Chapter 17

Generic Programming

CHAPTER GOALS

- To understand the objective of generic programming
- To be able to implement generic classes and methods
- To understand the execution of generic methods in the virtual machine
- To know the limitations of generic programming in Java

Generic programming involves the design and implementation of data structures and algorithms that work for multiple types. You are already familiar with the generic `ArrayList` class that can be used to collect elements of arbitrary types. In this chapter, you will learn how to implement your own generic classes.

CHAPTER CONTENTS

17.1 Generic Classes and Type Parameters

Generic programming is the creation of programming constructs that can be used with many different types. For example, the Java library programmers who implemented the `ArrayList` class used the technique of generic programming. As a result, you can form array lists that collect elements of different types, such as `ArrayList<String>`, `ArrayList<BankAccount>`, and so on.

The `LinkedList` class that we implemented in Section 15.2 is also an example of generic programming—you can store objects of any class inside a `LinkedList`. That `LinkedList` class achieves genericity by using *inheritance*. It uses references of type `Object` and is therefore capable of storing objects of any class. In contrast, the `ArrayList` class is a *generic class:* a class with a **type parameter** that is used to specify the type of the objects that you want to store. (Note that only our `LinkedList` implementation of Chapter 15 uses inheritance. The standard Java library has a generic `LinkedList` class that uses type parameters.)

In Java, generic programming can be achieved with inheritance or with type parameters.

When declaring a generic class, you specify a type variable for each type parameter. Here is how the standard Java library declares the `ArrayList` class, using the type variable `E` for the element type:

A generic class has one or more type parameters.

```
public class ArrayList<E>
{
   public ArrayList() { . . . }
   public void add(E element) { . . . }
   . . .
}
```

Here, `E` is a type variable, not a Java reserved word. You could use another name, such as `ElementType`, instead of `E`. However, it is customary to use short, uppercase names for type parameters.

In order to use a generic class, you need to *instantiate* the type parameter, that is, supply an actual type. You can supply any class or interface type, for example

Type parameters can be instantiated with class or interface types.

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

However, you cannot substitute any of the eight primitive types for a type parameter. It would be an error to declare an `ArrayList<double>`. Use the corresponding wrapper class instead, such as `ArrayList<Double>`.

When you instantiate a generic class, the type that you supply replaces all occurrences of the type variable in the declaration of the class. For example, the `add`

method for `ArrayList<BankAccount>` has the type variable `E` replaced with the type `BankAccount`:

```
public void add(BankAccount element)
```

Contrast that with the `add` method of the `LinkedList` class in Chapter 15:

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

The `add` method of the generic `ArrayList` class is safer. It is impossible to add a `String` object into an `ArrayList<BankAccount>`, but you can accidentally add a `String` into a `LinkedList` that is intended to hold bank accounts.

```
ArrayList<BankAccount> accounts1 = new ArrayList<BankAccount>();
LinkedList accounts2 = new LinkedList(); // Should hold BankAccount objects
accounts1.add("my savings"); // Compile-time error
accounts2.add("my savings"); // Not detected at compile time
```

The latter will result in a class cast exception when some other part of the code retrieves the string, believing it to be a bank account:

```
BankAccount account = (BankAccount) accounts2.getFirst(); // Run-time error
```

Type parameters make generic code safer and easier to read.

Code that uses the generic `ArrayList` class is also easier to read. When you spot an `ArrayList<BankAccount>`, you know right away that it must contain bank accounts. When you see a `LinkedList`, you have to study the code to find out what it contains.

In Chapters 15 and 16, we used inheritance to implement generic linked lists, hash tables, and binary trees, because you were already familiar with the concept of inheritance. Using type parameters requires new syntax and additional techniques—those are the topic of this chapter.

1. The standard library provides a class `HashMap<K, V>` with key type `K` and value type `V`. Declare a hash map that maps strings to integers.
2. The binary search tree class in Chapter 16 is an example of generic programming because you can use it with any classes that implement the `Comparable` interface. Does it achieve genericity through inheritance or type parameters?

17.2 Implementing Generic Types

In this section, you will learn how to implement your own generic classes. We will write a very simple generic class that stores *pairs* of objects, each of which can have an arbitrary type. For example,

```
Pair<String, Integer> result = new Pair<String, Integer>("Harry Morgan", 1729);
```

The `getFirst` and `getSecond` methods retrieve the first and second values of the pair.

```
String name = result.getFirst();
Integer number = result.getSecond();
```

This class can be useful when you implement a method that computes two values at the same time. A method cannot simultaneously return a `String` and an `Integer`, but it can return a single object of type `Pair<String, Integer>`.

The generic `Pair` class requires two type parameters, one for the type of the first element and one for the type of the second element.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

We need to choose variables for the type parameters. It is considered good form to use short uppercase names for type variables, such as those in the following table:

Type Variable	Meaning
E	Element type in a collection
K	Key type in a map
V	Value type in a map
T	General type
S, U	Additional general types

Type variables of a generic class follow the class name and are enclosed in angle brackets.

You place the type variables for a generic class after the class name, enclosed in angle brackets (< and >):

```
public class Pair<T, S>
```

When you declare the instance variables and methods of the `Pair` class, use the variable `T` for the first element type and `S` for the second element type:

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

   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public T getFirst() { return first; }
   public S getSecond() { return second; }
}
```

Use type parameters for the types of generic instance variables, method parameters, and return values.

Some people find it simpler to start out with a regular class, choosing some actual types instead of the type parameters. For example,

```
public class Pair // Here we start out with a pair of String and Integer values
{
   private String first;
   private Integer second;

   public Pair(String firstElement, Integer secondElement)
   {
      first = firstElement;
      second = secondElement;
   }
   public String getFirst() { return first; }
   public Integer getSecond() { return second; }
}
```

Now it is an easy matter to replace all `String` types with the type variable `S` and all `Integer` types with the type variable `T`.

This completes the declaration of the generic `Pair` class. It is ready to use whenever you need to form a pair of two objects of arbitrary types. The following sample program shows how to make use of a `Pair` for returning two values from a method.

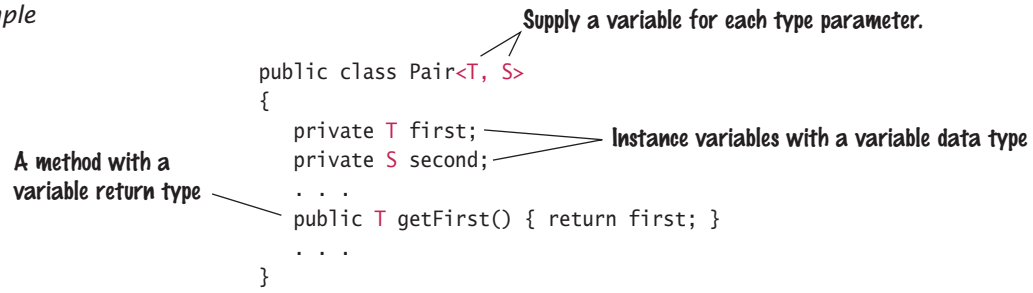
Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Syntax 17.1 Declaring a Generic Class

Syntax

```
accessSpecifier class GenericClassName<TypeVariable1, TypeVariable2, . . .>
{
   instance variables
   constructors
   methods
}
```

Example



ch17/pair/Pair.java

```
1  /**
2     This class collects a pair of elements of different types.
3  */
4  public class Pair<T, S>
5  {
6     private T first;
7     private S second;
8
9     /**
10       Constructs a pair containing two given elements.
11       @param firstElement the first element
12       @param secondElement the second element
13    */
14    public Pair(T firstElement, S secondElement)
15    {
16       first = firstElement;
17       second = secondElement;
18    }
19
20    /**
21       Gets the first element of this pair.
22       @return the first element
23    */
24    public T getFirst() { return first; }
25
26    /**
27       Gets the second element of this pair.
28       @return the second element
29    */
30    public S getSecond() { return second; }
31
32    public String toString() { return "(" + first + ", " + second + ")"; }
33 }
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

ch17/pair/PairDemo.java

```
1  public class PairDemo
2  {
3     public static void main(String[] args)
4     {
5        String[] names = { "Tom", "Diana", "Harry" };
6        Pair<String, Integer> result = firstContaining(names, "a");
7        System.out.println(result.getFirst());
8        System.out.println("Expected: Diana");
9        System.out.println(result.getSecond());
10       System.out.println("Expected: 1");
11    }
12
13    /**
14       Gets the first String containing a given string, together
15       with its index.
16       @param strings an array of strings
17       @param sub a string
18       @return a pair (strings[i], i) where strings[i] is the first
19       strings[i] containing str, or a pair (null, -1) if there is no
20       match.
21    */
22    public static Pair<String, Integer> firstContaining(
23       String[] strings, String sub)
24    {
25       for (int i = 0; i < strings.length; i++)
26       {
27          if (strings[i].contains(sub))
28          {
29             return new Pair<String, Integer>(strings[i], i);
30          }
31       }
32       return new Pair<String, Integer>(null, -1);
33    }
34 }
```

Program Run

```
Diana
Expected: Diana
1
Expected: 1
```

3. How would you use the generic `Pair` class to construct a pair of strings `"Hello"` and `"World"`?
4. What is the difference between an `ArrayList<Pair<String, Integer>>` and a `Pair<ArrayList<String>, Integer>`?

17.3 Generic Methods

A generic method is a method with a type parameter.

A generic method is a method with a type parameter. Such a method can occur in a class that in itself is not generic. You can think of it as a template for a set of methods that differ only by one or more types. For example, we may want to declare a method that can print an array of any type:

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
public class ArrayUtil
{
   /**
      Prints all elements in an array.
      @param a the array to print
   */
   public <T> static void print(T[] a)
   {
      . . .
   }
   . . .
}
```

As described in the previous section, it is often easier to see how to implement a generic method by starting with a concrete example. This method prints the elements in an array of *strings*.

```
public class ArrayUtil
{
   public static void print(String[] a)
   {
      for (String e : a)
         System.out.print(e + " ");
      System.out.println();
   }
   . . .
}
```

Supply the type parameters of a generic method between the modifiers and the method return type.

In order to make the method into a generic method, replace `String` with a type parameter, say `E`, to denote the element type of the array. Add a type parameter list, enclosed in angle brackets, between the modifiers (`public static`) and the return type (`void`):

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Syntax 17.2 Declaring a Generic Method

Syntax

modifiers <*TypeVariable*$_1$, *TypeVariable*$_2$, . . .> *returnType methodName*(*parameters*)
{
   body
}

Example

```
public static <E> void print(E[] a)
{
   for (E e : a)
      System.out.print(e + " ");
   System.out.println();
}
```

Supply the type variable before the return type.

Local variable with a variable data type

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

When you call the generic method, you need not specify which type to use for the type parameter. (In this regard, generic methods differ from generic classes.) Simply call the method with appropriate parameters, and the compiler will match up the type parameters with the parameter types. For example, consider this method call:

```
Rectangle[] rectangles = . . .;
ArrayUtil.print(rectangles);
```

When calling a generic method, you need not instantiate the type parameters.

The type of the `rectangles` parameter is `Rectangle[]`, and the type of the parameter variable is `E[]`. The compiler deduces that `E` is `Rectangle`.

This particular generic method is a static method in an ordinary class. You can also declare generic methods that are not static. You can even have generic methods in generic classes.

As with generic classes, you cannot replace type parameters with primitive types. The generic `print` method can print arrays of any type *except* the eight primitive types. For example, you cannot use the generic `print` method to print an array of type `int[]`. That is not a major problem. Simply implement a `print(int[] a)` method in addition to the generic `print` method.

5. Exactly what does the generic `print` method print when you pass an array of `BankAccount` objects containing two bank accounts with zero balances?
6. Is the `getFirst` method of the `Pair` class a generic method?

17.4 Constraining Type Parameters

Type parameters can be constrained with bounds.

It is often necessary to specify what types can be used in a generic class or method. Consider a generic `min` method that finds the smallest element in an array list of objects. How can you find the smallest element when you know nothing about the element type? You need to have a mechanism for comparing array elements. One solution is to require that the elements belong to a type that implements the `Comparable` interface. In this situation, we need to *constrain* the type parameter.

```
public static <E extends Comparable> E min(E[] a)
{
   E smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].compareTo(smallest) < 0) smallest = a[i];
   return smallest;
}
```

You can call `min` with a `String[]` array but not with a `Rectangle[]` array—the `String` class implements `Comparable`, but `Rectangle` does not.

The `Comparable` bound is necessary for calling the `compareTo` method. Had it been omitted, then the `min` method would not have compiled. It would have been illegal to call `compareTo` on `a[i]` if nothing is known about its type. (Actually, the `Comparable` interface is itself a generic type, but for simplicity we do not supply a type parameter. See Special Topic 17.1 on page 731 for more information.)

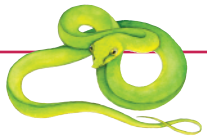
Very occasionally, you need to supply two or more type bounds. Then you separate them with the `&` character, for example

```
<E extends Comparable & Cloneable>
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

The `extends` reserved word, when applied to type parameters, actually means "extends or implements". The bounds can be either classes or interfaces, and the type parameter can be replaced with a class or interface type.

7. How would you constrain the type parameter for a generic `BinarySearchTree` class?
8. Modify the `min` method to compute the minimum of an array of elements that implements the `Measurable` interface of Chapter 9.



Common Error 17.1

Genericity and Inheritance

If `SavingsAccount` is a subclass of `BankAccount`, is `ArrayList<SavingsAccount>` a subclass of `ArrayList<BankAccount>`? Perhaps surprisingly, it is not. Inheritance of type parameters does not lead to inheritance of generic classes. There is no relationship between `ArrayList<SavingsAccount>` and `ArrayList<BankAccount>`.

This restriction is necessary for type checking. Without the restriction, it would be possible to add objects of unrelated types to a collection. Suppose it was possible to assign an `ArrayList<SavingsAccount>` object to a variable of type `ArrayList<BankAccount>`:

```
ArrayList<SavingsAccount> savingsAccounts = new ArrayList<SavingsAccount>();
ArrayList<BankAccount> bankAccounts = savingsAccounts;
   // Not legal, but suppose it was
BankAccount harrysChecking = new CheckingAccount();
   // CheckingAccount is another subclass of BankAccount
bankAccounts.add(harrysChecking); // OK—can add BankAccount object
```

But `bankAccounts` and `savingsAccounts` refer to the same array list! If the assignment was legal, we would be able to add a `CheckingAccount` into an `ArrayList<SavingsAccount>`.

In many situations, this limitation can be overcome by using wildcards—see Special Topic 17.1.

Special Topic 17.1

Wildcard Types

It is often necessary to formulate subtle constraints of type parameters. Wildcard types were invented for this purpose. There are three kinds of wildcard types:

Name	Syntax	Meaning
Wildcard with lower bound	`? extends B`	Any subtype of `B`
Wildcard with upper bound	`? super B`	Any supertype of `B`
Unbounded wildcard	`?`	Any type

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

A wildcard type is a type that can remain unknown. For example, we can declare the following method in the `LinkedList<E>` class:

```
public void addAll(LinkedList<? extends E> other)
{
   ListIterator<E> iter = other.listIterator();
   while (iter.hasNext()) add(iter.next());
}
```

The method adds all elements of `other` to the end of the linked list.

The `addAll` method doesn't require a specific type for the element type of `other`. Instead, it allows you to use any type that is a subtype of `E`. For example, you can use `addAll` to add a `LinkedList<SavingsAccount>` to a `LinkedList<BankAccount>`.

To see a wildcard with a `super` bound, have another look at the `min` method of the preceding section. Recall that `Comparable` is a generic interface; the type parameter of the `Comparable` interface specifies the parameter type of the `compareTo` method.

```
public interface Comparable<T>
{
   int compareTo(T other)
}
```

Therefore, we might want to specify a type bound:

```
public static <E extends Comparable<E>> E min(E[] a)
```

However, this bound is too restrictive. Suppose the `BankAccount` class implements `Comparable<BankAccount>`. Then the subclass `SavingsAccount` also implements `Comparable<BankAccount>` and *not* `Comparable<SavingsAccount>`. If you want to use the `min` method with a `SavingsAccount` array, then the type parameter of the `Comparable` interface should be *any supertype* of the array element type:

```
public static <E extends Comparable<? super E>> E min(E[] a)
```

Here is an example of an unbounded wildcard. The `Collections` class declares a method

```
public static void reverse(List<?> list)
```

You can think of that declaration as a shorthand for

```
public static <T> void reverse(List<T> list)
```

17.5 Type Erasure

The virtual machine erases type parameters, replacing them with their bounds or `Object`s.

Because generic types are a fairly recent addition to the Java language, the virtual machine that executes Java programs does not work with generic classes or methods. Instead, type parameters are "erased", that is, they are replaced with ordinary Java types. Each type parameter is replaced with its bound, or with `Object` if it is not bounded.

For example, the generic class `Pair<T, S>` turns into the following raw class:

```
public class Pair
{
   private Object first;
   private Object second;

   public Pair(Object firstElement, Object secondElement)
   {
      first = firstElement;
      second = secondElement;
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

```
   }
   public Object getFirst() { return first; }
   public Object getSecond() { return second; }
}
```

As you can see, the type parameters `T` and `S` have been replaced by `Object`. The result is an ordinary class.

The same process is applied to generic methods. After erasing the type parameter, the `min` method of the preceding section turns into an ordinary method. Note that in this example, the type parameter is replaced with its bound, the `Comparable` interface:

```
public static Comparable min(Comparable[] a)
{
   Comparable smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].compareTo(smallest) < 0) smallest = a[i];
   return smallest;
}
```

You cannot construct objects or arrays of a generic type.

Knowing about type erasure helps you understand limitations of Java generics. For example, you cannot construct new objects of a generic type. The following method, which tries to fill an array with copies of default objects, would be wrong:

```
public static <E> void fillWithDefaults(E[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new E(); // ERROR
}
```

To see why this is a problem, carry out the type erasure process, as if you were the compiler:

```
public static void fillWithDefaults(Object[] a)
{
   for (int i = 0; i < a.length; i++)
      a[i] = new Object(); // Not useful
}
```

Of course, if you start out with a `Rectangle[]` array, you don't want it to be filled with `Object` instances. But that's what the code would do after erasing types.

In situations such as this one, the compiler will report an error. You then need to come up with another mechanism for solving your problem. In this particular example, you can supply a default object:

```
public static <E> void fillWithDefaults(E[] a, E defaultValue)
{
   for (int i = 0; i < a.length; i++)
      a[i] = defaultValue;
}
```

Similarly, you cannot construct an array of a generic type.

```
public class Stack<E>
{
   private E[] elements;
   . . .
   public Stack()
   {
      elements = new E[MAX_SIZE]; // Error
   }
}
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Because the array construction expression `new E[]` would be erased to `new Object[]`, the compiler disallows it. A remedy is to use an array list instead:

```
public class Stack<E>
{
   private ArrayList<E> elements;
   . . .
   public Stack()
   {
      elements = new ArrayList<E>(); // Ok
   }
   . . .
}
```

Another solution is to use an array of objects and provide a cast when reading elements from the array:

```
public class Stack<E>
{
   private Object[] elements;
   private int size;
   . . .
   public Stack()
   {
      elements = new Object[MAX_SIZE]; // Ok
   }
   . . .
   public E pop()
   {
      size--;
      return (E) elements[size];
   }
}
```

The cast `(E)` generates a warning because it cannot be checked at run time.

These limitations are frankly awkward. It is hoped that a future version of Java will no longer erase types so that the current restrictions that are the consequence of erasure can be lifted.

SELF CHECK

9. What is the erasure of the `print` method in Section 17.3?

10. Could the `Stack` example be implemented as follows?

```
public class Stack<E>
{
   private E[] elements;
   . . .
   public Stack()
   {
      elements = (E[]) new Object[MAX_SIZE];
   }
   . . .
}
```

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Common Error 17.2

Using Generic Types in a Static Context

You cannot use type parameters to declare static variables, static methods, or static inner classes. For example, the following would be illegal:

```
public class LinkedList<E>
{
   private static E defaultValue; // ERROR
   . . .
   public static List<E> replicate(E value, int n) { . . . } // ERROR
   private static class Node { public E data; public Node next; } // ERROR
}
```

In the case of static variables, this restriction is very sensible. After the generic types are erased, there is only a single variable `LinkedList.defaultValue`, whereas the static variable declaration gives the false impression that there is a separate variable for each `LinkedList<E>`.

For static methods and inner classes, there is an easy workaround; simply add a type parameter:

```
public class LinkedList<E>
{
   . . .
   public static <T> List<T> replicate(T value, int n) { . . . } // OK
   private static class Node<T> { public T data; public Node<T> next; } // OK
}
```

Summary of Learning Objectives

Describe generic classes and type parameters.

- In Java, generic programming can be achieved with inheritance or with type parameters.
- A generic class has one or more type parameters.
- Type parameters can be instantiated with class or interface types.
- Type parameters make generic code safer and easier to read.

Implement generic classes and interfaces.

- Type variables of a generic class follow the class name and are enclosed in angle brackets.
- Use type parameters for the types of generic instance variables, method parameters, and return values.

Implement generic methods.

- A generic method is a method with a type parameter.
- Supply the type parameters of a generic method between the modifiers and the method return type.
- When calling a generic method, you need not instantiate the type parameters.

Specify constraints on type parameters.

- Type parameters can be constrained with bounds.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Recognize how erasure of type parameters places limitations on generic programming in Java.

- The virtual machine erases type parameters, replacing them with their bounds or `Objects`.
- You cannot construct objects or arrays of a generic type.

Media Resources

www.wiley.com/college/horstmann

- Lab Exercises
- Practice Quiz
- Code Completion Exercises

Review Exercises

★ **R17.1** What is a type parameter?

★ **R17.2** What is the difference between a generic class and an ordinary class?

★ **R17.3** What is the difference between a generic class and a generic method?

★ **R17.4** Find an example of a non-static generic method in the standard Java library.

★★ **R17.5** Find four examples of a generic class with two type parameters in the standard Java library.

★★ **R17.6** Find an example of a generic class in the standard library that is not a collection class.

★ **R17.7** Why is a bound required for the type parameter `T` in the following method?

```
<T extends Comparable> int binarySearch(T[] a, T key)
```

★★ **R17.8** Why is a bound not required for the type parameter `E` in the `HashSet<E>` class?

★ **R17.9** What is an `ArrayList<Pair<T, T>>`?

★★ **R17.10** Explain the type bounds of the following method of the `Collections` class:

```
public static <T extends Comparable<? super T>> void sort(List<T> a)
```

Why doesn't `T extends Comparable` or `T extends Comparable<T>` suffice?

★ **R17.11** What happens when you pass an `ArrayList<String>` to a method with parameter `ArrayList`? Try it out and explain.

★★★ **R17.12** What happens when you pass an `ArrayList<String>` to a method with parameter `ArrayList`, and the method stores an object of type `BankAccount` into the array list? Try it out and explain.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

★★ **R17.13** What is the result of the following test?

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
if (accounts instanceof ArrayList<String>) . . .
```

Try it out and explain.

★★ **R17.14** The `ArrayList<E>` class in the standard Java library must manage an array of objects of type `E`, yet it is not legal to construct a generic array of type `E[]` in Java. Locate the implementation of the `ArrayList` class in the library source code that is a part of the JDK. Explain how this problem is overcome.

Programming Exercises

★ **P17.1** Modify the generic `Pair` class so that both values have the same type.

★ **P17.2** Add a method `swap` to the `Pair` class of Exercise P17.1 that swaps the first and second elements of the pair.

★★ **P17.3** Implement a static generic method `PairUtil.swap` whose parameter is a `Pair` object, using the generic class declared in Section 17.2. The method should return a new pair, with the first and second element swapped.

★★ **P17.4** Write a static generic method `PairUtil.minmax` that computes the minimum and maximum elements of an array of type `T` and returns a pair containing the minimum and maximum value. Require that the array elements implement the `Measurable` interface of Chapter 9.

★★ **P17.5** Repeat the problem of Exercise P17.4, but require that the array elements implement the `Comparable` interface.

★★★ **P17.6** Repeat the problem of Exercise P17.5, but refine the bound of the type parameter to extend the generic `Comparable` type.

★★ **P17.7** Implement a generic version of the binary search algorithm.

★★★ **P17.8** Implement a generic version of the merge sort algorithm. Your program should compile without warnings.

P17.9 Implement a generic version of the `LinkedList` class of Chapter 15.

★★ **P17.10** Implement a generic version of the `BinarySearchTree` class of Chapter 16.

★★ **P17.11** Turn the `HashSet` implementation of Chapter 16 into a generic class. Use an array list instead of an array to store the buckets.

★★ **P17.12** Provide suitable `hashCode` and `equals` methods for the `Pair` class of Section 17.2 and implement a `HashMap` class, using a `HashSet<Pair<K, V>>`.

★★★ **P17.13** Implement a generic version of the permutation generator in Section 13.2. Generate all permutations of a `List<E>`.

★★ **P17.14** Write a generic static method `print` that prints the elements of any object that implements the `Iterable<E>` interface. The elements should be separated by commas. Place your method into an appropriate utility class.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.

Programming Projects

Project 17.1 Design and implement a generic version of the `DataSet` class of Chapter 9 that can be used to analyze data of any class that implements the `Measurable` interface. Make the `Measurable` interface generic as well. Supply an `addAll` method that lets you add all values from another data set with a compatible type. Supply a generic `Measurer<T>` interface to allow the analysis of data whose classes don't implement the `Measurable` type.

Project 17.2 Turn the `MinHeap` class of Chapter 16 into a generic class. As with the `TreeSet` class of the standard library, allow a `Comparator` to compare queue elements. If no comparator is supplied, assume that the element type implements the `Comparable` interface.

Answers to Self-Check Questions

1. `HashMap<String, Integer>`
2. It uses inheritance.
3. `new Pair<String, String>("Hello", "World")`
4. An `ArrayList<Pair<String, Integer>>` contains multiple pairs, for example `[(Tom, 1), (Harry, 3)]`. A `Pair<ArrayList<String>, Integer>` contains a list of strings and a single integer, such as `([Tom, Harry], 1)`.
5. The output depends on the implementation of the `toString` method in the `BankAccount` class.
6. No—the method has no type parameters. It is an ordinary method in a generic class.
7. `public class BinarySearchTree<E extends Comparable>`
8.
```
public static <E extends Measurable> E min(E[] a)
{
   E smallest = a[0];
   for (int i = 1; i < a.length; i++)
      if (a[i].getMeasure() < smallest.getMeasure())
         smallest = a[i];
   return smallest;
}
```
9.
```
public static void print(Object[] a)
{
   for (Object e : a)
      System.out.print(e + " ");
   System.out.println();
}
```
10. This code compiles (with a warning), but it is a poor technique. In the future, if type erasure no longer happens, the code will be *wrong*. The cast from `Object[]` to `String[]` will cause a class cast exception.

Big Java, 4th Edition, Cay Horstmann, Copyright © 2010 John Wiley and Sons, Inc. All rights reserved.