Applications often must manage collections of objects. Although you can use arrays for this purpose, they are not always a good choice. For example, arrays have fixed sizes, making it tricky to determine an optimal size when you need to store a variable number of objects. Also, arrays can be indexed by integers only, making them unsuitable for mapping arbitrary objects to other objects.

The standard class library provides the Collections Framework and legacy utility APIs to manage collections on behalf of applications. In Chapter 9 I first present this framework and then introduce you to these legacy APIs (in case you encounter them in legacy code). As you will discover, some of the legacy APIs are still useful.

**Note**  
Java’s Concurrency Utilities API suite (discussed in Chapter 10) extends the Collections Framework.

### Exploring Collections Framework Fundamentals

The *Collections Framework* is a group of types (mainly located in the `java.util` package) that offers a standard architecture for representing and manipulating *collections*, which are groups of objects stored in instances of classes designed for this purpose. This framework’s architecture is divided into three sections:

- **Core interfaces**: The framework provides core interfaces for manipulating collections independently of their implementations.
- **Implementation classes**: The framework provides classes that provide different core interface implementations to address performance and other requirements.
- **Utility classes**: The framework provides utility classes with methods for sorting arrays, obtaining synchronized collections, and more.
The core interfaces include `java.lang.Iterable`, `Collection`, `List`, `Set`, `SortedSet`, `NavigableSet`, `Queue`, `Deque`, `Map`, `SortedMap`, and `NavigableMap`. `Collection` extends `Iterable`; `List`, `Set`, and `Queue` each extend `Collection`; `SortedSet` extends `Set`; `NavigableSet` extends `SortedSet`; `Deque` extends `Queue`; `SortedMap` extends `Map`; and `NavigableMap` extends `SortedMap`.

Figure 9-1 illustrates the core interfaces hierarchy (arrows point to parent interfaces).

![Core Interfaces Hierarchy](image)

**Figure 9-1. The Collections Framework is based on a hierarchy of core interfaces**

The framework’s implementation classes include `ArrayList`, `LinkedList`, `TreeSet`, `HashSet`, `LinkedHashSet`, `EnumSet`, `PriorityQueue`, `ArrayDeque`, `TreeMap`, `HashMap`, `LinkedHashMap`, `IdentityHashMap`, `WeakHashMap`, and `EnumMap`. The name of each concrete class ends in a core interface name, identifying the core interface on which it is based.

**Note** Additional implementation classes are part of the concurrency utilities.

The framework’s implementation classes also include the abstract `AbstractCollection`, `AbstractList`, `AbstractSequentialList`, `AbstractSet`, `AbstractQueue`, and `AbstractMap` classes. These classes offer skeletal implementations of the core interfaces to facilitate the creation of concrete implementation classes.

Finally, the framework provides two utility classes: `Arrays` and `Collections`.

**Comparable Versus Comparator**

A collection implementation stores its elements in some *order* (arrangement). This order may be unsorted, or it may be sorted according to some criterion (such as alphabetical, numerical, or chronological).

A sorted collection implementation defaults to storing its elements according to their *natural ordering*. For example, the natural ordering of `java.lang.String` objects is *lexicographic* or *dictionary* (also known as alphabetical) order.