* The set of reserved Java programming language keywords {'import', 'class', 'public', 'protected'...}
* A set of people (friends, employees, clients, ...)
* The set of records returned by a database query
* The set of `Component` objects in a `Container`
* The set of all pairs
* The empty set {}

Sets have the following basic properties:

* They contains only one instance of each item
* They may be finite or infinite
* They can define abstract concepts

Sets are fundamental to logic, mathematics, and computer science, but also practical in everyday applications in business and systems. The idea of a "connection pool" is a set of open connections to a database server. Web servers have to manage sets of clients and connections. File descriptors provide another example of a set in the operating system.

A *map* is a special kind of set. It is a set of pairs, each pair representing a one-directional mapping from one element to another. Some examples of maps are:

* The map of IP addresses to domain names (DNS)
* A map from keys to database records
* A dictionary (words mapped to meanings)
* The conversion from base 2 to base 10

Like sets, the idea behind a map is much older than the Java programming language, older even than computer science. Sets and maps are important tools in mathematics and their properties are well understood. People also long recognized the usefulness of solving programming problems with sets and maps. A language called SETL (Set Language) invented in 1969 included sets as one of its only primitive data types (SETL also included garbage collection -- not widely accepted until Java technology was developed in the 1990s). Although sets and maps appear in many languages including C++, the Collections Framework is perhaps the best designed set and map package yet written for a popular language. (Users of C++ Standard Template Library (STL) and Smalltalk's collection hierarchy might argue that last point.)

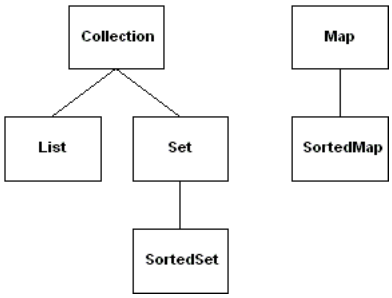
Also because they are sets, maps can be finite or infinite. An example of an infinite map is the conversion from base 2 to base 10. Unfortunately, the Collections Framework does not support infinite maps -- sometimes a mathematical function, formula, or algorithm is preferred. But when a problem can be solved with a finite map, the Collections Framework provides the Java programmer with a useful API.

Because the Collections Framework has formal definitions for the classes `Set`, , and `Map`, you'll notice the lower case words *set*, *collection*, and *map* to distinguish the implementation from the concept.

Section 3. Collection interfaces and classes

Introduction

Now that you have some set theory under your belt, you should be able to understand the Collections Framework more easily. The Collections Framework is made up of a set of interfaces for working with groups of objects. The different interfaces describe the different types of groups. For the most part, once you understand the interfaces, you understand the framework. While you always need to create specific implementations of the interfaces, access to the actual collection should be restricted to the use of the interface methods, thus allowing you to change the underlying data structure, without altering the rest of your code. The following diagrams shows the framework interface hierarchy.



One might think that `Map` would extend `Collection`. In mathematics, a map is just a collection of pairs. In the Collections Framework, however, the interfaces `Map` and `Collection` are distinct with no lineage in the hierarchy. The reasons for this distinction have to do with the ways that `Set` and `Map` are used in the Java libraries. The typical application of a `Map` is to provide access to values stored by keys. The set of collection operations are all there, but you work with a key-value pair instead of an isolated element. `Map` is therefore designed to support the basic operations of *`get()`* and *`put()`* , which are not required by `Set`. Moreover, there are methods that return `Set` views of `Map` objects:

```
Set set = aMap.keySet();
```

When designing software with the Collections Framework, it is useful to remember the following hierarchical relationships of the four basic interfaces of the framework:

* The `Collection` interface is a group of objects, with duplicates allowed.
* The `Set` interface extends `Collection` but forbids duplicates.
* The `List` interface extends `Collection`, allows duplicates, and introduces positional indexing.
* The `Map` interface extends neither `Set` nor `Collection`.

Moving on to the framework implementations, the concrete collection classes follow a naming convention, combining the underlying data structure with the framework interface. The following table shows the six collection implementations introduced with the Java 2 framework, in addition to the four historical collection classes. For information on how the historical collection classes changed, like how `Hashtable` was reworked into the framework, see the Historical collection classes on page 25.

Interface	Implementation	Historical
`Set`	`HashSet`	

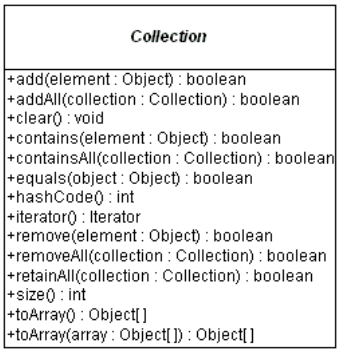
	TreeSet	
List	ArrayList	Vector
	LinkedList	Stack
Map	HashMap	Hashtable
	TreeMap	Properties

There are no implementations of the `Collection` interface. The historical collection classes are called such because they have been around since the 1.0 release of the Java class libraries.

If you are moving from the historical collection classes to the new framework classes, one of the primary differences is that all operations are unsynchronized with the new classes. While you can add synchronization to the new classes, you cannot remove it from the old.

Collection interface

The `Collection` interface is used to represent any group of objects, or elements. You use the interface when you wish to work with a group of elements in as general a manner as possible. Here is a list of the public methods of `Collection` in Unified Modeling Language (UML) notation.



The interface supports basic operations like adding and removing. When you try to remove an element, only a single instance of the element in the collection is removed, if present.

```
*   boolean add(Object element)
*   boolean remove(Object element)
```

The `Collection` interface also supports query operations:

```
*   int size()
*   boolean isEmpty()
*   boolean contains(Object element)
*   Iterator iterator()
```

Iterator interface

The *`iterator()`* method of the `Collection` interface returns an `Iterator`. An `Iterator` is similar to the `Enumeration` interface, which you may already be familiar with,