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
- To understand the relationship between generic types and inheritance
- To learn how to constrain type variables

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 produce array lists of arbitrary types. In this chapter, you will learn how to implement your own generic classes.

Chapter Contents

17.1 Type Variables

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 engaged in generic programming. As a result, you can form array lists that collect 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`. However, that `LinkedList` class achieves genericity with a different mechanism. It is a single `LinkedList` class that stores values of type `Object`. You can, if you like, store a `String` and a `BankAccount` object into the same `LinkedList`.

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

Our `LinkedList` class implements genericity by using *inheritance*. It stores objects of any class that inherits from `Object`. In contrast, the `ArrayList` class uses *type variables* to achieve genericity—you need to specify the type of the objects that you want to store.

Note that only our `LinkedList` class of Chapter 15 uses inheritance. The standard Java library has a `LinkedList` class that uses type variables. In the next section, we will add type variables to our `LinkedList` class as well.

A generic class has one or more type variables.

The `ArrayList` class is a *generic class:* it has been declared with a *type variable* `E`. The type variable denotes the element type:

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

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

Type variables can be instantiated with class or interface types.

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

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

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

The type that you supply replaces the type variable in the interface 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 our `LinkedList` class:

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

The `ArrayList` methods are safer. It is impossible to add a `String` object into an `ArrayList<BankAccount>`, but you can 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 give you grief 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 variables 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 variables requires new syntax and additional techniques—those are the topic of this chapter.

SYNTAX 17.1 Instantiating a Generic Class

GenericClassName<*Type*$_1$, *Type*$_2$, . . .>

Example:

```
ArrayList<BankAccount>
HashMap<String, Integer>
```

Purpose:

To supply specific types for the type variables of a generic class

SELF CHECK

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 variables?

17.2 Implementing Generic Classes

In this section, you will learn how to implement your own generic classes. We will first start out with a very simple generic class that stores pairs of objects. Then we will turn the `LinkedList` class of Chapter 15 into a generic class.

Our first example for writing a generic class stores *pairs* of objects, each of which can have an arbitrary type. For example,

```
Pair<String, BankAccount> result
      = new Pair<String, BankAccount>("Harry Hacker", harrysChecking);
```

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

```
String name = result.getFirst();
BankAccount account = 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 a `BankAccount`, but it can return a single object of type `Pair<String, BankAccount>`.

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

We need to give names to the type variables. It is considered good form to give short uppercase names for type variables, such as the following:

Type Variable Name	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 define the fields and methods of the class, use the type variable `T` for the first element type and `S` for the second element type:

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

   private T first;
   private S second;
}
```

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

This completes the definition of the generic `Pair` class. It is now ready to use whenever you need to form a pair of two objects of arbitrary types.

As a second example, let us turn our linked list class into a generic class. This class only requires one type variable for the element type, which we will call `E`.

```
public class LinkedList<E>
```

In the case of the linked list, there is a slight complication. Unlike the `Pair` class, the `LinkedList` class does not store the elements in its instance fields. Instead, a linked list manages a sequence of nodes, and the nodes store the data. Our `LinkedList` class uses an inner class `Node` for the nodes. The `Node` class must be modified to express the fact that each node stores an element of type `E`.

```
public class LinkedList<E>
{
   . . .
   private Node first;

   private class Node
   {
      public E data;
      public Node next;
   }
}
```

The implementation of some of the methods requires local variables whose type is variable, for example:

```
public E removeFirst()
{
   if (first == null)
      throw new NoSuchElementException();
   E element = first.data;
   first = first.next;
   return element;
}
```

Overall, the process is straightforward. Use the type `E` whenever you receive, return, or store an element object. Complexities arise only when your data structure uses helper classes, such as the nodes and iterators in a linked list. If the helpers are inner classes, you need not do anything special. However, helper types that are defined *outside* the generic class need to become generic classes as well.

Following is the complete reimplementation of our `LinkedList` class, as a generic class with a type variable.

SYNTAX 17.2 Defining a Generic Class

accessSpecifier `class` *GenericClassName*<*TypeVariable*$_1$, *TypeVariable*$_2$, . . .>
{
    constructors
    methods
    fields
}

Example:

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

Purpose:

To define a generic class with methods and fields that depend on type variables

ch17/genlist/LinkedList.java

```
import java.util.NoSuchElementException;

/**
   A linked list is a sequence of nodes with efficient
   element insertion and removal. This class
   contains a subset of the methods of the standard
   java.util.LinkedList class.
*/
public class LinkedList<E>
{
   /**
      Constructs an empty linked list.
   */
   public LinkedList()
   {
      first = null;
   }

   /**
      Returns the first element in the linked list.
      @return the first element in the linked list
   */
   public E getFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      return first.data;
   }

   /**
      Removes the first element in the linked list.
      @return the removed element
   */
   public E removeFirst()
   {
      if (first == null)
         throw new NoSuchElementException();
      E element = first.data;
      first = first.next;
      return element;
   }

   /**
      Adds an element to the front of the linked list.
      @param element the element to add
   */
   public void addFirst(E element)
   {
      Node newNode = new Node();
      newNode.data = element;
      newNode.next = first;
      first = newNode;
   }

   /**
      Returns an iterator for iterating through this list.
      @return an iterator for iterating through this list
   */
   public ListIterator<E> listIterator()
   {
      return new LinkedListIterator();
   }

   private Node first;

   private class Node
   {
      public E data;
      public Node next;
   }

   private class LinkedListIterator implements ListIterator<E>
   {
      /**
         Constructs an iterator that points to the front
         of the linked list.
      */
      public LinkedListIterator()
      {
         position = null;
         previous = null;
      }

      /**
         Moves the iterator past the next element.
         @return the traversed element
      */
      public E next()
      {
         if (!hasNext())
            throw new NoSuchElementException();
         previous = position; // Remember for remove

         if (position == null)
            position = first;
         else
            position = position.next;

         return position.data;
      }

      /**
         Tests if there is an element after the iterator
         position.
         @return true if there is an element after the iterator
         position
      */
      public boolean hasNext()
      {
         if (position == null)
            return first != null;
         else
            return position.next != null;
      }

      /**
         Adds an element before the iterator position
         and moves the iterator past the inserted element.
         @param element the element to add
      */
      public void add(E element)
      {
         if (position == null)
         {
            addFirst(element);
            position = first;
         }
         else
         {
            Node newNode = new Node();
            newNode.data = element;
            newNode.next = position.next;
            position.next = newNode;
            position = newNode;
         }
         previous = position;
      }

      /**
         Removes the last traversed element. This method may
         only be called after a call to the next() method.
      */
      public void remove()
      {
         if (previous == position)
            throw new IllegalStateException();

         if (position == first)
         {
            removeFirst();
         }
         else
         {
            previous.next = position.next;
         }
         position = previous;
      }

      /**
         Sets the last traversed element to a different
         value.
         @param element the element to set
      */
      public void set(E element)
      {
         if (position == null)
            throw new NoSuchElementException();
         position.data = element;
      }

      private Node position;
      private Node previous;
   }
}
```

ch17/genlist/ListIterator.java

```java
/**
   A list iterator allows access to a position in a linked list.
   This interface contains a subset of the methods of the
   standard java.util.ListIterator interface. The methods for
   backward traversal are not included.
*/
public interface ListIterator<E>
{
   /**
      Moves the iterator past the next element.
      @return the traversed element
   */
   E next();

   /**
      Tests if there is an element after the iterator
      position.
      @return true if there is an element after the iterator
      position
   */
   boolean hasNext();

   /**
      Adds an element before the iterator position
      and moves the iterator past the inserted element.
      @param element the element to add
   */
   void add(E element);

   /**
      Removes the last traversed element. This method may
      only be called after a call to the next method.
   */
   void remove();

   /**
      Sets the last traversed element to a different
      value.
      @param element the element to set
   */
   void set(E element);
}
```

ch17/genlist/ListTester.java

```java
/**
   A program that tests the LinkedList class.
*/
public class ListTester
{
    public static void main(String[] args)
    {
       LinkedList<String> staff = new LinkedList<String>();
       staff.addFirst("Tom");
       staff.addFirst("Romeo");
       staff.addFirst("Harry");
       staff.addFirst("Dick");

       // | in the comments indicates the iterator position

       ListIterator<String> iterator = staff.listIterator(); // |DHRT
       iterator.next(); // D|HRT
       iterator.next(); // DH|RT

       // Add more elements after second element

       iterator.add("Juliet"); // DHJ|RT
       iterator.add("Nina"); // DHJN|RT

       iterator.next(); // DHJNR|T

       // Remove last traversed element

       iterator.remove(); // DHJN|T

       // Print all elements

       iterator = staff.listIterator();
       while (iterator.hasNext())
       {
          String element = iterator.next();
          System.out.print(element + " ");
       }
       System.out.println();
       System.out.println("Expected: Dick Harry Juliet Nina Tom");
    }
 }
```

Output

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
```

Self Check

3. How would you use the generic `Pair` class to construct a pair of strings `"Hello"` and `"World"`?
4. What change was made to the `ListIterator` interface, and why was that change necessary?

17.3 Generic Methods

Generic methods can be defined inside ordinary and generic classes.

A generic method is a method with a type variable. You can think of it as a template for a set of methods that differ only by one or more types. One way of defining a generic method is by starting with a method that operates on a specific type. As an example, consider the following `print` method:

```
public class ArrayUtil
{
   /**
      Prints all elements in an array of strings.
      @param a  the array to print
   */
   public static void print(String[] a)
   {
      for (String e : a)
         System.out.print(e + " ");
      System.out.println();
   }
   . . .
}
```

This method prints the elements in an array of *strings*. However, we may want to print an array of `Rectangle` objects instead. Of course, the same algorithm works for an array of any type.

Supply the type variables 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 variable, say `E`, to denote the element type of the array. Add a type variable 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();
}
```

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

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

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

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 define generic methods that are not static. You can even have generic methods in generic classes.

SYNTAX 17.3 Defining a Generic Method

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

Example:

```
public static <E> void print(E[] a)
{
   . . .
}
```

Purpose:

To define a generic method that depends on type variables

As with generic classes, you cannot replace type variables 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.

SELF CHECK

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 Variables

Type variables 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 variable.

```
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 Advanced Topic 17.1 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>
```

The extends keyword, when applied to type variables, actually means "extends or implements". The bounds can be either classes or interfaces, and the type variable can be replaced with a class or interface type.

Self Check

7. How would you constrain the type variable 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. 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();
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 Advanced Topic 17.1.

Advanced Topic 17.1

Wildcard Types

It is often necessary to formulate subtle constraints of type variables. 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

A wildcard type is a type that can remain unknown. For example, we can define 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 defines 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 Raw Types

The virtual machine works with raw types, not with generic classes.

The raw type of a generic type is obtained by erasing the type variables.

The virtual machine that executes Java programs does not work with generic classes or methods. Instead, it uses *raw* types, in which the type variables are replaced with ordinary Java types. Each type variable is replaced with its bound, or with `Object` if it is not bounded.

The compiler *erases* the type variables when it compiles generic classes and methods. For example, the generic class `Pair<T, S>` turns into the following raw class:

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

   private Object first;
   private Object second;
}
```

As you can see, the type variables `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:

```
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;
}
```

Knowing about raw types helps you understand limitations of Java generics. For example, you cannot replace type variables with primitive types. Erasure turns type variables into the bounds type, such as `Object` or `Comparable`. The resulting types can never hold values of primitive types.

To interface with legacy code, you can convert between generic and raw types.

Raw types are necessary when you interface with *legacy code* that was written before generics were added to the Java language. For example, if a legacy method has a parameter `ArrayList` (without a type variable), you can pass an `ArrayList<String>` or `ArrayList<BankAccount>`. This is not completely safe—after all, the legacy method might insert an object of the wrong type. The compiler will issue a warning, but your program will compile and run.

Self Check

9. What is the erasure of the `print` method in Section 17.3?
10. What is the raw type of the `LinkedList<E>` class in Section 17.2?

Common Error 17.2

Writing Code That Does Not Work After Types Are Erased

Generic classes and methods were added to Java several years after the language became successful. The language designers decided to use the type erasure mechanism because it makes it easy to interface generic code with legacy programs. As a result, you may run into some programming restrictions when you write generic code.

For example, you cannot construct new objects of a generic type. For example, 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. Because an array construction expression `new E[]` would be erased to `new Object[]`, the compiler disallows it.

Common Error 17.3

Using Generic Types in a Static Context

You cannot use type variables to define static fields, 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 fields, this restriction is very sensible. After the generic types are erased, there is only a single field `LinkedList.defaultValue`, whereas the static field declaration gives the false impression that there is a separate field 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
}
```

CHAPTER SUMMARY

1. In Java, generic programming can be achieved with inheritance or with type variables.
2. A generic class has one or more type variables.
3. Type variables can be instantiated with class or interface types.
4. Type variables make generic code safer and easier to read.
5. Type variables of a generic class follow the class name and are enclosed in angle brackets.
6. Use type variables for the types of generic fields, method parameters, and return values.
7. Generic methods can be defined inside ordinary and generic classes.
8. Supply the type variables of a generic method between the modifiers and the method return type.
9. When calling a generic method, you need not instantiate the type variables.

10. Type variables can be constrained with bounds.
11. The virtual machine works with raw types, not with generic classes.
12. The raw type of a generic type is obtained by erasing the type variables.
13. To interface with legacy code, you can convert between generic and raw types.

REVIEW EXERCISES

★ **Exercise R17.1.** What is a type variable?

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

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

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

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

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

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

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

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

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

★★ **Exercise 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?

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

★★★ **Exercise 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.

★★ **Exercise 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.

★★★ **Exercise R17.14.** If a class implements the generic `Iterable` interface, then you can use its objects in the “for each” loop—see Advanced Topic 15.1. Describe the needed modifications to the `LinkedList<E>` class of Section 17.2.

Additional review exercises are available in WileyPLUS.

PROGRAMMING EXERCISES

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

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

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

★★ **Exercise 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.

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

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

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

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

★★ **Exercise P17.9.** Implement a generic version of the `BinarySearchTree` class of Chapter 16.

★★ **Exercise P17.10.** Turn the `HashSet` implementation of Chapter 16 into a generic class. Use an array list instead of an array to store the buckets.

★★ **Exercise P17.11.** Define suitable `hashCode` and `equals` methods for the `Pair` class of Section 17.2 and implement a `HashMap` class, using a `HashSet<Pair<K, V>>`.

★★★ **Exercise P17.12.** Implement a generic version of the permutation generator in Section 13.2. Generate all permutations of a `List<E>`.

★★ **Exercise P17.13.** 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.

Additional programming exercises are available in WileyPLUS.

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 `PriorityQueue` 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. `ListIterator<E>` is now a generic type. Its interface depends on the element type of the linked list.
5. The output depends on the definition 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. The `LinkedList` class of Chapter 15.