Class Declaration

```
public class CashRegister
{
   private int itemCount;
   private double totalPrice;

   public void addItem(double price)
   {
      itemCount++;
      totalPrice = totalPrice + price;
   }
   . . .
}
```

Instance variables

Method

Selected Operators and Their Precedence

(See Appendix B for the complete list.)

`[]`	Array element access
`++ -- !`	Increment, decrement, Boolean *not*
`* / %`	Multiplication, division, remainder
`+ -`	Addition, subtraction
`< <= > >=`	Comparisons
`== !=`	Equal, not equal
`&&`	Boolean *and*
`||`	Boolean *or*
`=`	Assignment

Conditional Statement

```
if (floor >= 13)
{
   actualFloor = floor - 1;
}
else if (floor >= 0)
{
   actualFloor = floor;
}
else
{
   System.out.println("Floor negative");
}
```

Condition

Executed when condition is true

Second condition (optional)

Executed when all conditions are false (optional)

Variable and Constant Declarations

```
int cansPerPack = 6;

final double CAN_VOLUME = 0.335;
```

Type, Name, Initial value

Method Declaration

```
public static double cubeVolume(double sideLength)
{
   double volume = sideLength * sideLength * sideLength;
   return volume;
}
```

Modifiers, Return type, Parameter type and name

Exits method and returns result.

Mathematical Operations

`Math.pow(x, y)`	Raising to a power x^y
`Math.sqrt(x)`	Square root $\sqrt{x}$
`Math.log10(x)`	Decimal log $\log_{10}(x)$
`Math.abs(x)`	Absolute value $\lvert x \rvert$

`Math.sin(x)`
`Math.cos(x)`
`Math.tan(x)`
	Sine, cosine, tangent of x (x in radians)

String Operations

```
String s = "Hello";
int n = s.length(); // 5
char ch = s.charAt(1); // 'e'
String t = s.substring(1, 4); // "ell"
String u = s.toUpperCase(); // "HELLO"
if (u.equals("HELLO")) ... // Use equals, not ==
for (int i = 0; i < s.length(); i++)
{
   char ch = s.charAt(i);
   Process ch
}
```

Loop Statements

```
while (balance < TARGET)
{
   year++;
   balance = balance * (1 + rate / 100);
}
```

Condition

Executed while condition is true

```
for (int i = 0; i < 10; i++)
{
   System.out.println(i);
}
```

Initialization, Condition, Update

```
do
{
   System.out.print("Enter a positive integer: ");
   input = in.nextInt();
}
while (input <= 0);
```

Loop body executed at least once

```
for (double value : values)
{
   sum = sum + value;
}
```

Set to a new element in each iteration

An array or collection

Executed for each element

If you want to write a program that collects objects (such as the stamps to the left), you have a number of choices. Of course, you can use an array list, but computer scientists have invented other mechanisms that may be better suited for the task. In this chapter, we introduce the collection classes and interfaces that the Java library offers. You will learn how to use the Java collection classes, and how to choose the most appropriate collection type for a problem.

15.1 An Overview of the Collections Framework

A collection groups together elements and allows them to be retrieved later.

When you need to organize multiple objects in your program, you can place them into a **collection**. The `ArrayList` class that was introduced in Chapter 7 is one of many collection classes that the standard Java library supplies. In this chapter, you will learn about the Java *collections framework*, a hierarchy of interface types and classes for collecting objects. Each interface type is implemented by one or more classes (see Figure 1).

At the root of the hierarchy is the `Collection` interface. That interface has methods for adding and removing elements, and so on. Table 1 on page 674 shows all the methods. Because all collections implement this interface, its methods are available for all collection classes. For example, the `size` method reports the number of elements in *any* collection.

The `List` interface describes an important category of collections. In Java, a *list* is a collection that remembers the order of its elements (see Figure 2). The `ArrayList` class implements the `List` interface. An `ArrayList` is simply a class containing an array that is expanded as needed. If you are not concerned about efficiency, you can use the `ArrayList` class whenever you need to collect objects. However, several common operations are inefficient with array lists. In particular, if an element is added or removed, the elements at larger positions must be moved.

The Java library supplies another class, `LinkedList`, that also implements the `List` interface. Unlike an array list, a linked list allows efficient insertion and removal of elements in the middle of the list. We will discuss that class in the next section.

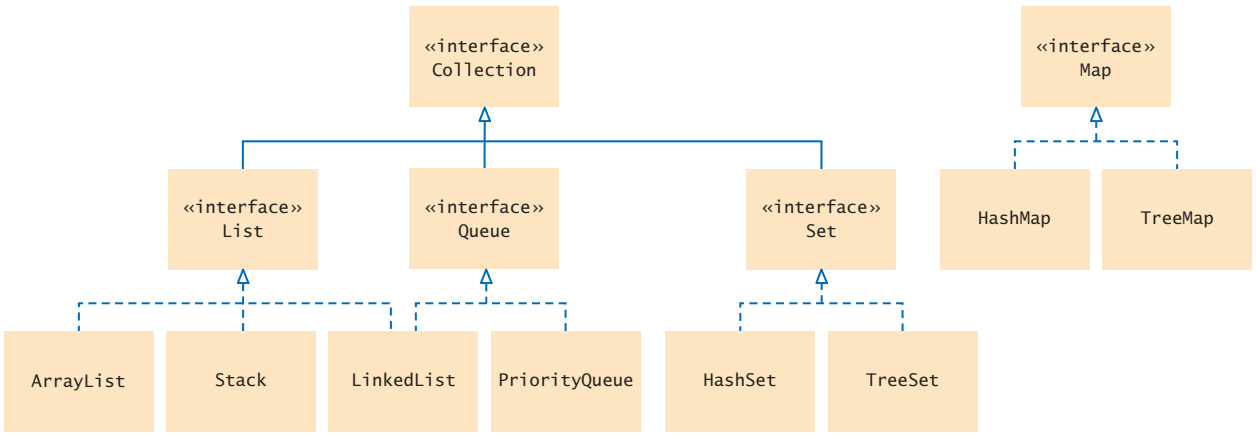


Figure 1 Interfaces and Classes in the Java Collections Framework

15.4 Maps

The HashMap and TreeMap classes both implement the Map interface.

A **map** allows you to associate elements from a *key set* with elements from a *value collection*. You use a map when you want to look up objects by using a key. For example, Figure 10 shows a map from the names of people to their favorite colors.

Just as there are two kinds of set implementations, the Java library has two implementations for the Map interface: HashMap and TreeMap.

After constructing a HashMap or TreeMap, you can store the reference to the map object in a Map reference:

```
Map<String, Color> favoriteColors = new HashMap<String, Color>();
```

Use the put method to add an association:

```
favoriteColors.put("Juliet", Color.RED);
```

You can change the value of an existing association, simply by calling put again:

```
favoriteColors.put("Juliet", Color.BLUE);
```

The get method returns the value associated with a key.

```
Color julietsFavoriteColor = favoriteColors.get("Juliet");
```

If you ask for a key that isn't associated with any values, then the get method returns null.

To remove an association, call the remove method with the key:

```
favoriteColors.remove("Juliet");
```

Table 5 Working with Maps

`Map<String, Integer> scores;`	Keys are strings, values are Integer wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<String, Integer>();`	Use a HashMap if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or null if the key is not present. n is 100, n2 is null.
`System.out.println(scores);`	Prints scores.toString(), a string of the form {Harry=90, Sally=100}

`for (String key : scores.keySet())`
`{`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

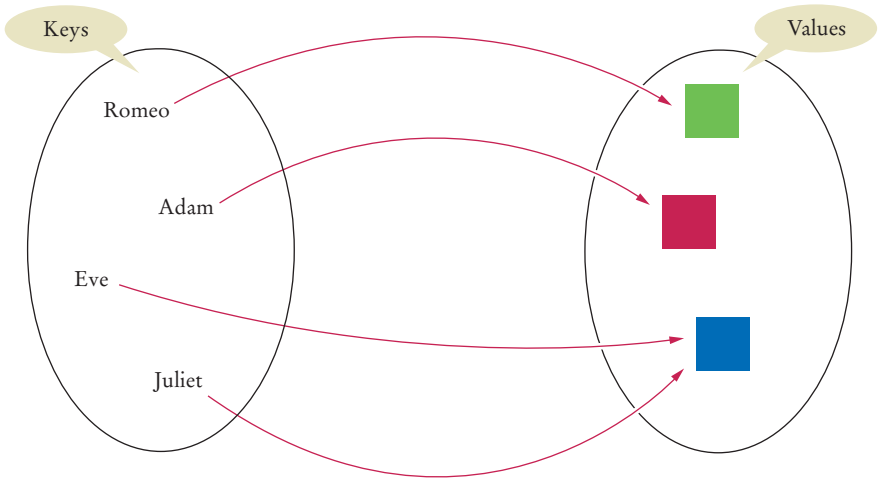


Figure 10 A Map

To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.

Sometimes you want to enumerate all keys in a map. The `keySet` method yields the set of keys. You can then ask the key set for an iterator and get all keys. From each key, you can find the associated value with the `get` method. Thus, the following instructions print all key/value pairs in a map `m`:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
   Color value = m.get(key);
   System.out.println(key + "->" + value);
}
```

This sample program shows a map in action:

section_4/MapDemo.java

```
1  import java.awt.Color;
2  import java.util.HashMap;
3  import java.util.Map;
4  import java.util.Set;
5
6  /**
7     This program demonstrates a map that maps names to colors.
8  */
9  public class MapDemo
10 {
11    public static void main(String[] args)
12    {
13       Map<String, Color> favoriteColors = new HashMap<String, Color>();
14       favoriteColors.put("Juliet", Color.BLUE);
15       favoriteColors.put("Romeo", Color.GREEN);
16       favoriteColors.put("Adam", Color.RED);
17       favoriteColors.put("Eve", Color.BLUE);
18
19       // Print all keys and values in the map
20
21       Set<String> keySet = favoriteColors.keySet();
```

```
22         for (String key : keySet)
23         {
24            Color value = favoriteColors.get(key);
25            System.out.println(key + " : " + value);
26         }
27      }
28   }
```

Program Run

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

16. What is the difference between a set and a map?
17. Why is the collection of the keys of a map a set and not a list?
18. Why is the collection of the values of a map not a set?
19. Suppose you want to track how many times each word occurs in a document. Declare a suitable map variable.
20. What is a `Map<String, HashSet<String>>`? Give a possible use for such a structure.

Practice It Now you can try these exercises at the end of the chapter: R15.17, E15.4, E15.5.

HOW TO 15.1

Choosing a Collection

Suppose you need to store objects in a collection. You have now seen a number of different data structures. This How To reviews how to pick an appropriate collection for your application.

Step 1 Determine how you access the values.

You store values in a collection so that you can later retrieve them. How do you want to access individual values? You have several choices:

- Values are accessed by an integer position. Use an `ArrayList`.
- Values are accessed by a key that is not a part of the object. Use a map.
- Values are accessed only at one of the ends. Use a queue (for first-in, first-out access) or a stack (for last-in, first-out access).
- You don't need to access individual values by position. Refine your choice in Steps 3 and 4.

Step 2 Determine the element types or key/value types.

For a list or set, determine the type of the elements that you want to store. For example, if you collect a set of books, then the element type is `Book`.

Similarly, for a map, determine the types of the keys and the associated values. If you want to look up books by ID, you can use a `Map<Integer, Book>` or `Map<String, Book>`, depending on your ID type.

Step 3 Determine whether element or key order matters.

When you visit elements from a collection or keys from a map, do you care about the order in which they are visited? You have several choices:

- Elements or keys must be sorted. Use a `TreeSet` or `TreeMap`. Go to Step 6.
- Elements must be in the same order in which they were inserted. Your choice is now narrowed down to a `LinkedList` or an `ArrayList`.
- It doesn't matter. As long as you get to visit all elements, you don't care in which order. If you chose a map in Step 1, use a `HashMap` and go to Step 5.

Step 4 For a collection, determine which operations must be efficient.

You have several choices:

- Finding elements must be efficient. Use a `HashSet`.
- It must be efficient to add or remove elements at the beginning, or, provided that you are already inspecting an element there, another position. Use a `LinkedList`.
- You only insert or remove at the end, or you collect so few elements that you aren't concerned about speed. Use an `ArrayList`.

Step 5 For hash sets and maps, decide whether you need to implement the `hashCode` and `equals` methods.

- If your elements or keys belong to a class that someone else implemented, check whether the class has its own `hashCode` and `equals` methods. If so, you are all set. This is the case for most classes in the standard Java library, such as `String`, `Integer`, `Rectangle`, and so on.
- If not, decide whether you can compare the elements by identity. This is the case if you never construct two distinct elements with the same contents. In that case, you need not do anything—the `hashCode` and `equals` methods of the `Object` class are appropriate.
- Otherwise, you need to implement your own `equals` and `hashCode` methods—see Section 9.5.2 and Special Topic 15.1.

Step 6 If you use a tree, decide whether to supply a comparator.

Look at the class of the set elements or map keys. Does that class implement the `Comparable` interface? If so, is the sort order given by the `compareTo` method the one you want? If yes, then you don't need to do anything further. This is the case for many classes in the standard library, in particular for `String` and `Integer`.

If not, then your element class must implement the `Comparable` interface (Section 10.3), or you must declare a class that implements the `Comparator` interface (see Special Topic 14.5).

WORKED EXAMPLE 15.1 **Word Frequency**

Learn how to create a program that reads a text file and prints a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file. Go to `wiley.com/go/javaexamples` and download Worked Example 15.1.

Special Topic 15.1

Hash Functions

If you use a hash set or hash map with your own classes, you may need to implement a hash function. A **hash function** is a function that computes an integer value, the **hash code**, from an object in such a way that different objects are likely to yield different hash codes. Because hashing is so important, the `Object` class has a `hashCode` method. The call

```
int h = x.hashCode();
```

computes the hash code of any object `x`. If you want to put objects of a given class into a `HashSet` or use the objects as keys in a `HashMap`, the class should override this method. The method should be implemented so that different objects are likely to have different hash codes.

A good hash function produces different hash values for each object so that they are scattered about in a hash table.

A hash function computes an integer value from an object.

For example, the `String` class declares a hash function for strings that does a good job of producing different integer values for different strings. Table 6 shows some examples of strings and their hash codes.

A good hash function minimizes *collisions*—identical hash codes for different objects.

It is possible for two or more distinct objects to have the same hash code; this is called a *collision*. For example, the strings `"Ugh"` and `"VII"` happen to have the same hash code, but these collisions are very rare for strings (see Exercise P15.4).

The `hashCode` method of the `String` class combines the characters of a string into a numerical code. The code isn't simply the sum of the character values—that would not scramble the character values enough. Strings that are permutations of another (such as `"eat"` and `"tea"`) would all have the same hash code.

Here is the method the standard library uses to compute the hash code for a string:

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
   h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

For example, the hash code of `"eat"` is

```
31 * (31 * 'e' + 'a') + 't' = 100184
```

Table 6 Sample Strings and Their Hash Codes

String	Hash Code
`"eat"`	100184
`"tea"`	114704
`"Juliet"`	–2065036585
`"Ugh"`	84982
`"VII"`	84982

CHAPTER SUMMARY

Understand the architecture of the Java collections framework.

- A collection groups together elements and allows them to be retrieved later.
- A list is a collection that remembers the order of its elements.
- A set is an unordered collection of unique elements.
- A map keeps associations between key and value objects.

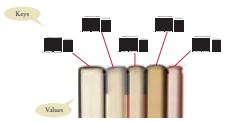
Understand and use linked lists.

- A linked list consists of a number of nodes, each of which has a reference to the next node.
- Adding and removing elements at a given position in a linked list is efficient.
- Visiting the elements of a linked list in sequential order is efficient, but random access is not.
- You use a list iterator to access elements inside a linked list.

Choose a set implementation and use it to manage sets of values.

- The `HashSet` and `TreeSet` classes both implement the `Set` interface.
- Set implementations arrange the elements so that they can locate them quickly.
- You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.
- You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.
- Sets don't have duplicates. Adding a duplicate of an element that is already present is ignored.
- A set iterator visits the elements in the order in which the set implementation keeps them.
- You cannot add an element to a set at an iterator position.

Use maps to model associations between keys and values.



- The `HashMap` and `TreeMap` classes both implement the `Map` interface.
- To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.
- A hash function computes an integer value from an object.
- A good hash function minimizes *collisions*—identical hash codes for different objects.
- Override `hashCode` methods in your own classes by combining the hash codes for the instance variables.
- A class's `hashCode` method must be compatible with its `equals` method.

Use the Java classes for stacks, queues, and priority queues.

- A stack is a collection of elements with "last-in, first-out" retrieval.
- A queue is a collection of elements with "first-in, first-out" retrieval.
- When removing an element from a priority queue, the element with the most urgent priority is retrieved.

Solve programming problems using stacks and queues.

- A stack can be used to check whether parentheses in an expression are balanced.
- Use a stack to evaluate expressions in reverse Polish notation.
- Using two stacks, you can evaluate expressions in standard algebraic notation.
- Use a stack to remember choices you haven't yet made so that you can backtrack to them.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

java.util.Collection<E>
- add
- contains
- iterator
- remove
- size

java.util.HashMap<K, V>
java.util.HashSet<K, V>
java.util.Iterator<E>
- hasNext
- next
- remove

java.util.LinkedList<E>
- addFirst
- addLast
- getFirst
- getLast
- removeFirst
- removeLast

java.util.List<E>
- listIterator

java.util.ListIterator<E>
- add
- hasPrevious
- previous
- set

java.util.Map<K, V>
- get
- keySet
- put
- remove

java.util.Queue<E>
- peek

java.util.PriorityQueue<E>
- remove

java.util.Set<E>
java.util.Stack<E>
- peek
- pop
- push

java.util.TreeMap<K, V>
java.util.TreeSet<K, V>

REVIEW QUESTIONS

R15.1 An invoice contains a collection of purchased items. Should that collection be implemented as a list or set? Explain your answer.

R15.2 Consider a program that manages an appointment calendar. Should it place the appointments into a list, stack, queue, or priority queue? Explain your answer.

R15.3 One way of implementing a calendar is as a map from date objects to event objects. However, that only works if there is a single event for a given date. How can you use another collection type to allow for multiple events on a given date?

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.

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

A generic class has one or more type parameters.

```
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 variables.

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

Type parameters can be instantiated with class or interface types.

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

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

When you instantiate a generic class, the type that you supply replaces all occurrences of the type variable in the declaration of the class. For example, the `add` method for `ArrayList<BankAccount>` has the type variable `E` replaced with the type `BankAccount`:

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

Contrast that with the `add` method of the `LinkedList` class in Chapter 16:

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

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download 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.

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);
```

■■ **E18.9** 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>>`.

■■■ **E18.10** Implement a generic version of the permutation generator in Section 13.4. Generate all permutations of a `List<E>`.

■■ **E18.11** 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.

■■ **E18.12** 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.

■■ **E18.13** Make the `Measurer` interface from Chapter 10 into a generic class. Provide a static method `T max(T[] values, Measurer<T> meas)`.

■ **E18.14** Provide a static method `void append(ArrayList<T> a, ArrayList<T> b)` that appends the elements of `b` to `a`.

■■ **E18.15** Modify the method of Exercise E18.14 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.

■■ **E18.16** Modify the method of Exercise E18.14 so that it leaves the first array list unchanged and returns a new array list containing the elements of both array lists.

■■ **E18.17** Modify the method of Exercise E18.16 so that it receives and returns arrays, not array lists. *Hint:* `Arrays.copyOf`.

■ **E18.18** Provide a static method that reverses the elements of a generic array list.

■ **E18.19** Provide a static method that returns the reverse of a generic array list, without modifying the original list.

■■ **E18.20** 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.

■■ **E18.21** Provide a static method that checks whether the elements of a generic array list are in increasing order. The elements must be comparable.

PROGRAMMING PROJECTS

■■ **P18.1** 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 10.

■■ **P18.2** Repeat Exercise P18.1, but require that the array elements implement the `Comparable` interface.

■■■ **P18.3** Repeat Exercise P18.2, but refine the bound of the type parameter to extend the generic `Comparable` type.

■■ **P18.4** Make the `Measurable` interface from Chapter 10 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`.

■■■ **P18.5** Enhance Exercise P18.4 so that the elements of the `ArrayList<T>` can implement `Measurable<U>` for appropriate types `U`.

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`.

```
            java.util.LinkedList<E> implements List<E>, Queue<E>, Serializable
         java.util.ArrayList<E> implements List<E>, Serializable
      java.util.AbstractQueue<E>
         java.util.PriorityQueue<E> implements Serializable
      java.util.AbstractSet<E>
         java.util.HashSet<E> implements Serializable, Set<E>
         java.util.TreeSet<E> implements Serializable, SortedSet<E>
   java.util.AbstractMap<K, V>
      java.util.HashMap<K, V> implements Map<K, V>, Serializable
         java.util.LinkedHashMap<K, V>
      java.util.TreeMap<K, V> implements Serializable, Map<K, V>
   java.util.Arrays
   java.util.Collections
   java.util.Calendar
      java.util.GregorianCalendar
   java.util.Date implements Serializable
   java.util.Dictionary<K, V>
      java.util.Hashtable<K, V>
         java.util.Properties implements Serializable
   java.util.EventObject implements Serializable
      java.awt.AWTEvent
         java.awt.event.ActionEvent
         java.awt.event.ComponentEvent
            java.awt.event.InputEvent
               java.awt.event.KeyEvent
               java.awt.event.MouseEvent
      javax.swing.event.ChangeEvent
   java.util.Random implements Serializable
   java.util.Scanner
   java.util.TimeZone implements Serializable
   java.util.concurrent.locks.ReentrantLock implements Lock, Serializable
   java.util.logging.Level implements Serializable
   java.util.logging.Logger
   javax.swing.ButtonGroup implements Serializable
   javax.swing.ImageIcon implements Serializable
   javax.swing.Keystroke implements Serializable
   javax.swing.Timer implements Serializable
   javax.swing.border.AbstractBorder implements Serializable
      javax.swing.border.EtchedBorder
      javax.swing.border.TitledBorder
   javax.xml.parsers.DocumentBuilder
   javax.xml.parsers.DocumentBuilderFactory
   javax.xml.xpath.XPathFactory
java.lang.Comparable<T>
java.lang.Runnable
java.sql.Connection
java.sql.ResultSet
java.sql.ResultSetMetaData
java.sql.Statement
   java.sql.PreparedStatement
java.util.Collection<E>
   java.util.List<E>
   java.util.Set<E>
      java.util.SortedSet<E>
java.util.Comparator<T>
java.util.EventListener
   java.awt.event.ActionListener
   java.awt.event.KeyListener
   java.awt.event.MouseListener
   javax.swing.event.ChangeListener
```

Class java.util.GregorianCalendar

- **GregorianCalendar**()
  This constructs a calendar object that represents the current date and time.
- **GregorianCalendar**(int year, int month, int day)
  This constructs a calendar object that represents the start of the given date.
  Parameters: year, month, day The given date

Class java.util.HashMap<K, V>

- **HashMap**<K, V>()
  This constructs an empty hash map.

Class java.util.HashSet<E>

- **HashSet**<E>()
  This constructs an empty hash set.

Class java.util.InputMismatchException

This exception is thrown if the next available input item does not match the type of the requested item.

Interface java.util.Iterator<E>

- boolean **hasNext**()
  This method checks whether the iterator is past the end of the list.
  Returns: true if the iterator is not yet past the end of the list
- E **next**()
  This method moves the iterator over the next element in the linked list. This method throws an exception if the iterator is past the end of the list.
  Returns: The object that was just skipped over
- void **remove**()
  This method removes the element that was returned by the last call to next or previous. This method throws an exception if there was an add or remove operation after the last call to next or previous.

Class java.util.LinkedHashMap<K, V>

- **LinkedHashMap**<K, V>()
  This constructs an empty linked hash map. The iterator of a linked hash map visits the entries in the order in which they were added to the map.

Class java.util.LinkedList<E>

- void **addFirst**(E element)
- void **addLast**(E element)
  These methods add an element before the first or after the last element in this list.
  Parameters: element The element to be added

H

I

M

Input

```
Scanner in = new Scanner(System.in);
  // Can also use new Scanner(new File("input.txt"));

int n = in.nextInt();
double x = in.nextDouble();
String word = in.next();
String line = in.nextLine();

while (in.hasNextDouble())
{
   double x = in.nextDouble();
   Process x
}
```

Output

Does not advance to new line.

```
System.out.print("Enter a value: ");
```

Use + to concatenate values.

```
System.out.println("Volume: " + volume);
```

Field width, Precision

```
System.out.printf("%-10s %10d %10.2f", name, qty, price);
```

Left-justified string, Integer, Floating-point number

```
PrintWriter out = new PrintWriter("output.txt");

out.close();
```

Use print/println/printf to write output to file.

Remember to close output file.

Arrays

Element type, Element type, Length

```
int[] numbers = new int[5];
int[] squares = { 0, 1, 4, 9, 16 };
int[][] magicSquare =
   {
      { 16,  3,  2, 13},
      {  5, 10, 11,  8},
      {  9,  6,  7, 12},
      {  4, 15, 14,  1}
   };

for (int i = 0; i < numbers.length; i++)
{
   numbers[i] = i * i;
}

for (int element : numbers)
{
   Process element
}

System.out.println(Arrays.toString(numbers));
   // Prints [0, 1, 4, 9, 16]
```

All elements are zero.

Array Lists

Initially empty

Use wrapper type, Integer, Double, etc., for primitive types.

Element type

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

Add elements to the end

```
names.add("Ann");
names.add("Cindy"); // [Ann, Cindy], names.size() is now 2

names.add(1, "Bob"); // [Ann, Bob, Cindy]
names.remove(2); // [Ann, Bob]
names.set(1, "Bill"); // [Ann, Bill]

String name = names.get(0); // Gets "Ann"
System.out.println(names); // Prints [Ann, Bill]
```

Linked Lists, Sets, and Iterators

```
LinkedList<String> names = new LinkedList<String>();
names.add("Bob"); // Adds at end

ListIterator<String> iter = names.listIterator();
iter.add("Ann"); // Adds before current position

String name = iter.next(); // Returns "Ann"
iter.remove(); // Removes "Ann"

Set<String> names = new HashSet<String>();
names.add("Ann"); // Adds to set if not present
names.remove("Bob"); // Removes if present

Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
   Process iter.next()
}
```

Maps

Key type, Value type

```
Map<String, Integer> scores = new HashMap<String, Integer>();

scores.put("Bob", 10);
Integer score = scores.get("Bob");
```

Returns null if key not present

```
for (String key : scores.keySet())
{
   Process key and scores.get(key)
}
```