CHAPTER

19

GENERICS

Objectives

- To describe the benefits of generics (§19.2).
- To use generic classes and interfaces (§19.2).
- To define generic classes and interfaces (§19.3).
- To explain why generic types can improve reliability and readability (§19.3).
- To define and use generic methods and bounded generic types (§19.4).
- To develop a generic sort method to sort an array of **`Comparable`** objects (§19.5).
- To use raw types for backward compatibility (§19.6).
- To explain why wildcard generic types are necessary (§19.7).
- To describe generic-type erasure and list certain restrictions and limitations on generic types caused by type erasure (§19.8).
- To design and implement generic matrix classes (§19.9).

19.1 Introduction

Key Point

Generics enable you to detect errors at compile time rather than at runtime.

what is generics?

You have used a generic class `ArrayList` in Chapter 11, and generic interface `Comparable` in Chapter 13. *Generics* let you parameterize types. With this capability, you can define a class or a method with generic types that the compiler can replace with concrete types. For example, Java defines a generic `ArrayList` class for storing the elements of a generic type. From this generic class, you can create an `ArrayList` object for holding strings, and an `ArrayList` object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

why generics?

The key benefit of generics is to enable errors to be detected at compile time rather than at runtime. A generic class or method permits you to specify allowable types of objects that the class or method can work with. If you attempt to use an incompatible object, the compiler will detect that error.

This chapter explains how to define and use generic classes, interfaces, and methods and demonstrates how generics can be used to improve software reliability and readability. It can be intertwined with Chapter 13, Abstract Classes and Interfaces.

19.2 Motivations and Benefits

Key Point

The motivation for using Java generics is to detect errors at compile time.

Java has allowed you to define generic classes, interfaces, and methods since JDK 1.5. Several interfaces and classes in the Java API were modified using generics. For example, prior to JDK 1.5, the `java.lang.Comparable` interface was defined as shown in Figure 19.1a, but since JDK 1.5, it has been modified as shown in Figure 19.1b.

```
package java.lang;

public interface Comparable {
  public int compareTo(Object o)
}
```

(a) Prior to JDK 1.5

```
package java.lang;

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

(b) JDK 1.5

FIGURE 19.1 The `java.lang.Comparable` interface was modified in JDK 1.5 with a generic type.

formal generic type
actual concrete type
generic instantiation

Here, `<T>` represents a *formal generic type*, which can be replaced later with an *actual concrete type*. Replacing a generic type is called a *generic instantiation*. By convention, a single capital letter such as `E` or `T` is used to denote a formal generic type.

To see the benefits of using generics, let us examine the code in Figure 19.2. The statement in Figure 19.2a declares that `c` is a reference variable whose type is `Comparable` and invokes the `compareTo` method to compare a `Date` object with a string. The code compiles fine, but it has a runtime error because a string cannot be compared with a date.

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

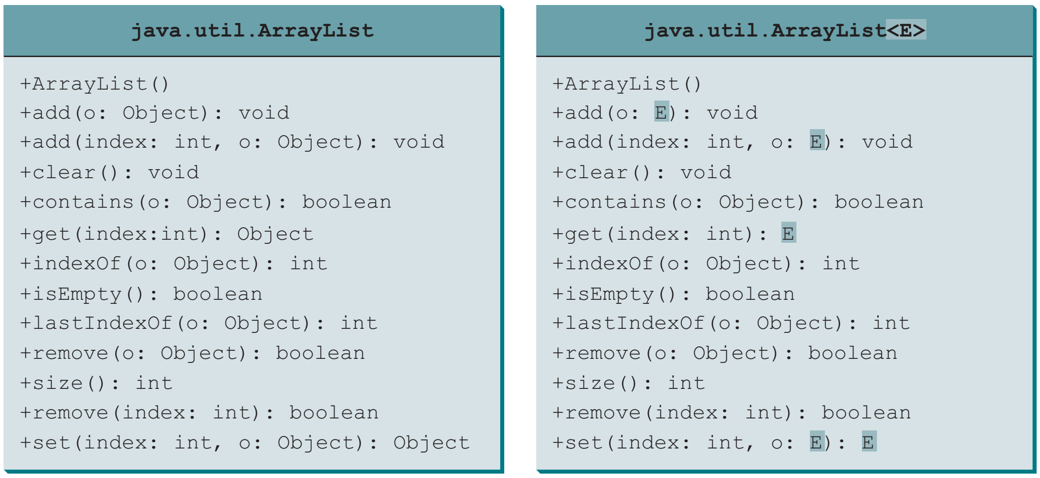
(b) JDK 1.5

FIGURE 19.2 The new generic type detects possible errors at compile time.

The statement in Figure 19.2b declares that `c` is a reference variable whose type is `Comparable<Date>` and invokes the `compareTo` method to compare a `Date` object with a string. This code has a compile error because the argument passed to the `compareTo` method must be of the **`Date`** type. Since the errors can be detected at compile time rather than at runtime, the generic type makes the program more reliable.

reliable

`ArrayList` was introduced in Section 11.11, The **`ArrayList`** Class. This class has been a generic class since JDK 1.5. Figure 19.3 shows the class diagram for **`ArrayList`** before and since JDK 1.5, respectively.


java.util.ArrayList
+ArrayList()
+add(o: Object): void
+add(index: int, o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index:int): Object
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: Object): Object
java.util.ArrayList<E>
+ArrayList()
+add(o: E): void
+add(index: int, o: E): void
+clear(): void
+contains(o: Object): boolean
+get(index: int): E
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: E): E

(a) `ArrayList` before JDK 1.5 (b) `ArrayList` since JDK 1.5

FIGURE 19.3 **`ArrayList`** is a generic class since JDK 1.5.

For example, the following statement creates a list for strings:

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

You can now add *only strings* into the list. For instance,

only strings allowed

```
list.add("Red");
```

If you attempt to add a nonstring, a compile error will occur. For example, the following statement is now illegal because **`list`** can contain only strings.

```
list.add(Integer.valueOf(1));
```

Generic types must be reference types. You cannot replace a generic type with a primitive type such as **`int`**, **`double`**, or **`char`**. For example, the following statement is wrong:

generic reference type

```
ArrayList<int> intList = new ArrayList<>();
```

To create an **`ArrayList`** object for **`int`** values, you have to use

```
ArrayList<Integer> intList = new ArrayList<>();
```

You can add an **`int`** value to **`intList`**. For example,

```
intList.add(5);
```

Java automatically wraps **`5`** into an **`Integer`** object. This is called *autoboxing*, as introduced in Section 10.8, Automatic Conversion between Primitive Types and Wrapper Class Types.

autoboxing

no casting needed

Casting is not needed to retrieve a value from a list with a specified element type because the compiler already knows the element type. For example, the following statements create a list that contains strings, add strings to the list, and retrieve strings from the list.

```
ArrayList<String> list = new ArrayList<>();
list.add("Red");
list.add("White");
String s = list.get(0); // No casting is needed
```

Prior to JDK 1.5, without using generics, you would have had to cast the return value to `String` as

```
String s = (String)(list.get(0)); // Casting needed prior to JDK 1.5
```

autounboxing

If the elements are of wrapper types, such as `Integer`, `Double`, and `Character`, you can directly assign an element to a primitive-type variable. This is called *autounboxing*, as introduced in Section 10.8. For example, see the following code:

```
1 ArrayList<Double> list = new ArrayList<>();
2 list.add(5.5); // 5.5 is automatically converted to a Double object
3 list.add(3.0); // 3.0 is automatically converted to a Double object
4 Double doubleObject = list.get(0); // No casting is needed
5 double d = list.get(1); // Automatically converted to double
```

In lines 2 and 3, `5.5` and `3.0` are automatically converted into `Double` objects and added to `list`. In line 4, the first element in `list` is assigned to a `Double` variable. No casting is necessary because `list` is declared for `Double` objects. In line 5, the second element in `list` is assigned to a `double` variable. The object in `list.get(1)` is automatically converted into a primitive-type value.

Check Point

19.2.1 Are there any compile errors in (a) and (b)?

```
ArrayList dates = new ArrayList();
dates.add(new Date());
dates.add(new String());
```

(a) Prior to JDK 1.5

```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
dates.add(new String());
```

(b) Since JDK 1.5

19.2.2 What is wrong in (a)? Is the code in (b) correct?

```
ArrayList dates = new ArrayList();
dates.add(new Date());
Date date = dates.get(0);
```

(a) Prior to JDK 1.5

```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
Date date = dates.get(0);
```

(b) Since JDK 1.5

19.2.3 What are the benefits of using generic types?

19.3 Defining Generic Classes and Interfaces

Key Point

A generic type can be defined for a class or interface. A concrete type must be specified when using the class to create an object or using the class or interface to declare a reference variable.

Let us revise the stack class in Section 11.13, Case Study: A Custom Stack Class, to generalize the element type with a generic type. The new stack class, named **`GenericStack`**, is shown in Figure 19.4 and is implemented in Listing 19.1.

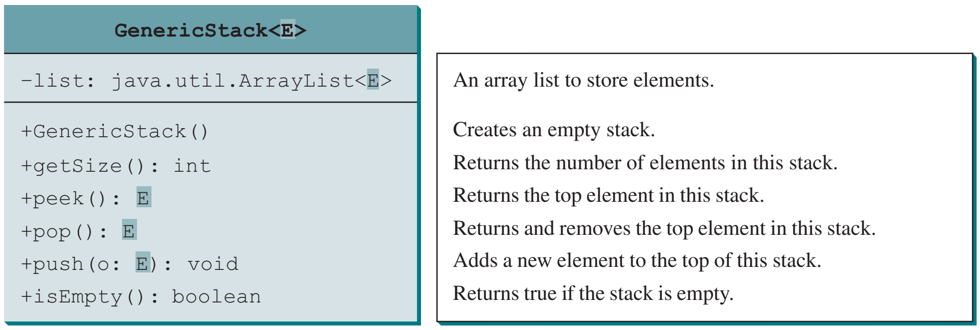

GenericStack<E>
-list: java.util.ArrayList<E>
+GenericStack()
+getSize(): int
+peek(): E
+pop(): E
+push(o: E): void
+isEmpty(): boolean
An array list to store elements.
Creates an empty stack.
Returns the number of elements in this stack.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns true if the stack is empty.

FIGURE 19.4 The **`GenericStack`** class encapsulates the stack storage and provides the operations for manipulating the stack.

LISTING 19.1 GenericStack.java

```java
public class GenericStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();

  public int getSize() {
    return list.size();
  }

  public E peek() {
    return list.get(getSize() - 1);
  }

  public void push(E o) {
    list.add(o);
  }

  public E pop() {
    E o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  @Override
  public String toString() {
    return "stack: " + list.toString();
  }
}
```

generic type E declared
generic array list
`getSize`
`peek`
`push`
`pop`
`isEmpty`

The following example creates a stack to hold strings and adds three strings to the stack:

```java
GenericStack<String> stack1 = new GenericStack<>();
stack1.push("London");
stack1.push("Paris");
stack1.push("Berlin");
```

This example creates a stack to hold integers and adds three integers to the stack:

```
GenericStack<Integer> stack2 = new GenericStack<>();
stack2.push(1); // autoboxing 1 to an Integer object
stack2.push(2);
stack2.push(3);
```

benefits of using generic types

Instead of using a generic type, you could simply make the type element `Object`, which can accommodate any object type. However, using a specific concrete type can improve software reliability and readability because certain errors can be detected at compile time rather than at runtime. For example, because `stack1` is declared `GenericStack<String>`, only strings can be added to the stack. It would be a compile error if you attempted to add an integer to `stack1`.

generic class constructor

Caution

To create a stack of strings, you use `new GenericStack<String>()` or `new GenericStack<>()`. This could mislead you into thinking that the constructor of `GenericStack` should be defined as

```
public GenericStack<E>()
```

This is wrong. It should be defined as

```
public GenericStack()
```

multiple generic parameters

Note

Occasionally, a generic class may have more than one parameter. In this case, place the parameters together inside the brackets, separated by commas—for example, `<E1, E2, E3>`.

inheritance with generics

Note

You can define a class or an interface as a subtype of a generic class or interface. For example, the `java.lang.String` class is defined to implement the `Comparable` interface in the Java API as follows:

```
public class String implements Comparable<String>
```

Check Point

19.3.1 What is the generic definition for `java.lang.Comparable` in the Java API?

19.3.2 Since you create an instance of `ArrayList` of strings using `new ArrayList<String>()`, should the constructor in the `ArrayList` class be defined as

```
public ArrayList<E>()
```

19.3.3 Can a generic class have multiple generic parameters?

19.3.4 How do you declare a generic type in a class?

19.4 Generic Methods

Key Point

A generic type can be defined for a static method.

generic method

You can define generic interfaces (e.g., the `Comparable` interface in Figure 19.1b) and classes (e.g., the `GenericStack` class in Listing 19.1). You can also use generic types to define generic methods. For example, Listing 19.2 defines a generic method `print` (lines 10–14) to print an array of objects. Line 6 passes an array of integer objects to invoke the generic `print` method. Line 7 invokes `print` with an array of strings.

Listing 19.2 GenericMethodDemo.java

```java
public class GenericMethodDemo {
  public static void main(String[] args ) {
    Integer[] integers = {1, 2, 3, 4, 5};
    String[] strings = {"London", "Paris", "New York", "Austin"};

    GenericMethodDemo.<Integer>print(integers);
    GenericMethodDemo.<String>print(strings);
  }

  public static <E> void print(E[] list) {
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println();
  }
}
```

generic method

declare a generic method

To declare a generic method, you place the generic type **<E>** immediately after the keyword **static** in the method header. For example,

```java
public static <E> void print(E[] list)
```

invoke generic method

To invoke a generic method, prefix the method name with the actual type in angle brackets. For example,

```java
GenericMethodDemo.<Integer>print(integers);
GenericMethodDemo.<String>print(strings);
```

or simply invoke it as follows:

```java
print(integers);
print(strings);
```

In the latter case, the actual type is not explicitly specified. The compiler automatically discovers the actual type.

bounded generic type

A generic type can be specified as a subtype of another type. Such a generic type is called *bounded*. For example, Listing 19.3 revises the **equalArea** method in Listing 13.4, TestGeometricObject.java, to test whether two geometric objects have the same area. The bounded generic type **<E extends GeometricObject>** (line 10) specifies that **E** is a generic subtype of **GeometricObject**. You must invoke **equalArea** by passing two instances of **GeometricObject**.

Listing 19.3 BoundedTypeDemo.java

```java
 1  public class BoundedTypeDemo {
 2    public static void main(String[] args ) {
 3      Rectangle rectangle = new Rectangle(2, 2);
 4      Circle circle = new Circle(2);
 5
 6      System.out.println("Same area? " +
 7        equalArea(rectangle, circle));
 8    }
 9
10    public static <E extends GeometricObject> boolean equalArea(
11        E object1, E object2) {
12      return object1.getArea() == object2.getArea();
13    }
14  }
```

Rectangle in Listing 13.3

Circle in Listing 13.2

bounded generic type

Note
An unbounded generic type `<E>` is the same as `<E extends Object>`.

generic class parameter vs. generic method parameter

Note
To define a generic type for a class, place it after the class name, such as `GenericStack<E>`. To define a generic type for a method, place the generic type before the method return type, such as `<E> void max(E o1, E o2)`.

Check Point

19.4.1 How do you declare a generic method? How do you invoke a generic method?

19.4.2 What is a bounded generic type?

19.5 Case Study: Sorting an Array of Objects

Key Point

You can develop a generic method for sorting an array of `Comparable` *objects.*

This section presents a generic method for sorting an array of `Comparable` objects. The objects are instances of the `Comparable` interface and they are compared using the `compareTo` method. To test the method, the program sorts an array of integers, an array of double numbers, an array of characters, and an array of strings. The program is shown in Listing 19.4.

LISTING 19.4 GenericSort.java

sort Integer objects
sort Double objects
sort Character objects
sort String objects

```java
public class GenericSort {
  public static void main(String[] args) {
    // Create an Integer array
    Integer[] intArray = {Integer.valueOf(2), Integer.valueOf(4),
      Integer.valueOf(3)};

    // Create a Double array
    Double[] doubleArray = {Double.valueOf(3.4), Double.valueOf(1.3),
      Double.valueOf(-22.1)};

    // Create a Character array
    Character[] charArray = {Character.valueOf('a'),
      Character.valueOf('J'), Character.valueOf('r')};

    // Create a String array
    String[] stringArray = {"Tom", "Susan", "Kim"};

    // Sort the arrays
    sort(intArray);
    sort(doubleArray);
    sort(charArray);
    sort(stringArray);

    // Display the sorted arrays
    System.out.print("Sorted Integer objects: ");
    printList(intArray);
    System.out.print("Sorted Double objects: ");
    printList(doubleArray);
    System.out.print("Sorted Character objects: ");
    printList(charArray);
    System.out.print("Sorted String objects: ");
    printList(stringArray);
  }

35    /** Sort an array of comparable objects */
36    public static <E extends Comparable<E>> void sort(E[] list) {
37      E currentMin;
38      int currentMinIndex;
39
40      for (int i = 0; i < list.length - 1; i++) {
41        // Find the minimum in the list[i+1..list.length-2]
42        currentMin = list[i];
43        currentMinIndex = i;
44
45        for (int j = i + 1; j < list.length; j++) {
46          if (currentMin.compareTo(list[j]) > 0) {
47            currentMin = list[j];
48            currentMinIndex = j;
49          }
50        }
51
52        // Swap list[i] with list[currentMinIndex] if necessary;
53        if (currentMinIndex != i) {
54          list[currentMinIndex] = list[i];
55          list[i] = currentMin;
56        }
57      }
58    }
59
60    /** Print an array of objects */
61    public static void printList(Object[] list) {
62      for (int i = 0; i < list.length; i++)
63        System.out.print(list[i] + " ");
64      System.out.println();
65    }
66  }
```

generic sort method

compareTo

```
Sorted Integer objects: 2 3 4
Sorted Double objects: -22.1 1.3 3.4
Sorted Character objects: J a r
Sorted String objects: Kim Susan Tom
```

The algorithm for the **sort** method is the same as in Listing 7.8, SelectionSort.java. The **sort** method in that program sorts an array of **double** values. The **sort** method in this example can sort an array of any object type, provided that the objects are also instances of the **Comparable** interface. The generic type is defined as **<E extends Comparable<E>>** (line 36). This has two meanings. First, it specifies that **E** is a subtype of **Comparable**. Second, it specifies that the elements to be compared are of the **E** type as well.

The **sort** method uses the **compareTo** method to determine the order of the objects in the array (line 46). **Integer**, **Double**, **Character**, and **String** implement **Comparable**, so the objects of these classes can be compared using the **compareTo** method. The program creates arrays of **Integer** objects, **Double** objects, **Character** objects, and **String** objects (lines 4–16) and invokes the **sort** method to sort these arrays (lines 19–22).

Check Point

19.5.1 Given **int[] list = {1, 2, -1}**, can you invoke **sort(list)** using the **sort** method in Listing 19.4?

19.5.2 Given **int[] list = {Integer.valueOf(1), Integer.valueOf(2), Integer.valueOf(-1)}**, can you invoke **sort(list)** using the **sort** method in Listing 19.4?

19.6 Raw Types and Backward Compatibility

Key Point

A generic class or interface used without specifying a concrete type, called a raw type, enables backward compatibility with earlier versions of Java.

You can use a generic class without specifying a concrete type such as the following:

```
GenericStack stack = new GenericStack(); // raw type
```

This is roughly equivalent to

```
GenericStack<Object> stack = new GenericStack<Object>();
```

raw type
backward compatibility

A generic class such as **`GenericStack`** and **`ArrayList`** used without a type parameter is called a *raw type*. Using raw types allows for backward compatibility with earlier versions of Java. For example, a generic type has been used in **`java.lang.Comparable`** since JDK 1.5, but a lot of code still uses the raw type **`Comparable`**, as given in Listing 19.5:

LISTING 19.5 Max.java

raw type

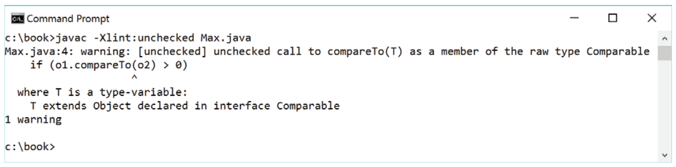
```
1  public class Max {
2    /** Return the maximum of two objects */
3    public static Comparable max(Comparable o1, Comparable o2) {
4      if (o1.compareTo(o2) > 0)
5        return o1;
6      else
7        return o2;
8    }
9  }
```

`Comparable o1` and **`Comparable o2`** are raw type declarations. Be careful: *raw types are unsafe*. For example, you might invoke the **`max`** method using

```
Max.max("Welcome", 23); // 23 is autoboxed into an Integer object
```

-Xlint:unchecked

This would cause a runtime error because you cannot compare a string with an integer object. The Java compiler displays a warning on line 3 when compiled with the option ***`-Xlint:unchecked`***, as shown in Figure 19.5.



```
c:\book>javac -Xlint:unchecked Max.java
Max.java:4: warning: [unchecked] unchecked call to compareTo(T) as a member of the raw type Comparable
    if (o1.compareTo(o2) > 0)
                    ^
  where T is a type-variable:
    T extends Object declared in interface Comparable
1 warning

c:\book>
```

FIGURE 19.5 The unchecked warnings are displayed using the compiler option **`-Xlint:unchecked`**. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

A better way to write the **`max`** method is to use a generic type, as given in Listing 19.6.

LISTING 19.6 MaxUsingGenericType.java

bounded type

```
1  public class MaxUsingGenericType {
2    /** Return the maximum of two objects */
3    public static <E  extends Comparable<E>> E max(E o1, E o2) {
4      if (o1.compareTo(o2) > 0)
5        return o1;
```

```
    else
      return o2;
  }
}
```

If you invoke the **max** method using

```
// 23 is autoboxed into an Integer object
MaxUsingGenericType.max("Welcome", 23);
```

a compile error will be displayed because the two arguments of the **max** method in **MaxUsingGenericType** must have the same type (e.g., two strings or two integer objects). Furthermore, the type **E** must be a subtype of **Comparable<E>**.

As another example, in the following code you can declare a raw type **stack** in line 1, assign **new GenericStack<String>** to it in line 2, and push a string and an integer object to the stack in lines 3 and 4:

```
1  GenericStack stack;
2  stack = new GenericStack<String>();
3  stack.push("Welcome to Java");
4  stack.push(Integer.valueOf(2));
```

However, line 4 is unsafe because the stack is intended to store strings, but an **Integer** object is added into the stack. Line 3 should be okay, but the compiler will show warnings for both line 3 and line 4, because it cannot follow the semantic meaning of the program. All the compiler knows is that stack is a raw type, and performing certain operations is unsafe. Therefore, warnings are displayed to alert potential problems.

Tip

Since raw types are unsafe, this book will not use them from here on.

Check Point

19.6.1 What is a raw type? Why is a raw type unsafe? Why is the raw type allowed in Java?

19.6.2 What is the syntax to declare an **ArrayList** reference variable using the raw type and assign a raw type **ArrayList** object to it?

19.7 Wildcard Generic Types

Key Point

You can use unbounded wildcards, bounded wildcards, or lower bound wildcards to specify a range for a generic type.

What are wildcard generic types, and why are they needed? Listing 19.7 gives an example to demonstrate the needs. The example defines a generic **max** method for finding the maximum in a stack of numbers (lines 12–22). The main method creates a stack of integer objects, adds three integers to the stack, and invokes the **max** method to find the maximum number in the stack.

LISTING 19.7 WildCardNeedDemo.java

```
 1  public class WildCardNeedDemo {
 2    public static void main(String[] args ) {
 3      GenericStack<Integer> intStack = new GenericStack<>();
 4      intStack.push(1); // 1 is autoboxed into an Integer object
 5      intStack.push(2);
 6      intStack.push(-2);
 7
 8      System.out.print("The max number is " + max(intStack));
 9    }
10
```

GenericStack<Integer> type

GenericStack<Number> type

```
11    /** Find the maximum in a stack of numbers */
12    public static double max(GenericStack<Number> stack) {
13      double max = stack.pop().doubleValue(); // Initialize max
14
15      while (!stack.isEmpty()) {
16        double value = stack.pop().doubleValue();
17        if (value > max)
18          max = value;
19      }
20
21      return max;
22    }
23  }
```

The program in Listing 19.7 has a compile error in line 8 because `intStack` is not an instance of `GenericStack<Number>`. Thus, you cannot invoke `max(intStack)`.

The fact is `Integer` is a subtype of `Number`, but `GenericStack<Integer>` is not a subtype of `GenericStack<Number>`. To circumvent this problem, use wildcard generic types. A wildcard generic type has three forms: `?`, `? extends T`, and `? super T`, where `T` is a generic type.

unbounded wildcard

bounded wildcard

lower bound wildcard

The first form, `?`, called an *unbounded wildcard*, is the same as `? extends Object`. The second form, `? extends T`, called a *bounded wildcard*, represents `T` or a subtype of `T`. The third form, `? super T`, called a *lower bound wildcard*, denotes `T` or a supertype of `T`.

You can fix the error by replacing line 12 in Listing 19.7 as follows:

```
public static double max(GenericStack<? extends Number> stack) {
```

`<? extends Number>` is a wildcard type that represents `Number` or a subtype of `Number`, so it is legal to invoke `max(new GenericStack<Integer>())` or `max(new GenericStack<Double>())`.

Listing 19.8 shows an example of using the `?` wildcard in the `print` method that prints objects in a stack and empties the stack. `<?>` is a wildcard that represents any object type. It is equivalent to `<? extends Object>`. What happens if you replace `GenericStack<?>` with `GenericStack<Object>`? It would be wrong to invoke `print(intStack)` because `intStack` is not an instance of `GenericStack<Object>`. Note that `GenericStack<Integer>` is not a subtype of `GenericStack<Object>` even though `Integer` is a subtype of `Object`.

Listing 19.8 AnyWildCardDemo.java

GenericStack<Integer> type

wildcard type

```
public class AnyWildCardDemo {
  public static void main(String[] args) {
    GenericStack<Integer> intStack = new GenericStack<>();
    intStack.push(1); // 1 is autoboxed into an Integer object
    intStack.push(2);
    intStack.push(-2);

    print(intStack);
  }

  /** Prints objects and empties the stack */
  public static void print(GenericStack<?> stack) {
    while (!stack.isEmpty()) {
      System.out.print(stack.pop() + " ");
    }
  }
}
```

When is the wildcard `<? super T>` needed? Consider the example in Listing 19.9. The example creates a stack of strings in `stack1` (line 3) and a stack of objects in `stack2` (line 4) and invokes `add(stack1, stack2)` (line 8) to add the strings in `stack1` into `stack2`. `GenericStack<? super T>` is used to declare `stack2` in line 13. If `<? super T>` is replaced by `<T>`, a compile error will occur on `add(stack1, stack2)` in line 8 because `stack1`'s type is `GenericStack<String>` and `stack2`'s type is `GenericStack<Object>`. `<? super T>` represents type `T` or a supertype of `T`. `Object` is a supertype of `String`.

why `<? Super T>`

LISTING 19.9 SuperWildCardDemo.java

```
 1  public class SuperWildCardDemo {
 2    public static void main(String[] args) {
 3      GenericStack<String> stack1 = new GenericStack<>();
 4      GenericStack<Object> stack2 = new GenericStack<>();
 5      stack2.push("Java");
 6      stack2.push(2);
 7      stack1.push("Sun");
 8      add(stack1, stack2);
 9      AnyWildCardDemo.print(stack2);
10    }
11
12    public static <T> void add(GenericStack<T> stack1,
13        GenericStack<? super T> stack2) {
14      while (!stack1.isEmpty())
15        stack2.push(stack1.pop());
16    }
17  }
```

`GenericStack<String>` type

`<? Super T>` type

This program will also work if the method header in lines 12 and 13 is modified as follows:

```
public static <T> void add(GenericStack<? extends T> stack1,
    GenericStack<T> stack2)
```

The inheritance relationship involving generic types and wildcard types is summarized in Figure 19.6. In this figure, `A` and `B` represent classes or interfaces, and `E` is a generic-type parameter.

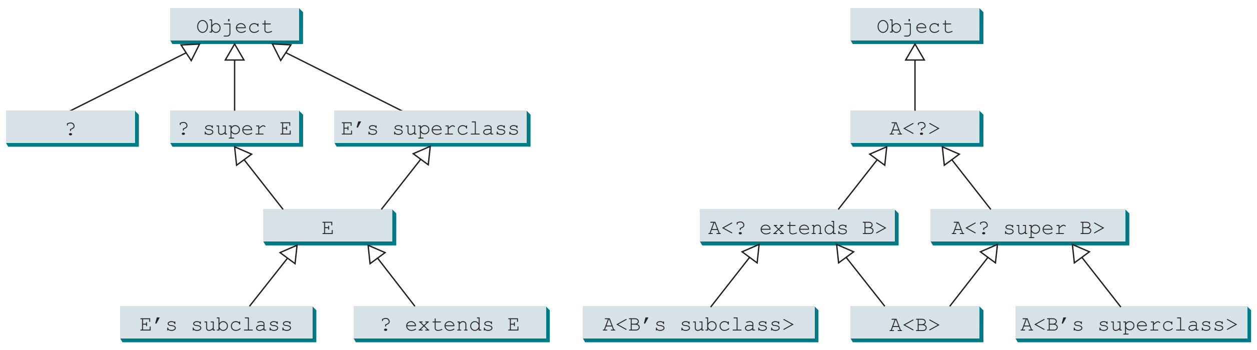


FIGURE 19.6 The relationship between generic types and wildcard types.

Check Point

19.7.1 Is `GenericStack` the same as `GenericStack<Object>`?

19.7.2 What is an unbounded wildcard, a bounded wildcard, and a lower bound wildcard?

19.7.3 What happens if lines 12 and 13 in Listing 19.9 are changed to

```
public static <T> void add(GenericStack<T> stack1,
    GenericStack<T> stack2)
```

19.7.4 What happens if lines 12 and 13 in Listing 19.9 are changed to

```
public static <T> void add(GenericStack<? extends T> stack1,
    GenericStack<T> stack2)
```

19.8 Erasure and Restrictions on Generics

The information on generics is used by the compiler but is not available at runtime. This is called type erasure.

Key Point

type erasure

Generics are implemented using an approach called *type erasure*: The compiler uses the generic-type information to compile the code, but erases it afterward. Thus, the generic information is not available at runtime. This approach enables the generic code to be backward compatible with the legacy code that uses raw types.

erase generics

The generics are present at compile time. Once the compiler confirms that a generic type is used safely, it converts the generic type to a raw type. For example, the compiler checks whether the following code in (a) uses generics correctly, then translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

```
ArrayList<String> list = new ArrayList<>();
list.add("Oklahoma");
String state = list.get(0);
```

(a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

(b)

replace generic type

When generic classes, interfaces, and methods are compiled, the compiler replaces the generic type with the `Object` type. For example, the compiler would convert the following method in (a) into (b).

```
public static <E> void print(E[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(a)

```
public static void print(Object[] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(b)

replace bounded type

If a generic type is bounded, the compiler replaces it with the bounded type. For example, the compiler would convert the following method in (a) into (b).

```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(a)

```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(b)

important fact

It is important to note a generic class is shared by all its instances regardless of its actual concrete type. Suppose `list1` and `list2` are created as follows:

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<Integer> list2 = new ArrayList<>();
```

Although `ArrayList<String>` and `ArrayList<Integer>` are two types at compile time, only one `ArrayList` class is loaded into the JVM at runtime. `list1` and `list2` are both instances of `ArrayList`, so the following statements display `true`:

```
System.out.println(list1 instanceof ArrayList);
System.out.println(list2 instanceof ArrayList);
```

However, the expression `list1 instanceof ArrayList<String>` is wrong. Since `ArrayList<String>` is not stored as a separate class in the JVM, using it at runtime makes no sense.

Because generic types are erased at runtime, there are certain restrictions on how generic types can be used. Here are some of the restrictions:

Restriction 1: Cannot Use *new E()*

You cannot create an instance using a generic-type parameter. For example, the following statement is wrong:

no new E()

```
E object = new E();
```

The reason is `new E()` is executed at runtime, but the generic type `E` is not available at runtime.

Restriction 2: Cannot Use *new E[]*

You cannot create an array using a generic type parameter. For example, the following statement is wrong:

no new E[capacity]

```
E[] elements = new E[capacity];
```

You can circumvent this limitation by creating an array of the `Object` type then casting it to `E[]`, as follows:

```
E[] elements = (E[])new Object[capacity];
```

unavoidable compile warning

However, casting to `(E[])` causes an unchecked compile warning. The warning occurs because the compiler is not certain that casting will succeed at runtime. For example, if `E` is `String` and `new Object[]` is an array of `Integer` objects, `(String[])(new Object[])` will cause a `ClassCastException`. This type of compile warning is a limitation of Java generics and is unavoidable.

Generic array creation using a generic class is not allowed, either. For example, the following code is wrong:

```
ArrayList<String>[] list = new ArrayList<String>[10];
```

You can use the following code to circumvent this restriction:

```
ArrayList<String>[] list = (ArrayList<String>[])new
  ArrayList[10];
```

However, you will still get a compile warning.

Restriction 3: A Generic Type Parameter of a Class Is Not Allowed in a Static Context

Since all instances of a generic class have the same runtime class, the static variables and methods of a generic class are shared by all its instances. Therefore, it is illegal to refer to a generic-type parameter for a class in a static method, field, or initializer. For example, the following code is illegal:

```
public class Test<E> {
  public static void m(E o1) {  // Illegal
  }
```

```
    public static E o1; // Illegal

    static {
      E o2; // Illegal
    }
  }
```

Restriction 4: Exception Classes Cannot Be Generic

A generic class may not extend `java.lang.Throwable`, so the following class declaration would be illegal:

```
public class MyException<T> extends Exception {
}
```

Why? If it were allowed, you would have a `catch` clause for `MyException<T>` as follows:

```
try {
  ...
}
catch (MyException<T> ex) {
  ...
 }
```

The JVM has to check the exception thrown from the `try` clause to see if it matches the type specified in a `catch` clause. This is impossible, because the type information is not present at runtime.

Check Point

19.8.1 What is erasure? Why are Java generics implemented using erasure?

19.8.2 If your program uses `ArrayList<String>` and `ArrayList<Date>`, does the JVM load both of them?

19.8.3 Can you create an instance using `new E()` for a generic type `E`? Why?

19.8.4 Can a method that uses a generic class parameter be static? Why?

19.8.5 Can you define a custom generic exception class? Why?

19.9 Case Study: Generic Matrix Class

Key Point

This section presents a case study on designing classes for matrix operations using generic types.

The addition and multiplication operations for all matrices are similar except that their element types differ. Therefore, you can design a superclass that describes the common operations shared by matrices of all types regardless of their element types, and you can define subclasses tailored to specific types of matrices. This case study gives implementations for two types: `int` and `Rational`. For the `int` type, the wrapper class `Integer` should be used to wrap an `int` value into an object, so the object is passed in the methods for operations.

The class diagram is shown in Figure 19.7. The methods `addMatrix` and `multiplyMatrix` add and multiply two matrices of a generic type `E[][]`. The static method `printResult` displays the matrices, the operator, and their result. The methods `add`, `multiply`, and `zero` are abstract because their implementations depend on the specific type of the array elements. For example, the `zero()` method returns `0` for the `Integer` type and `0/1` for the `Rational` type. These methods will be implemented in the subclasses in which the matrix element type is specified.

`IntegerMatrix` and `RationalMatrix` are concrete subclasses of `GenericMatrix`. These two classes implement the `add`, `multiply`, and `zero` methods defined in the `GenericMatrix` class.

Listing 19.10 implements the `GenericMatrix` class. `<E extends Number>` in line 1 specifies the generic type is a subtype of `Number`. Three abstract methods—`add`, `multiply`,

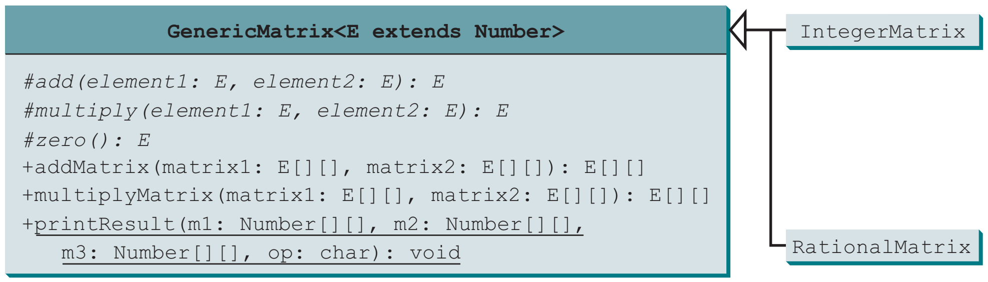

GenericMatrix<E extends Number>
#add(element1: E, element2: E): E
#multiply(element1: E, element2: E): E
#zero(): E
+addMatrix(matrix1: E[][], matrix2: E[][]): E[][]
+multiplyMatrix(matrix1: E[][], matrix2: E[][]): E[][]
+printResult(m1: Number[][], m2: Number[][], m3: Number[][], op: char): void
IntegerMatrix
RationalMatrix

FIGURE 19.7 The `GenericMatrix` class is an abstract superclass for `IntegerMatrix` and `RationalMatrix`.

and `zero`—are defined in lines 3, 6, and 9. These methods are abstract because we cannot implement them without knowing the exact type of the elements. The `addMaxtrix` (lines 12–30) and `multiplyMatrix` (lines 33–57) methods implement the methods for adding and multiplying two matrices. All these methods must be nonstatic because they use generic-type `E` for the class. The `printResult` method (lines 60–84) is static because it is not tied to specific instances.

The matrix element type is a generic subtype of `Number`. This enables you to use an object of any subclass of `Number` as long as you can implement the abstract `add`, `multiply`, and `zero` methods in subclasses.

The `addMatrix` and `multiplyMatrix` methods (lines 12–57) are concrete methods. They are ready to use as long as the `add`, `multiply`, and `zero` methods are implemented in the subclasses.

The `addMatrix` and `multiplyMatrix` methods check the bounds of the matrices before performing operations. If the two matrices have incompatible bounds, the program throws an exception (lines 16 and 36).

LISTING 19.10 GenericMatrix.java

```
 1 public abstract class GenericMatrix<E extends Number> {             bounded generic type
 2   /** Abstract method for adding two elements of the matrices */
 3   protected abstract E add(E o1, E o2);                              abstract method
 4
 5   /** Abstract method for multiplying two elements of the matrices */
 6   protected abstract E multiply(E o1, E o2);                         abstract method
 7
 8   /** Abstract method for defining zero for the matrix element */
 9   protected abstract E zero();                                       abstract method
10
11   /** Add two matrices */
12   public E[][] addMatrix(E[][] matrix1, E[][] matrix2) {             add two matrices
13     // Check bounds of the two matrices
14     if ((matrix1.length != matrix2.length) ||
15         (matrix1[0].length != matrix2[0].length)) {
16       throw new RuntimeException(
17         "The matrices do not have the same size");
18     }
19
20     E[][] result =
21       (E[][])new Number[matrix1.length][matrix1[0].length];
22
23     // Perform addition
24     for (int i = 0; i < result.length; i++)
25       for (int j = 0; j < result[i].length; j++) {
26         result[i][j] = add(matrix1[i][j], matrix2[i][j]);
27       }
28
```

```
    return result;
  }

  /** Multiply two matrices */
  public E[][] multiplyMatrix(E[][] matrix1, E[][] matrix2) {
    // Check bounds
    if (matrix1[0].length != matrix2.length) {
      throw new RuntimeException(
        "The matrices do not have compatible size");
    }

    // Create result matrix
    E[][] result =
      (E[][])new Number[matrix1.length][matrix2[0].length];

    // Perform multiplication of two matrices
    for (int i = 0; i < result.length; i++) {
      for (int j = 0; j < result[0].length; j++) {
        result[i][j] = zero();

        for (int k = 0; k < matrix1[0].length; k++) {
          result[i][j] = add(result[i][j],
            multiply(matrix1[i][k], matrix2[k][j]));
        }
      }
    }

    return result;
  }

  /** Print matrices, the operator, and their operation result */
  public static void printResult(
      Number[][] m1, Number[][] m2, Number[][] m3, char op) {
    for (int i = 0; i < m1.length; i++) {
      for (int j = 0; j < m1[0].length; j++)
        System.out.print(" " + m1[i][j]);

      if (i == m1.length / 2)
        System.out.print("  " + op + "  ");
      else
        System.out.print("     ");

      for (int j = 0; j < m2.length; j++)
        System.out.print(" " + m2[i][j]);

      if (i == m1.length / 2)
        System.out.print("  =  ");
      else
        System.out.print("     ");

      for (int j = 0; j < m3.length; j++)
        System.out.print(m3[i][j] + " ");

      System.out.println();
    }
  }
}
```

multiply two matrices

display result

Listing 19.11 implements the **IntegerMatrix** class. The class extends **GenericMatrix<Integer>** in line 1. After the generic instantiation, the **add** method in **GenericMatrix<Integer>**

is now `Integer add(Integer o1, Integer o2)`. The `add`, `multiply`, and `zero` methods are implemented for `Integer` objects. These methods are still protected because they are invoked only by the `addMatrix` and `multiplyMatrix` methods.

Design Pattern Note

The code in the `GenericMatrix` class applies the template method pattern, which implements a method using abstract methods whose concrete implementation will be provided in the subclasses. In the `GenericMatrix`, the `addMatrix` and `multiplyMatrix` methods are implemented using the abstract `add`, `multiply`, and `zero` methods whose concrete implementation will be provided in the subclasses `IntegerMatrix` and `RationalMatrix`.

Listing 19.11 IntegerMatrix.java

```java
public class IntegerMatrix extends GenericMatrix<Integer> {
  @Override /** Add two integers */
  protected Integer add(Integer o1, Integer o2) {
    return o1 + o2;
  }

  @Override /** Multiply two integers */
  protected Integer multiply(Integer o1, Integer o2) {
    return o1 * o2;
  }

  @Override /** Specify zero for an integer */
  protected Integer zero() {
    return 0;
  }
}
```

extends generic type

implement add

implement `multiply`

implement `zero`

Listing 19.12 implements the `RationalMatrix` class. The `Rational` class was introduced in Listing 13.13, Rational.java. `Rational` is a subtype of `Number`. The `RationalMatrix` class extends `GenericMatrix<Rational>` in line 1. After the generic instantiation, the `add` method in `GenericMatrix<Rational>` is now `Rational add(Rational r1, Rational r2)`. The `add`, `multiply`, and `zero` methods are implemented for `Rational` objects. These methods are still protected because they are invoked only by the `addMatrix` and `multiplyMatrix` methods.

Listing 19.12 RationalMatrix.java

```java
public class RationalMatrix extends GenericMatrix<Rational> {
  @Override /** Add two rational numbers */
  protected Rational add(Rational r1, Rational r2) {
    return r1.add(r2);
  }

  @Override /** Multiply two rational numbers */
  protected Rational multiply(Rational r1, Rational r2) {
    return r1.multiply(r2);
  }

  @Override /** Specify zero for a Rational number */
  protected Rational zero() {
    return new Rational(0, 1);
  }
}
```

extends generic type

implement add

implement `multiply`

implement `zero`

Listing 19.13 gives a program that creates two **Integer** matrices (lines 4 and 5) and an **IntegerMatrix** object (line 8), and adds and multiplies two matrices in lines 12 and 16.

LISTING 19.13 TestIntegerMatrix.java

create matrices

create IntegerMatrix

add two matrices

multiply two matrices

```
 1  public class TestIntegerMatrix {
 2    public static void main(String[] args) {
 3      // Create Integer arrays m1, m2
 4      Integer[][] m1 = new Integer[][]{{1, 2, 3}, {4, 5, 6}, {1, 1, 1}};
 5      Integer[][] m2 = new Integer[][]{{1, 1, 1}, {2, 2, 2}, {0, 0, 0}};
 6
 7      // Create an instance of IntegerMatrix
 8      IntegerMatrix integerMatrix = new IntegerMatrix();
 9
10      System.out.println("\nm1 + m2 is ");
11      GenericMatrix.printResult(
12        m1, m2, integerMatrix.addMatrix(m1, m2), '+');
13
14      System.out.println("\nm1 * m2 is ");
15      GenericMatrix.printResult(
16        m1, m2, integerMatrix.multiplyMatrix(m1, m2), '*');
17    }
18  }
```

```
m1 + m2 is
 1 2 3     1 1 1     2 3 4
 4 5 6  +  2 2 2  =  6 7 8
 1 1 1     0 0 0     1 1 1

m1 * m2 is
 1 2 3     1 1 1     5  5  5
 4 5 6  *  2 2 2  =  14 14 14
 1 1 1     0 0 0     3  3  3
```

Listing 19.14 gives a program that creates two **Rational** matrices (lines 4–10) and a **RationalMatrix** object (line 13) and adds and multiplies two matrices in lines 17 and 19.

LISTING 19.14 TestRationalMatrix.java

create matrices

create RationalMatrix

add two matrices

```
 1  public class TestRationalMatrix {
 2    public static void main(String[] args) {
 3      // Create two Rational arrays m1 and m2
 4      Rational[][] m1 = new Rational[3][3];
 5      Rational[][] m2 = new Rational[3][3];
 6      for (int i = 0; i < m1.length; i++)
 7        for (int j = 0; j < m1[0].length; j++) {
 8          m1[i][j] = new Rational(i + 1, j + 5);
 9          m2[i][j] = new Rational(i + 1, j + 6);
10        }
11
12      // Create an instance of RationalMatrix
13      RationalMatrix rationalMatrix = new RationalMatrix();
14
15      System.out.println("\nm1 + m2 is ");
16      GenericMatrix.printResult(
17        m1, m2, rationalMatrix.addMatrix(m1, m2), '+');
18
19      System.out.println("\nm1 * m2 is ");
```

```
    GenericMatrix.printResult(
      m1, m2, rationalMatrix.multiplyMatrix(m1, m2), '*');
  }
}
```

multiply two matrices

```
m1 + m2 is
 1/5 1/6 1/7     1/6 1/7 1/8     11/30 13/42 15/56
 2/5 1/3 2/7  +  1/3 2/7 1/4  =  11/15 13/21 15/28
 3/5 1/2 3/7     1/2 3/7 3/8     11/10 13/14 45/56

m1 * m2 is
 1/5 1/6 1/7     1/6 1/7 1/8     101/630 101/735 101/840
 2/5 1/3 2/7  *  1/3 2/7 1/4  =  101/315 202/735 101/420
 3/5 1/2 3/7     1/2 3/7 3/8     101/210 101/245 101/280
```

Check Point

19.9.1 Why are the **add**, **multiple**, and **zero** methods defined abstract in the **GenericMatrix** class?

19.9.2 How are the **add**, **multiple**, and **zero** methods implemented in the **IntegerMatrix** class?

19.9.3 How are the **add**, **multiple**, and **zero** methods implemented in the **RationalMatrix** class?

19.9.4 What would be wrong if the **printResult** method is defined as follows?

```
public static void printResult(
  E[][] m1, E[][] m2, E[][] m3, char op)
```

Key Terms

actual concrete type 774
bounded generic type 779
bounded wildcard
 (**<? extends E>**) 784
formal generic type 774
generic instantiation 774
lower bound wildcard
 (**<? super E>**) 784
raw type 782
unbounded wildcard (**<?>**) 784
type erasure 786

Chapter Summary

1. *Generics* give you the capability to parameterize types. You can define a class or a method with generic types, which are substituted with concrete types.

2. The key benefit of generics is to enable errors to be detected at compile time rather than at runtime.

3. A generic class or method permits you to specify allowable types of objects that the class or method can work with. If you attempt to use a class or method with an incompatible object, the compiler will detect the error.

4. A generic type defined in a class, interface, or a static method is called a *formal generic type*, which can be replaced later with an *actual concrete type*. Replacing a generic type is called a *generic instantiation*.

5. A generic class such as `ArrayList` used without a type parameter is called a *raw type*. Use of raw types allows for backward compatibility with the earlier versions of Java.

6. A wildcard generic type has three forms: `?`, `? extends T`, and `? super T`, where `T` is a generic type. The first form, `?`, called an *unbounded wildcard*, is the same as `? extends Object`. The second form, `? extends T`, called a *bounded wildcard*, represents `T` or a subtype of `T`. The third form, `? super T`, called a *lower bound wildcard*, denotes `T` or a supertype of `T`.

7. Generics are implemented using an approach called *type erasure*. The compiler uses the generic-type information to compile the code but erases it afterward, so the generic information is not available at runtime. This approach enables the generic code to be backward compatible with the legacy code that uses raw types.

8. You cannot create an instance using a generic-type parameter such as `new E()`.

9. You cannot create an array using a generic-type parameter such as `new E[10]`.

10. You cannot use a generic-type parameter of a class in a static context.

11. Generic-type parameters cannot be used in exception classes.

Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

Programming Exercises

19.1 (*Revising Listing19.1*) Revise the `GenericStack` class in Listing 19.1 to implement it using an array rather than an `ArrayList`. You should check the array size before adding a new element to the stack. If the array is full, create a new array that doubles the current array size and copy the elements from the current array to the new array.

19.2 (*Implement `GenericStack` using inheritance*) In Listing 19.1, `GenericStack` is implemented using composition. Define a new stack class that extends `ArrayList`.

Draw the UML diagram for the classes then implement `GenericStack`. Write a test program that prompts the user to enter five strings and displays them in reverse order.

19.3 (*Pair of objects of the same type*) Create a `Pair` class that encapsulates two objects of the same data type in an instance of `Pair`.

19.4 (*Using wildcards*) Write a generic static method that returns the smallest value in an instance of `Pair` from Programming Exercise 19.3.

19.5 (*Inheritance between generic classes*) Create a `Triplet` class that encapsulates three objects of the same data type in a given instance of `Triplet`.

19.6 (*Several types*) Create an Association class that encapsulates two objects of different types. Similar to Programming Exercise 19.5, create a Transition class that does the same of Association class with three objects.

19.7 (*Sum of an association*) Knowing that any object of type `java.lang.Number` can be evaluated as a double with its `doubleValue()` method, write a method that computes and returns the sum of the three numbers in an instance of Transition as defined in Programming Exercise 19.6.

19.8 (*Shuffle `ArrayList`*) Write the following method that shuffles an `ArrayList`:

```
public static <E> void shuffle(ArrayList<E> list)
```

19.9 (*Sort `ArrayList`*) Write the following method that sorts an `ArrayList`:

```
public static <E extends Comparable<E>>
  void sort(ArrayList<E> list)
```

Write a test program that prompts the user to enter 10 integers, invokes this method to sort the numbers, and displays the numbers in increasing order.

19.10 (*Smallest element in `ArrayList`*) Write the following method that returns the smallest element in an `ArrayList`:

```
public static <E extends Comparable<E>> E min(ArrayList<E> list)
```

19.11 (*`ComplexMatrix`*) Use the `Complex` class introduced in Programming Exercise 13.17 to develop the `ComplexMatrix` class for performing matrix operations involving complex numbers. The `ComplexMatrix` class should extend the `GenericMatrix` class and implement the `add`, `multiple`, and `zero` methods. You need to modify `GenericMatrix` and replace every occurrence of `Number` by `Object` because `Complex` is not a subtype of `Number`. Write a test program that creates two matrices and displays the result of addition and multiplication of the matrices by invoking the `printResult` method.

***19.12** (Revising Listing 23.4) to make it be generic bubble sort.