Matching / 使用 Class<T>参数进行类型匹配	469
8.9.3	Generic Type Information in the Virtual Machine / 虚拟机中的泛型类型信息	469
8.9.4	Type Literals / TypeLiteral	473
Chapter 9: Collections / 集合类		481
9.1	The Java Collections Framework / Java 集合类框架	482
9.1.1	Separating Collection Interfaces and Implementation / 将集合类的接口与实现分离	482
9.1.2	The Collection Interface / Collection 接口	485
9.1.3	Iterators / 迭代器	485
9.1.4	Generic Utility Methods / 泛型的实用方法	489
9.2	Interfaces in the Collections Framework / 集合类框架中的接口	492
9.3	Concrete Collections / 具体的集合类	494
9.3.1	Linked Lists / 链表	496
9.3.2	Array Lists / 数组列表	507
9.3.3	Hash Sets / 散列集	507
9.3.4	Tree Sets / 树形集	511
9.3.5	Queues and Deques / 队列与双端队列	516

9.3.6	Priority Queues / 优先级队列	518
9.4	Maps / 映射	519
9.4.1	Basic Map Operations / 基本映射操作	519
9.4.2	Updating Map Entries / 更新映射表项	523
9.4.3	Map Views / 映射视图	525
9.4.4	Weak Hash Maps / 弱散列映射	526
9.4.5	Linked Hash Sets and Maps / LinkedHashSet 与 LinkedHashMap	527
9.4.6	Enumeration Sets and Maps / EnumSet 与 EnumMap	529
9.4.7	Identity Hash Maps / IdentityHashMap	530
9.5	Views and Wrappers / 视图与包装器	532
9.5.1	Small Collections / 小型集合	532
9.5.2	Subranges / 子范围	534
9.5.3	Unmodifiable Views / 不可修改视图	535
9.5.4	Synchronized Views / 同步视图	536
9.5.5	Checked Views / 检查用视图	536
9.5.6	A Note on Optional Operations / 可选操作说明	537
9.6	Algorithms / 算法	541
9.6.1	Why Generic Algorithms? / 为什么要使用泛型算法	541
9.6.2	Sorting and Shuffling / 排序与混排	543
9.6.3	Binary Search / 二分查找	546
9.6.4	Simple Algorithms / 简单算法	547
9.6.5	Bulk Operations / 主要操作	549
9.6.6	Converting between Collections and Arrays / 集合与数组之间的转换	550
9.6.7	Writing Your Own Algorithms / 编写自己的算法	551
9.7	Legacy Collections / 遗留的集合类	552
9.7.1	The Hashtable Class / Hashtable 类	553
9.7.2	Enumerations / Enumeration	553
9.7.3	Property Maps / 属性映射	555
9.7.4	Stacks / 栈	558
9.7.5	Bit Sets / 位集	559

Chapter 10: Graphical User Interface Programming /	
图形用户界面编程	565
10.1 A History of Java User Interface Toolkits /	
Java 用户界面工具包的历史	565
10.2 Displaying Frames / 显示框架	567
10.2.1 Creating a Frame / 创建框架	568
10.2.2 Frame Properties / 框架属性	570
10.3 Displaying Information in a Component /	
在组件中显示信息	574
10.3.1 Working with 2D Shapes / 处理 2D 图形	579
10.3.2 Using Color / 使用颜色	587
10.3.3 Using Fonts / 使用字体	589
10.3.4 Displaying Images / 显示图片	597
10.4 Event Handling / 事件处理	598
10.4.1 Basic Event Handling Concepts /	
事件处理的基本概念	598
10.4.2 Example: Handling a Button Click /	
示例: 处理按钮点击事件	600
10.4.3 Specifying Listeners Concisely /	
设置监听器的简洁方法	604
10.4.4 Adapter Classes / 适配器类	605
10.4.5 Actions / 动作	608
10.4.6 Mouse Events / 鼠标事件	614
10.4.7 The AWT Event Hierarchy / AWT 事件层次	620
10.5 The Preferences API / Preferences API	624
Chapter 11: User Interface Components with Swing /	
Swing 用户界面组件	631
11.1 Swing and the Model-View-Controller Design Pattern /	
Swing 与模型-视图-控制器设计模式	632
11.2 Introduction to Layout Management /	
布局管理简介	636

11.2.1	Layout Managers / 布局管理器	637
11.2.2	Border Layout / 边框布局	639
11.2.3	Grid Layout / 网格布局	642
11.3	Text Input / 文本输入	643
11.3.1	Text Fields / 文本框	643
11.3.2	Labels and Labeling Components / 标签与标签组件	645
11.3.3	Password Fields / 密码框	647
11.3.4	Text Areas / 文本区域	647
11.3.5	Scroll Panes / 滚动窗格	648
11.4	Choice Components / 选择组件	651
11.4.1	Checkboxes / 复选框	651
11.4.2	Radio Buttons / 单选按钮	654
11.4.3	Borders / 边框	658
11.4.4	Combo Boxes / 组合框	661
11.4.5	Sliders / 滑动条	665
11.5	Menus / 菜单	671
11.5.1	Menu Building / 菜单构建	672
11.5.2	Icons in Menu Items / 菜单项中的图标	675
11.5.3	Checkbox and Radio Button Menu Items / 复选框和单选按钮菜单项	676
11.5.4	Pop-Up Menus / 弹出菜单	677
11.5.5	Keyboard Mnemonics and Accelerators / 键盘助记符与快捷键	679
11.5.6	Enabling and Disabling Menu Items / 启用和禁用菜单项	682
11.5.7	Toolbars / 工具栏	687
11.5.8	Tooltips / 工具提示	689
11.6	Sophisticated Layout Management / 复杂的布局管理	690
11.6.1	The Grid Bag Layout / 网格袋布局	691
11.6.2	Custom Layout Managers / 定制布局管理器	702
11.7	Dialog Boxes / 对话框	706
11.7.1	Option Dialogs / 选项对话框	707
11.7.2	Creating Dialogs / 创建对话框	712

11.7.3	Data Exchange / 数据交换	716
11.7.4	File Dialogs / 文件对话框	723
Chapter 12: Concurrency / 并发		733
12.1	What Are Threads? / 什么是线程	734
12.2	Thread States / 线程状态	739
12.2.1	New Threads / 新创建线程	740
12.2.2	Runnable Threads / 可运行线程	740
12.2.3	Blocked and Waiting Threads / 被阻塞线程与等待线程	741
12.2.4	Terminated Threads / 被终止的线程	742
12.3	Thread Properties / 线程属性	743
12.3.1	Interrupting Threads / 中断线程	743
12.3.2	Daemon Threads / 守护线程	746
12.3.3	Thread Names / 线程名	747
12.3.4	Handlers for Uncaught Exceptions / 未捕获异常的处理程序	747
12.3.5	Thread Priorities / 线程优先级	749
12.4	Synchronization / 同步	750
12.4.1	An Example of a Race Condition / 竞争条件的一个案例	750
12.4.2	The Race Condition Explained / 竞争条件详解	752
12.4.3	Lock Objects / 锁对象	755
12.4.4	Condition Objects / 条件对象	758
12.4.5	The synchronized Keyword / synchronized 关键字	764
12.4.6	Synchronized Blocks / 同步块	768
12.4.7	The Monitor Concept / 监视器概念	770
12.4.8	Volatile Fields / volatile 字段	771
12.4.9	Final Variables / final 变量	772
12.4.10	Atomics / 原子	773
12.4.11	Deadlocks / 死锁	775
12.4.12	Thread-Local Variables / 线程局部变量	778

12.4.13	Why the stop and suspend Methods Are Deprecated / 为什么弃用 stop 和 suspend 方法	779
12.5	Thread-Safe Collections / 线程安全的集合	781
12.5.1	Blocking Queues / 阻塞队列	781
12.5.2	Efficient Maps, Sets, and Queues / 高效的映射、集和队列	789
12.5.3	Atomic Update of Map Entries / 映射表项的原子更新	790
12.5.4	Bulk Operations on Concurrent Hash Maps / 并发散列映射上的主要操作	794
12.5.5	Concurrent Set Views / 并发的集视图	796
12.5.6	Copy on Write Arrays / 写时复制的数组	797
12.5.7	Parallel Array Algorithms / 并行数组算法	797
12.5.8	Older Thread-Safe Collections / 较早的线程安全的集合	799
12.6	Tasks and Thread Pools / 任务和线程池	800
12.6.1	Callables and Futures / Callable 与 Future	800
12.6.2	Executors / 执行器	802
12.6.3	Controlling Groups of Tasks / 控制任务组	806
12.6.4	The Fork-Join Framework / Fork-Join 框架	811
12.7	Asynchronous Computations / 异步计算	814
12.7.1	Completable Futures / CompletableFuture	815
12.7.2	Composing Completable Futures / 组合 CompletableFuture	817
12.7.3	Long-Running Tasks in User Interface Callbacks / 用户界面回调中的长期运行任务	823
12.8	Processes / 进程	831
12.8.1	Building a Process / 构建进程	832
12.8.2	Running a Process / 运行进程	834
12.8.3	Process Handles / 进程 handle	835

Generic Programming

In this chapter

- 8.1 Why Generic Programming?, page 432
- 8.2 Defining a Simple Generic Class, page 434
- 8.3 Generic Methods, page 437
- 8.4 Bounds for Type Variables, page 438
- 8.5 Generic Code and the Virtual Machine, page 441
- 8.6 Restrictions and Limitations, page 447
- 8.7 Inheritance Rules for Generic Types, page 457
- 8.8 Wildcard Types, page 459
- 8.9 Reflection and Generics, page 467

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 implementors. 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;
    }
}
```