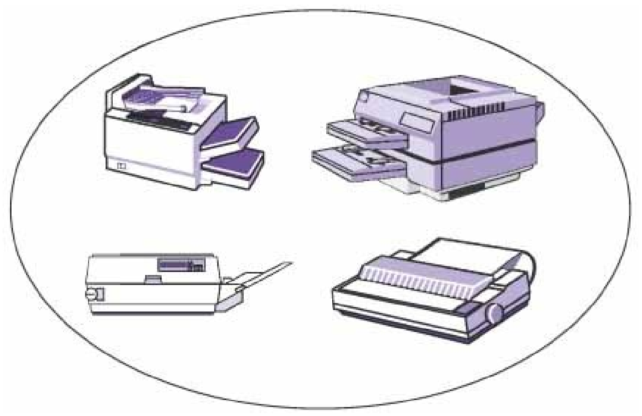
Introduction to Sets

You have encountered two important data structures: arrays and lists. Both have one characteristic in common: These data structures keep the elements in the same order in which you inserted them. However, in many applications, you don't really care about the order of the elements in a collection. For example, a server may keep a collection of objects representing available printers. The order of the objects doesn't really matter.



In mathematics, such an unordered collection is called a ***set***. You have probably learned some set theory in a course in mathematics, and you may know that sets are a fundamental mathematical notion. A set is an unordered collection of distinct elements. Elements can be added, located, and removed. But what does that mean for data structures? If the data structure is no longer responsible for remembering the order of element insertion, can it give us better performance for some of its operations? It turns out that it can indeed, as you will see later in this chapter.

Let's list the fundamental operations on a set:

- Adding an element
- Removing an element
- Containment testing (does the set contain a given object?)
- Listing all elements (in arbitrary order)

In mathematics, a set rejects duplicates. If an object is already in the set, an attempt to add it again is ignored. That's useful in many programming situations as well. For example, if we keep a set of available printers, each printer should occur at most once in the set. Thus, we will interpret the `add` and `remove` operations of sets just as we do in mathematics: Adding an element has no effect if the element is already in the set, and attempting to remove an element that isn't in the set is silently ignored. Sets don't have duplicates. Adding a duplicate of an element that is already present is silently ignored. Of course, we could use a linked list to implement a set. But adding, removing, and containment testing would be relatively slow, because they all have to do a linear search through the list. (Adding requires a search through the list to make sure that we don't add a duplicate.) As you will see later in this chapter, there are data structures that can handle these operations much more quickly. In fact, there are two different data structures for this purpose, called ***hash tables*** and ***trees***. The standard Java library provides set implementations based on both data structures, called `HashSet` and `TreeSet`. Both of these data structures implement the `Set` interface

Quality Tip 16.1

It is considered good style to store a reference to a `HashSet` or `TreeSet` in a variable of type `Set`.

```
Set<String> names = new HashSet<String>();
```

This way, you have to change only one line if you decide to use a `TreeSet` instead. Also, methods that operate on sets should specify parameters of type `Set`:

```
public static void print(Set<String> s)
```

Then the method can be used for all set implementations.

Using Tree Sets and Tree Maps

Both the `HashSet` and the `TreeSet` classes implement the `Set` interface. Thus, if you need a set of objects, you have a choice. If you have a good hash function for your objects, then hashing is usually faster than tree-based algorithms. But the balanced trees used in the `TreeSet` class can *guarantee* reasonable performance, whereas the `HashSet` is entirely at the mercy of the hash function. The `TreeSet` class uses a form of balanced binary tree that guarantees that adding and removing an element takes $O(\log(n))$ time. If you don't want to define a hash function, then a tree set is an attractive option. Tree sets have another advantage: The iterators visit elements in *sorted order* rather than the completely random order given by the hash codes. To use a `TreeSet`, your objects must belong to a class that implements the `Comparable` interface or you must supply a `Comparator` object. That is the same requirement as for using the sort and `binarySearch` methods in the standard library. To use a tree set, the elements must be comparable. To use a `TreeMap`, the same requirement holds for the *keys*. There is no requirement for the values. For example, the `String` class implements the `Comparable` interface. The `compareTo` method compares strings in dictionary order. Thus, you can form tree sets of strings, and use strings as keys for tree maps. If the class of the tree set elements doesn't implement the `Comparable` interface, or the sort order of the `compareTo` method isn't the one you want, then you can define your own comparison by supplying a `Comparator` object to the `TreeSet` or `TreeMap` constructor. For example,

```
Comparator comp = new CoinComparator();
Set s = new TreeSet(comp);
```

A `Comparator` object compares two elements and returns a negative integer if the first is less than the second, zero if they are identical, and a positive value otherwise. The example program below constructs a `TreeSet` of `Coin` objects.

TreeSetTester.java

```
import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

// A program to a test a tree set with a comparator for coins.

public class TreeSetTester
{
  public static void main(String[] args)
   {
      Coin coin1 = new Coin(0.25, "quarter");
      Coin coin2 = new Coin(0.25, "quarter");
      Coin coin3 = new Coin(0.01, "penny");
      Coin coin4 = new Coin(0.05, "nickel");

      class CoinComparator imxplements Comparator<Coin>
      {
         public int compare(Coin first, Coin second)
         {
            if (first.getValue() < second.getValue()) return -1;
            if (first.getValue() == second.getValue()) return 0;
            return 1;
         }
      }

      Comparator<Coin> comp = new CoinComparator();
      Set<Coin> coins = new TreeSet<Coin>(comp);
      coins.add(coin1);
      coins.add(coin2);
      coins.add(coin3);
      coins.add(coin4);

      for (Coin c : coins)
      System.out.print(c.getValue() + " ");
   }
}
```

Output

```
0.01 0.05 0.25
```

Taken from: BIG JAVA, 3rd Edition by Cay Horstmann, Wiley 2008