BIG JAVA

Late Objects

CAY HORSTMANN

San Jose State University

WILEY

John Wiley & Sons, Inc.

CHAPTER 18

GENERIC CLASSES

CHAPTER GOALS

To understand the objective of generic programming

To implement generic classes and methods

To explain the execution of generic methods in the virtual machine

To describe the limitations of generic programming in Java

CHAPTER CONTENTS

In the supermarket, a generic product can be sourced from multiple suppliers. In computer science, generic programming involves the design and implementation of data structures and algorithms that work for multiple types. You have already seen 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 and methods.

18.1 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 16.1 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. For example, you can add elements of type `String` because the `String` class extends `Object`. In contrast, the `ArrayList` and `LinkedList` classes from the standard Java library are *generic classes*. Each of these classes has a type parameter for specifying the type of its elements. For example, an `ArrayList<String>` stores `String` elements.

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

A generic class has one or more type parameters.

When declaring a generic class, you supply a variable for each type parameter. For example, the standard library declares the class `ArrayList<E>`, where `E` is the type variable that denotes the element type. You use the same variable in the declaration of the methods, whenever you need to refer to that type. For example, the `ArrayList<E>` class declares methods

```
public void add(E element)
public E get(int index)
```

You could use another name, such as `ElementType`, instead of `E`. However, it is customary to use short, uppercase names for type parameters.

Type parameters can be instantiated with class or interface types.

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

```
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 16:

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

ONLINE EXAMPLE

Sample programs that demonstrate safety problems when using collections without type parameters.

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.addFirst("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 16 and 17, 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.

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 17 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?
3. Does the following code contain an error? If so, is it a compile-time or run-time error?
   ```
   ArrayList<Integer> a = new ArrayList<Integer>();
   String s = a.get(0);
   ```
4. Does the following code contain an error? If so, is it a compile-time or run-time error?
   ```
   ArrayList<Double> a = new ArrayList<Double>();
   a.add(3);
   ```
5. Does the following code contain an error? If so, is it a compile-time or run-time error?
   ```
   LinkedList a = new LinkedList();
   a.addFirst("3.14");
   double x = (Double) a.removeFirst();
   ```

Practice It Now you can try these exercises at the end of the chapter: R18.4, R18.5, R18.6.

18.2 Implementing Generic Types

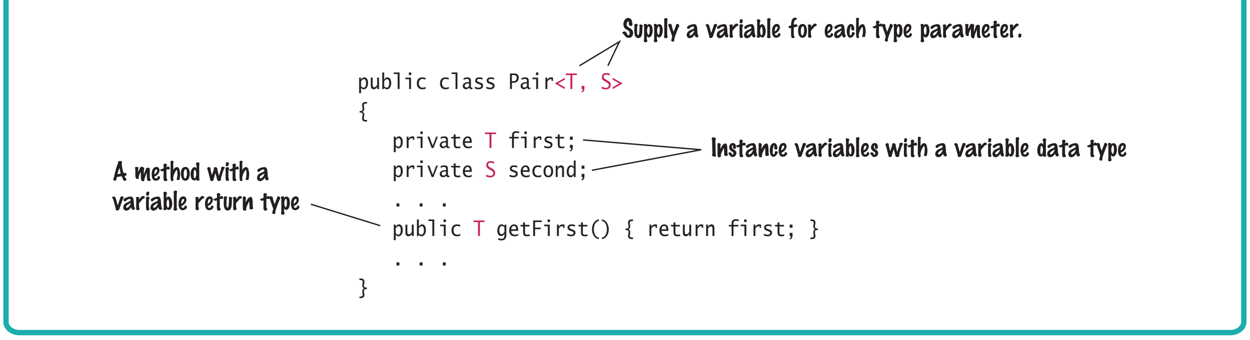
In this section, you will learn how to implement your own generic classes. We will write a very simple generic class that stores *pairs* of objects, each of which can have an arbitrary type. For example,

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

Syntax 18.1 Declaring a Generic Class

Syntax

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



```
public class Pair<T, S>
{
   private T first;
   private S second;
   . . .
   public T getFirst() { return first; }
   . . .
}
```

Supply a variable for each type parameter.
Instance variables with a variable data type
A method with a variable return type

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.

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 parameter variables, 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 `T` and all `Integer` types with the type variable `S`.

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.

section_2/Pair.java

```
/**
   This class collects a pair of elements of different types.
*/
public class Pair<T, S>
{
   private T first;
   private S second;

   /**
      Constructs a pair containing two given elements.
      @param firstElement the first element
      @param secondElement the second element
   */
   public Pair(T firstElement, S secondElement)
   {
      first = firstElement;
      second = secondElement;
   }

   /**
      Gets the first element of this pair.
      @return the first element
   */
   public T getFirst() { return first; }

   /**
      Gets the second element of this pair.
      @return the second element
   */
   public S getSecond() { return second; }

   public String toString() { return "(" + first + ", " + second + ")"; }
}
```

section_2/PairDemo.java

```
public class PairDemo
{
   public static void main(String[] args)
   {
      String[] names = { "Tom", "Diana", "Harry" };
      Pair<String, Integer> result = firstContaining(names, "a");
      System.out.println(result.getFirst());
      System.out.println("Expected: Diana");
      System.out.println(result.getSecond());
      System.out.println("Expected: 1");
   }

   /**
      Gets the first String containing a given string, together
      with its index.
      @param strings an array of strings
      @param sub a string
      @return a pair (strings[i], i) where strings[i] is the first
      strings[i] containing str, or a pair (null, -1) if there is no
      match.
   */
   public static Pair<String, Integer> firstContaining(
      String[] strings, String sub)
   {
      for (int i = 0; i < strings.length; i++)
      {
         if (strings[i].contains(sub))
         {
            return new Pair<String, Integer>(strings[i], i);
         }
      }
      return new Pair<String, Integer>(null, -1);
   }
}
```

Program Run

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

SELF CHECK

6. How would you use the generic `Pair` class to construct a pair of strings `"Hello"` and `"World"`?
7. How would you use the generic `Pair` class to construct a pair containing “`Hello`” and `1729`?
8. What is the difference between an `ArrayList<Pair<String, Integer>>` and a `Pair<ArrayList<String>, Integer>`?
9. Write a method `roots` with a `Double` parameter variable `x` that returns both the positive and negative square root of `x` if `x` ≥ `0` or `null` otherwise.
10. How would you implement a class `Triple` that collects three values of arbitrary types?

Practice It Now you can try these exercises at the end of the chapter: P18.1, P18.2, P18.9.

18.3 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:

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

Syntax 18.2 Declaring a Generic Method

Syntax *modifiers <TypeVariable$_1$, TypeVariable$_2$, . . .> returnType methodName(parameters)*

```
{
   body
}
```

Supply the type variable before the return type.

```
public static <E> String toString(ArrayList<E> a)
{
   String result = "";
   for (E e : a)
   {
      result = result + e + " ";
   }
   return result;
}
```

Local variable with a variable data type

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();
}
```

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

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 arguments, and the compiler will match up the type parameters with the argument types. For example, consider this method call:

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

ONLINE EXAMPLE

A sample program with a generic method for printing an array of objects and a non-generic method for printing an array of integers.

The type of the `rectangles` argument 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.

SELF CHECK

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

13. Consider this `fill` method:

```
public static <T> void fill(List<T> lst, T value)
{
   for (int i = 0; i < lst.size(); i++) { lst.set(i, value); }
}
```

If you have an array list

```
ArrayList<String> a = new ArrayList<String>(10);
```

how do you fill it with ten `"*"`?

14. What happens if you pass `42` instead of `"*"` to the `fill` method?

15. Consider this `fill` method:

```
public static <T> fill(T[] arr, T value)
{
   for (int i = 0; i < arr.length; i++) { arr[i] = value; }
}
```

What happens when you execute the following statements?

```
String[] a = new String[10];
fill(a, 42);
```

Practice It Now you can try these exercises at the end of the chapter: P18.3, P18.4, P18.19.

18.4 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 method that finds the average of the values in an array list of objects. How can you compute averages when you know nothing about the element type? You need to have a mechanism for measuring the elements. In Section 9.6, we designed an interface for that purpose:

You can place restrictions on the type parameters of generic classes and methods.

```
public interface Measurable
{
   double getMeasure();
}
```

We can constrain the type of the elements, requiring that the type implement the `Measurable` type. In Java, this is achieved by adding the clause `extends Measurable` after the type parameter:

```
public static <E extends Measurable> double average(ArrayList<E> objects)
```

This means, "`E` or one of its superclasses extends or implements `Measurable`". In this situation, we say that `E` is a subtype of the `Measurable` type.

Here is the complete `average` method:

```
public static <E extends Measurable> double average(ArrayList<E> objects)
{
   if (objects.size() == 0) { return 0; }
   double sum = 0;
   for (E obj : objects)
   {
         sum = sum + obj.getMeasure();
      }
      return sum / objects.size();
   }
```

Note the call obj.getMeasure(). The variable obj has type E, and E is a subtype of Measurable. Therefore, we know that it is legal to apply the getMeasure method to obj.

If the BankAccount class implements the Measurable interface, then you can call the average method with an array list of BankAccount objects. But you cannot compute the average of an array list of strings because the String class does not implement the Measurable interface.

Now consider the task of finding the minimum in an array list. We can return the element with the smallest measure (see Self Check 17). However, the Measurable interface was created for this book and is not widely used. Instead, we will use the Comparable interface type that many classes implement. The Comparable interface is itself a generic type. The type parameter specifies the type of the parameter variable of the compareTo method:

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

For example, String implements Comparable<String>. You can compare strings with other strings, but not with objects of different classes.

If the array list has elements of type E, then we want to require that E implements Comparable<E>. Here is the method:

```
public static <E extends Comparable<E>> E min(ArrayList<E> objects)
{
   E smallest = objects.get(0);
   for (int i = 1; i < objects.size(); i++)
   {
      E obj = objects.get(i);
      if (obj.compareTo(smallest) < 0)
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

Because of the type constraint, we know that obj has a method

```
int compareTo(E other)
```

Therefore, the call

```
obj.compareTo(smallest)
```

is valid.

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

```
<E extends Comparable<E> & Measurable>
```

ONLINE EXAMPLE

A sample program that demonstrates a constraint on a type parameter.

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.

SELF CHECK

16. How would you constrain the type parameter for a generic `BinarySearchTree` class?

17. Modify the `min` method to compute the minimum of an array list of elements that implements the `Measurable` interface.

18. Could we have declared the `min` method of Self Check 17 without type parameters, like this?

```
public static Measurable min(ArrayList<Measurable> a)
```

19. Could we have declared the `min` method of Self Check 17 without type parameters for arrays, like this?

```
public static Measurable min(Measurable[] a)
```

20. How would you implement the generic `average` method for arrays?

21. Is it necessary to use a generic `average` method for arrays of measurable objects?

Practice It Now you can try these exercises at the end of the chapter: P18.5, P18.7, P18.20.

Common Error 18.1

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 18.1.

Common Error 18.2

The Array Store Exception

In Common Error 18.1, you saw that one cannot assign a subclass list to a superclass list. For example, an `ArrayList<SavingsAccount>` cannot be used where an `ArrayList<BankAccount>` is expected.

This is surprising, because you *can* perform the equivalent assignment with arrays. For example,

```
SavingsAccount[] savingsAccounts = new SavingsAccount[10];
BankAccount bankAccounts = savingsAccounts; // Legal
```

But there was a reason the assignment wasn't legal for array lists—it would have allowed storing a `CheckingAccount` into `savingsAccounts`.

Let's try that with arrays:

```
BankAccount harrysChecking = new CheckingAccount();
bankAccounts[harrysChecking]; // Throws ArrayStoreException
```

This code compiles. The object `harrysChecking` is a `CheckingAccount` and hence a `BankAccount`. But `bankAccounts` and `savingsAccounts` are references to the same array—an array of type `SavingsAccount[]`. When the program runs, that array refuses to store a `CheckingAccount`, and throws an `ArrayStoreException`.

Both `ArrayList` and arrays avoid the type error, but they do it in different ways. The `ArrayList` class avoids it at compile-time, and arrays avoid it at run-time. Generally, we prefer a compile-time error notification, but the cost is steep, as you can see from Special Topic 18.1. It is a lot of work to tell the compiler precisely which conversions should be permitted.

Special Topic 18.1

Wildcard Types

It is often necessary to formulate subtle constraints on 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

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:

```
public static <E extends Comparable<E>> E min(ArrayList<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 list, then the type parameter of the `Comparable` interface should be *any supertype* of the array list's element type:

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

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

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

ONLINE EXAMPLE

A program that demonstrates the need for wildcards.

You can think of that declaration as a shorthand for

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

Common Error 18.2 compares this limitation with the seemingly more permissive behavior of arrays in Java.

18.5 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;
   }
   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. Consider this method:

```
public static <E extends Measurable> E min(E[] objects)
{
   E smallest = objects[0];
   for (int i = 1; i < objects.length; i++)
   {
      E obj = objects[i];
      if (obj.getMeasure() < smallest.getMeasure())
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

In the Java virtual machine, generic types are erased.

When erasing the type parameter, it is replaced with its bound, the `Measurable` interface:

```
public static Measurable min(Measurable[] objects)
{
   Measurable smallest = objects[0];
   for (int i = 1; i < objects.length; i++)
   {
      Measurable obj = objects[i];
      if (obj.getMeasure() < smallest.getMeasure())
      {
         smallest = obj;
      }
   }
   return smallest;
}
```

You cannot construct objects or arrays of a generic type.

Knowing about type erasure helps you understand the 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 fill(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
      }
   }
```

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 currentSize;
   . . .
   public Stack()
   {
      elements = new Object[MAX_SIZE]; // Ok
   }
   . . .
   public E pop()
   {
      size--;
      return (E) elements[currentSize];
   }
}
```

ONLINE EXAMPLE

This program shows how to implement a generic stack as an array of objects.

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 due to erasure can be lifted.

SELF CHECK

22. Suppose we want to eliminate the type bound in the `min` method of Section 18.5, by declaring the parameter variable as an array of `Comparable<E>` objects. Why doesn't this work?

23. What is the erasure of the `print` method in Section 18.3?

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

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

25. The `ArrayList<E>` class has a method

```
Object[] toArray()
```

Why doesn't the method return an `E[]`?

26. The `ArrayList<E>` class has a second method

```
E[] toArray(E[] a)
```

Why can this method return an array of type `E[]`? (*Hint:* Special Topic 18.2.)

27. Why can't the method

```
static <T> T[] copyOf(T[] original, int newLength)
```

be implemented without reflection?

Practice It Now you can try these exercises at the end of the chapter: R18.11, R18.14, P18.22.

Common Error 18.3

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
}
```

Special Topic 18.2

Reflection

As you have seen, type erasure makes it impossible for a generic method to construct a generic array. There is an advanced technique called *reflection* that you can sometimes use to overcome this limitation. Reflection lets you work with classes in a running program.

In Java, the virtual machine keeps a `Class` object for each class that has been loaded. That object has information about each class, as well as methods to construct new objects of the class.

Given an object, you can get its class object by calling `getClass`:

```
Class objsClass = obj.getClass();
```

You can then make a new instance of that class by calling the `newInstance` method:

```
Object newObj = objsClass.newInstance();
```

This method throws an exception if it cannot access a constructor with no arguments.

Given an array, you can get the type of the elements this way:

```
Class arrayClass = array.getClass();
Class elementClass = arrayClass.getComponentType();
```

If you want to create a new array, use the `Array.newInstance` method:

```
Object[] newArray = Array.newInstance(elementClass, length);
```

Using these methods, you can implement the `fillWithDefaults` method:

```
public static <E> void fillWithDefaults(E[] a)
{
   Class arrayClass = a.getClass();
   Class elementClass = arrayClass.getComponentType();
   try
   {
      for (int i = 0; i < a.length; i++)
      {
         a[i] = elementClass.newInstance();
      }
   }
   catch (. . .) { . . . }
}
```

Note that we must ask for the element type of `a`. It does no good asking for `a[0].getClass`. The array might have length 0, or `a[0]` might be `null`, or `a[0]` might be an instance of a subclass of `E`.

Here is another example. The `Arrays` class implements a method

```
static <T> T[] copyOf(T[] original, int newLength)
```

That method can't simply call

```
T[] result = new T[newLength]; // Error
```

Instead, it must construct a new array with the same element type as the original:

```
Class arrayClass = original.getClass();
Class elementClass = arrayClass.getComponentType();
T[] newArray = (T[]) Array.newInstance(elementClass, newLength);
```

For this technique to work, you must have an element or array of the desired type. You couldn't use it to build a `Stack<E>` that uses an `E[]` array because the stack starts out empty.

WORKED EXAMPLE 18.1 Making a Generic Binary Search Tree Class

In this Worked Example, we will turn the binary search tree class from Chapter 17 into a generic `BinarySearchTree<E>` that stores elements of type `E`.

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

CHAPTER SUMMARY

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 parameter variables, 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.

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 `Object`s.
- You cannot construct objects or arrays of a generic type.

REVIEW EXERCISES

- ■ **R18.1** What is a type parameter?
- ■ **R18.2** What is the difference between a generic class and an ordinary class?
- ■ **R18.3** What is the difference between a generic class and a generic method?
- ■ **R18.4** Find an example of a non-static generic method in the standard Java library.
- ■■ **R18.5** Find four examples of a generic class with two type parameters in the standard Java library.

■■ **R18.6** Find an example of a generic class in the standard library that is not a collection class.

■ **R18.7** Why is a bound required for the type parameter `T` in the following method?

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

■■ **R18.8** Why is a bound not required for the type parameter `E` in the `HashSet<E>` class?

■ **R18.9** What is an `ArrayList<Pair<T, T>>`?

■■ **R18.10** 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?

■ **R18.11** What happens when you pass an `ArrayList<String>` to a method with an `ArrayList` parameter variable? Try it out and explain.

■■■ **R18.12** What happens when you pass an `ArrayList<String>` to a method with an `ArrayList` parameter variable, and the method stores an object of type `BankAccount` into the array list? Try it out and explain.

■■ **R18.13** What is the result of the following test?

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

Try it out and explain.

■■ **R18.14** 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

■ **P18.1** Modify the generic `Pair` class so that both values have the same type.

■ **P18.2** Add a method `swap` to the `Pair` class of Exercise P18.1 that swaps the first and second elements of the pair.

■■ **P18.3** Implement a static generic method `PairUtil.swap` whose argument is a `Pair` object, using the generic class declared in Section 18.2. The method should return a new pair, with the first and second element swapped.

■■ **P18.4** 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.

■■ **P18.5** Repeat Exercise P18.4, but require that the array elements implement the `Comparable` interface.

■■■ **P18.6** Repeat Exercise P18.5, but refine the bound of the type parameter to extend the generic `Comparable` type.

■■ **P18.7** Implement a generic version of the binary search algorithm.

■■ **P18.8** Implement a generic version of the selection sort algorithm.

■■■ **P18.9** Implement a generic version of the merge sort algorithm. Your program should compile without warnings.

■ **P18.10** Implement a generic version of the `LinkedList` class of Chapter 16.

■■ **P18.11** Turn the `HashSet` implementation of Chapter 16 into a generic class. Use an array list instead of an array to store the buckets.

■■ **P18.12** Provide suitable `hashCode` and `equals` methods for the `Pair` class of Section 18.2 and implement a `HashMap` class, using a `HashSet<Pair<K, V>>`.

■■■ **P18.13** Implement a generic version of the permutation generator in Section 13.5. Generate all permutations of a `List<E>`.

■■ **P18.14** 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.

■■ **P18.15** Turn the `MinHeap` class of Chapter 17 into a generic class. As with the `TreeSet` class of the standard library, allow a `Comparator` to compare elements. If no comparator is supplied, assume that the element type implements the `Comparable` interface.

■■ **P18.16** Make the `Measurable` interface from Chapter 9 into a generic class. Provide a static method that returns the largest element of an `ArrayList<T>`, provided that the elements are instances of `Measurable<T>`. Be sure to return a value of type `T`.

■■■ **P18.17** Enhance Exercise P18.16 so that the elements of the `ArrayList<T>` can implement `Measurable<U>` for appropriate types `U`.

■■ **P18.18** Make the `Measurer` interface from Chapter 9 into a generic class. Provide a static method `T max(T[] values, Measurer<T> meas)`.

■ **P18.19** Provide a static method `void append(ArrayList<T> a, ArrayList<T> b)` that appends the elements of `b` to `a`.

■■ **P18.20** Modify the method of Exercise P18.19 so that the second array list can contain elements of a subclass. For example, if `people` is an `ArrayList<Person>` and `students` is an `ArrayList<Student>`, then `append(people, students)` should compile but `append(students, people)` should not.

■■ **P18.21** Modify the method of Exercise P18.19 so that it leaves the first array list unchanged and returns a new array list containing the elements of both array lists.

■■ **P18.22** Modify the method of Exercise P18.21 so that it receives and returns arrays, not array lists. *Hint:* `Arrays.copyOf`.

■ **P18.23** Provide a static method that reverses the elements of a generic array list.

■ **P18.24** Provide a static method that returns the reverse of a generic array list, without modifying the original list.

■■ **P18.25** Provide a static method that checks whether a generic array list is a palindrome; that is, whether the values at index `i` and `n - 1 - i` are equal to each other, where `n` is the size of the array list.

■■ **P18.26** Provide a static method that checks whether the elements of a generic array list are in increasing order. The elements must be comparable.

ANSWERS TO SELF-CHECK QUESTIONS

1. `HashMap<String, Integer>`
2. It uses inheritance.
3. This is a compile-time error. You cannot assign the `Integer` expression `a.get(0)` to a string.
4. This is a compile-time error. The compiler won't convert 3 to a `Double`. *Remedy:* Call `a.add(3.0)`.
5. This is a run-time error. `a.removeFirst()` yields a `String` that cannot be converted into a `Double`. *Remedy:* Call `a.addFirst(3.14);`
6. `new Pair<String, String>("Hello", "World")`
7. `new Pair<String, Integer>("Hello", 1729)`
8. 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)`.
9.
```
public static Pair<Double, Double> roots(
      Double x)
{
   if (x >= 0)
   {
      double r = Math.sqrt(x);
      return new Pair<Double, Double>(r, -r);
   }
   else { return null; }
}
```
10. You have three type parameters: `Triple<T, S, U>`. Add an instance variable `U third`, a constructor argument for initializing it, and a method `U getThird()` for returning it.
11. The output depends on the implementation of the `toString` method in the `BankAccount` class.
12. No—the method has no type parameters. It is an ordinary method in a generic class.
13. `fill(a, "*")`
14. You get a compile-time error. An integer cannot be converted to a string.
15. You get a run-time error. Unfortunately, the call compiles, with `T = Object`. This choice is justified because a `String[]` array is convertible to an `Object[]` array, and 42 becomes `new Integer(42)`, which is convertible to an `Object`. But when the program tries to store an `Integer` in the `String[]` array, an exception is thrown.
16.
```
public class BinarySearchTree<E
      extends Comparable<E>>
```
or, if you read Special Topic 18.1,
```
public class BinarySearchTree<E
      extends Comparable<? superE>>
```
17.
```
public static <E extends Measurable> E min(
      ArrayList<E> objects)
{
   E smallest = objects.get(0);
   for (int i = 1; i < objects.size(); i++)
   {
      E obj = objects.get(i);
      if (obj.getMeasure()
          < smallest.getMeasure())
      {
         smallest = obj;
      }
   }
   return smallest;
}
```
18. No. As described in Common Error 18.1, you cannot convert an `ArrayList<BankAccount>` to an `ArrayList<Measurable>`, even if `BankAccount` implements `Measurable`.
19. Yes, but this method would not be as useful. Suppose `accounts` is an array of `BankAccount` objects. With this method, `min(accounts)` would return a result of type `Measurable`, whereas the generic method yields a `BankAccount`.
20.
```
public static <E extends Measurable>
     double average(E[] objects)
{
   if (objects.length == 0) { return 0; }
   double sum = 0;
   for (E obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objects.length;
}
```
21. No. You can define
```
public static double average(
      Measurable[] objects)
{
   if (objects.length == 0) { return 0; }
   double sum = 0;
   for (Measurable obj : objects)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objects.length;
}
```

For example, if `BankAccount` implements `Measurable`, a `BankAccount[]` array is convertible to a `Measurable[]` array. Contrast with Self Check 19, where the return type was a generic type. Here, the return type is `double`, and there is no need for using generic types.

22.
```
public static <E> Comparable<E> min(
      Comparable<E>[] objects)
```
is an error. You cannot have an array of a generic type.

23.
```
public static void print(Object[] a)
{
   for (Object e : a)
   {
      System.out.print(e + " ");
   }
   System.out.println();
}
```

24. 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.

25. Internally, `ArrayList` uses an `Object[]` array. Because of type erasure, it can't make an `E[]` array. The best it can do is make a copy of its internal `Object[]` array.

26. It can use reflection to discover the element type of the parameter `a`, and then construct another array with that element type (or just call the `Arrays.copyOf` method).

27. The method needs to construct a new array of type `T`. However, that is not possible in Java without reflection.