OBJECT ORIENTED ANALYSIS & DESIGN
DATA STRUCTURES & ALGORITHMS

Collections (util)

Objectives

- Explore a wide variety of utilities provided by utility and text packages of the J2SDK
- Learn the architecture of the Java collections framework
- Select appropriate collection classes and interfaces for specific behavioral requirements
- Create, build, and traverse collections using the provided collection classes
- Define your own collections that conform to the collections framework
- Generate random numbers

The Utility Packages

- In the first version of Java, the java.util package contained general purpose utility classes and interfaces
- Utility packages are in the java.util package and the packages inside it
- The java.util package contains the collections framework and a number of utility types not related to collections

The Utility Packages

Package name	Since	Description
`java.util`	1.0	Contains the collections framework and a number of utility types not related to collections; Figure 4.2 shows noncollection types.
`java.util.jar`	1.2	Contains classes to read and write Java Archive (jar) files, including maintaining the manifest file for the jar.
`java.util.logging`	1.4	Contains types used to format and write logging messages that monitor activities of programs. You typically use logs as diagnostic aids when maintaining or servicing programs deployed to a production environment.
`java.util.prefs`	1.4	Contains types for storing and accessing user or system preferences in an implantation-dependent persistent storage media. For example, user preferences could include the initial location and size of the window that holds a program's graphical user interface, and a system preference could be the location of files.
`java.util.regex`	1.4	Contains classes for matching character sequences against patterns presented as regular expressions. Using these classes and regular expressions is discussed in the latter half of this chapter.
`java.util.zip`	1.1	Contains types for reading and writing files in standard zip and gzip formats, compressing and decompressing data using the algorithm used in zip and gzip files, and calculating CRC-32 and Adler-32 checksums for streams of characters.
`java.text`	1.1	Contains types for handling text, dates, times, and numbers including monetary values and messages. The classes are locale sensitive so you can use them to format output according to the user's cultural environment. Some examples are supplied later in this chapter.

The Collections Framework

The collections framework consists of:

- Interfaces that define the behavior of collections
- Concrete classes that provide general-purpose implementations of the interfaces that you can use directly
- Abstract classes that implement the interfaces of the collections framework that you can extend to create collections for specialized data structures

The Collections Framework

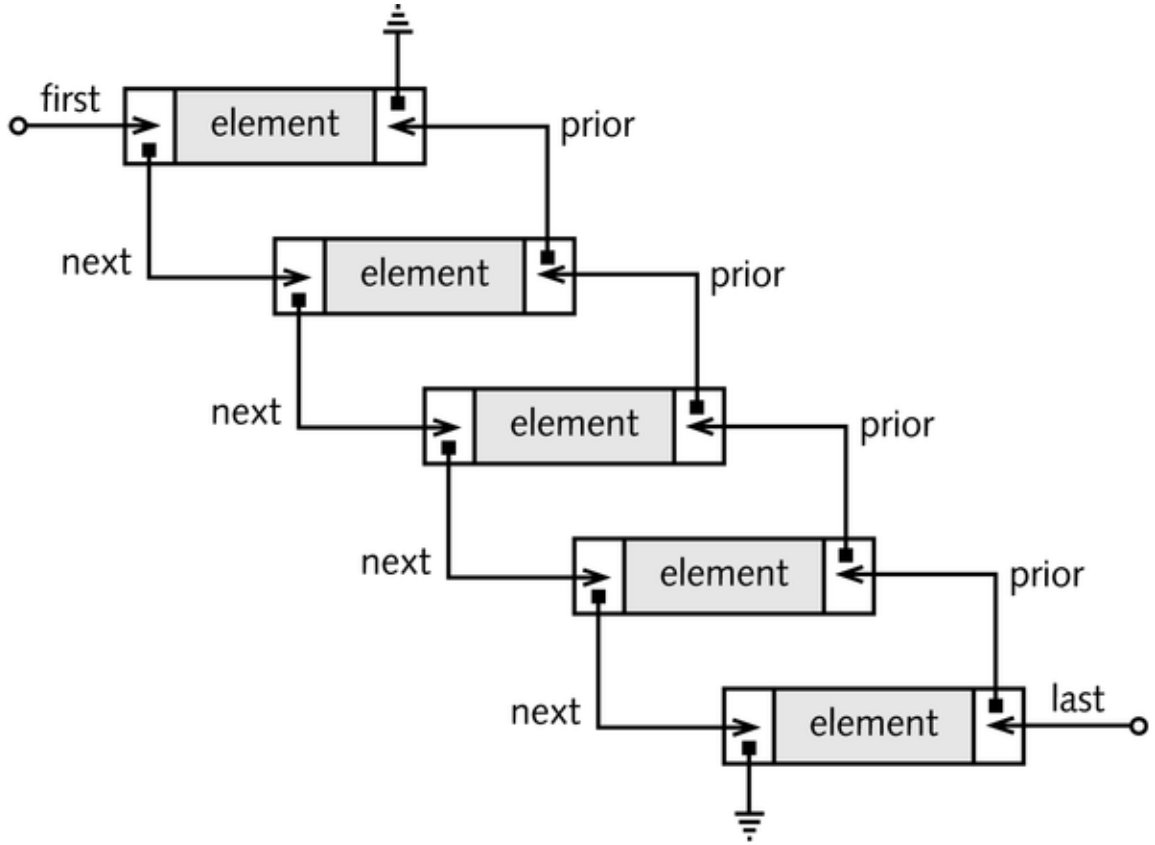
The goals of the Collections framework are to:

- Reduce the programming effort of developers by providing the most common data structures
- Provide a set of types that are easy to use, extend, and understand
- Increase flexibility by defining a standard set of interfaces for collections to implement
- Improve program quality through the reuse of tested software components

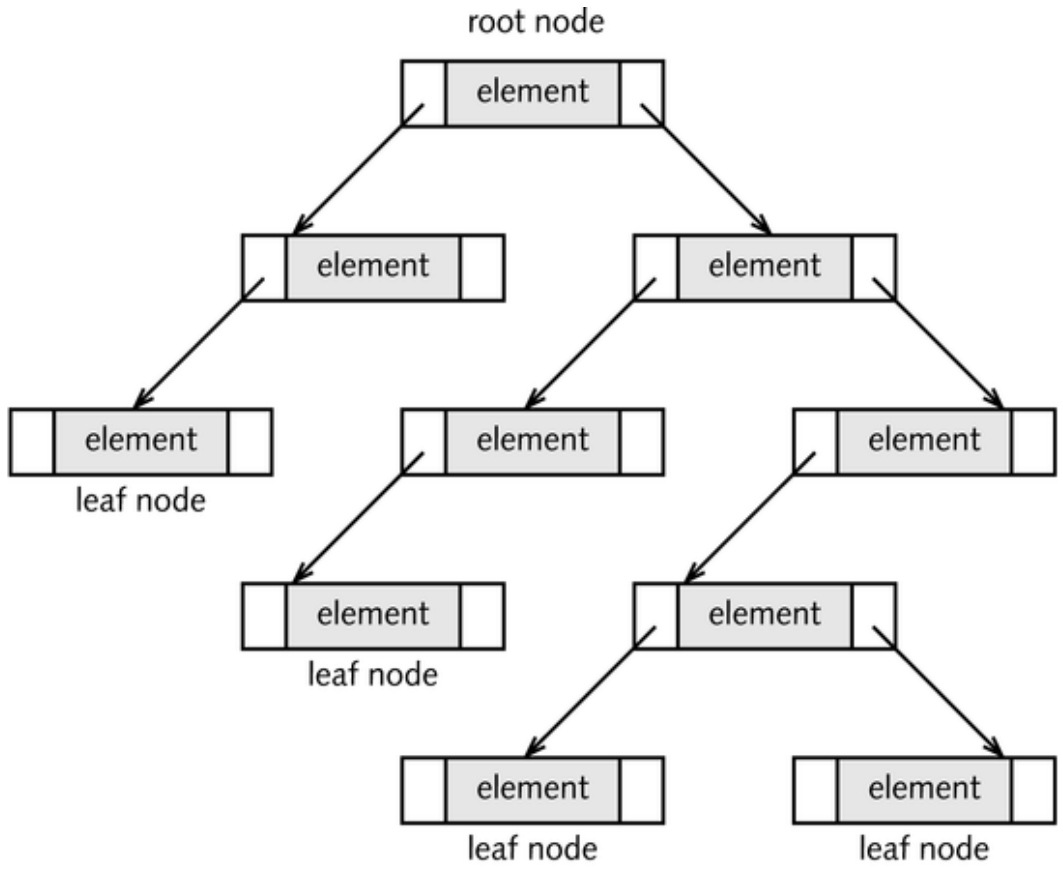
The Collections Framework

- **Bag:** a group of elements
- **Iterator:** a cursor, a helper object that clients of the collections use to access elements one at a time
- **List:** a group of elements that are accessed in order
- **Set:** a group of unique elements
- **Tree:** a group of elements with a hierarchical structure

A Doubly Linked List



A Binary Tree



Example Tree class

```
class tree{
 String val;
 tree left=null;
 tree right=null;
 tree (String s){ val=s;}// constructor
 static tree ins(tree t,String s){
        if (t==null) return new tree(s);
        if( t.val.compareTo(s)<0)
                    t.right=ins(t.right,s);
        if(t.val.compareTo(s)>=0)
                    t.left=ins(t.left,s);
        return t;
  }
```

More of tree class

```
  void ins(String s){ ins(this,s);}
  void visit(Visitor v){
        v.act(val);
        if(left!=null) left.visit(v);
        if(right!=null) right.visit(v);
  }
  class printer implements visitor{
   void act(Object o){System.out.println(o.toString());}
   }
  void printTree() { visit(new Printer());}
}
interface visitor{ void act(Object o) ;}
```

The Collections Framework

- **The root interface of the framework is Collection**

- **The Set interface:**
  - ✔ Extends the Collection interface
  - ✔ Defines standard behavior for a collection that does not allow duplicate elements

- **The List interface:**
  - ✔ Extends the Collection interface
  - ✔ Defines the standard behavior for ordered collections (sequences)

The Collection Interface

Methods:

- boolean add(Object element)
- boolean addAll(Collection c)
- void clear()
- boolean contains(Object element)
- boolean containsAll(Collection c)
- boolean equals(Object o)
- int hashCode()
- boolean isEmpty()

The Collection Interface (Cont.)

Methods:

- Iterator iterator()
- boolean remove(Object element)
- boolean removeAll(Collection c)
- boolean retainAll(Collection c)
- int size()
- Object[] toArray()
- Object[] toArray(Object[] a)

The Set Interface

Methods:

- **The boolean add(Object element) method:**
  - ✔ Ensures collection contains the specified element
  - ✔ Returns false if element is already part of the set
- **The boolean addAll(Collection c) method:**
  - ✔ Adds all the elements in the specified collection to this collection
  - ✔ Returns false if all the elements in the specified collection are already part of the set

The List Interface

Methods:

- boolean add(Object element)
- void add(int index, Object element)
- boolean addAll(Collection c)
- boolean addAll(int index, Collection c)
- Object get(int index)
- int indexOf(Object element)
- int lastIndexOf(Object element)

Note that the elements of the list have numbered positions – their indices

The List Interface (Cont.)

Methods:

- ListIterator listIterator()
- ListIterator listIterator(int index)
- boolean remove(Object element)
- Object remove(int index)
- Object set(int index, Object element)
- List subList(int beginIndex, int endIndex)

Traversing Collections with Iterators

Iterator interface methods:

- boolean hasNext()
- Object next()
- void remove()

```
Iterator i = s.iterator();
 while (i.hasNext()){
   Object n = i.next();
   … do something with n
}
```

Traversing Collections with Iterators

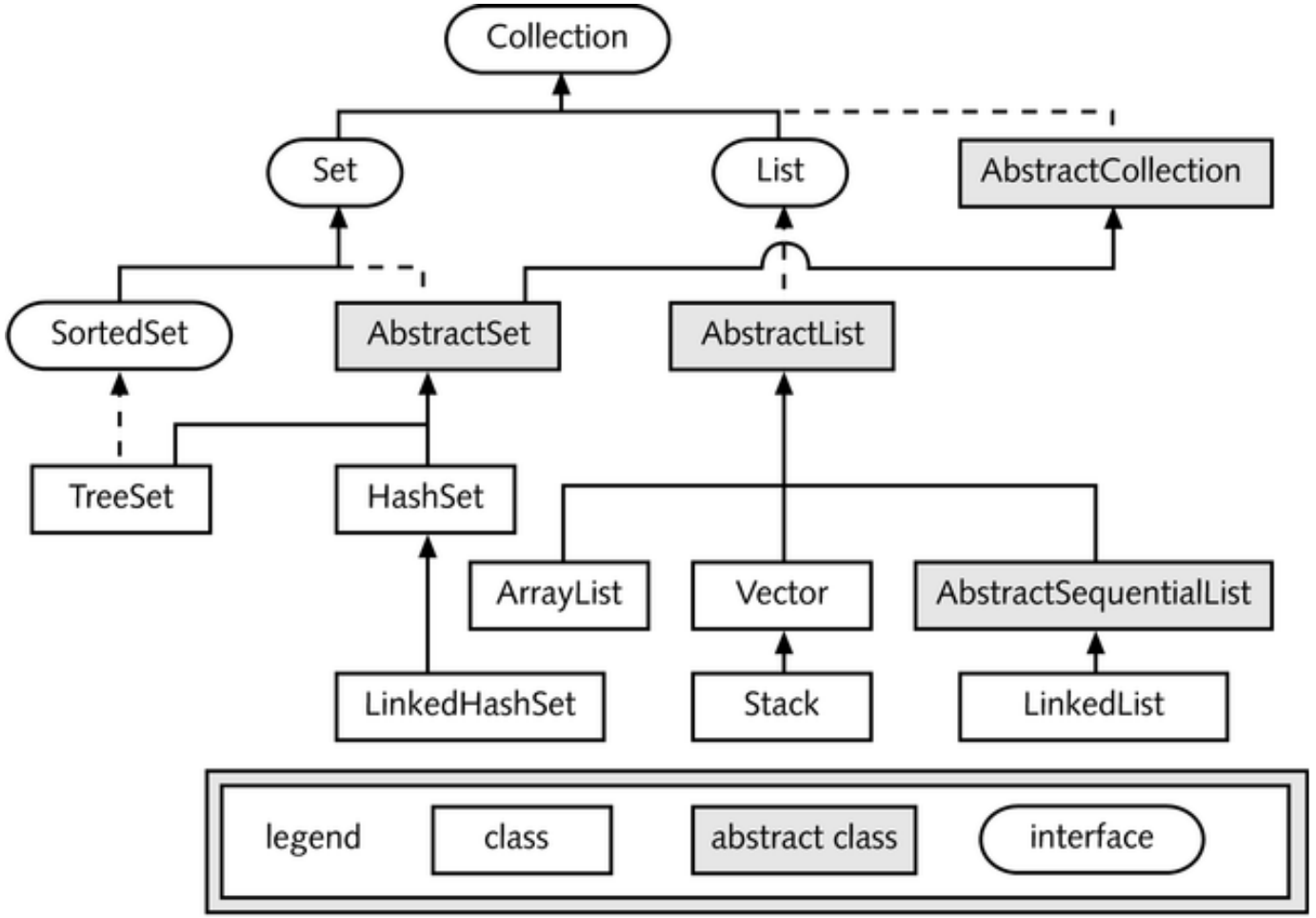
ListIterator interface methods:

- void add(Object element)

  The element is inserted immediately before the next element that would be returned by next,

- boolean hasPrevious()
- int nextIndex()
- Object previous()
- int previousIndex()
- void set(Object element)

General Purpose Implementations



General Purpose Sets

Three framework classes implement the Set interface:

- HashSet
- TreeSet
- LinkedHashSet

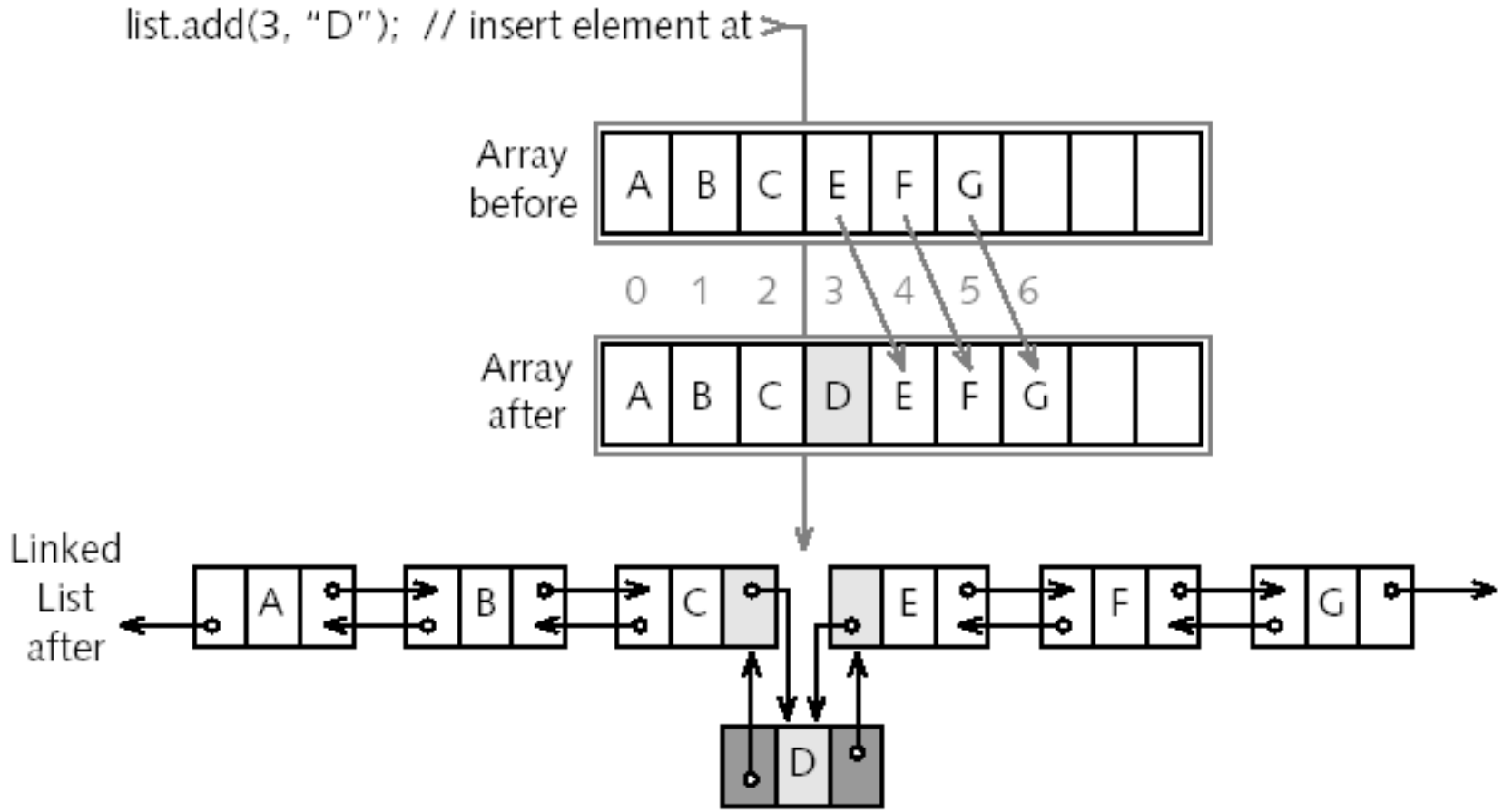
Note that it is possible to add elements of different classes to the same set as the following example illustrates.

```
package examples.collections;
import java.util.*;
/** A class to demonstrate the use of the Set
  * interface in the java.util package
  */
public class HashSetExample {
   /** Test method for the class
     * @param args not used
     */
   public static void main( String[] args ) {
      // create a set and initialize it
      Set s1 = new HashSet();
      s1.add( new Integer( 6 ) );
      s1.add(  "Hello" );
      s1.add( new Double( -3.423 ) );
      s1.add( new java.util.Date( ) );
      // iterate to display the set values
      Iterator i1 = s1.iterator();
      while ( i1.hasNext() ) {
           System.out.print( i1.next() + " " );
      }
   }
}
```

General Purpose Lists

Four concrete classes in the framework are implementations of the List interface:

- ArrayList
- LinkedList
- Vector
- Stack


list.add(3, "D"); // insert element at
Array before
A B C E F G
0 1 2 3 4 5 6
Array after
A B C D E F G
Linked List after
A
B
C
D
E
F
G

Array lists vs linked lists

- Array lists are more compact and thus use less memory and may impose less of a garbage collection overhead.

- Linked lists are more efficient however, when insertions and deletions are common, as they do not require shifting of existing elements.

Arrays as Collections

toArray: converts a Collection object into an array

java.util.Arrays: provides static methods that operate on array objects

Arrays class: useful when integrating your programs with APIs that

- Require array arguments
- Return arrays

Sorted Collections

SortedSet adds methods to the Set interface:

- Comparator comparator()
- Object first()
- SortedSet headSet(Object element)
- Object last()
- SortedSet subSet(int beginElement, int endElement)
- SortedSet tailSet(Object element)

Maps

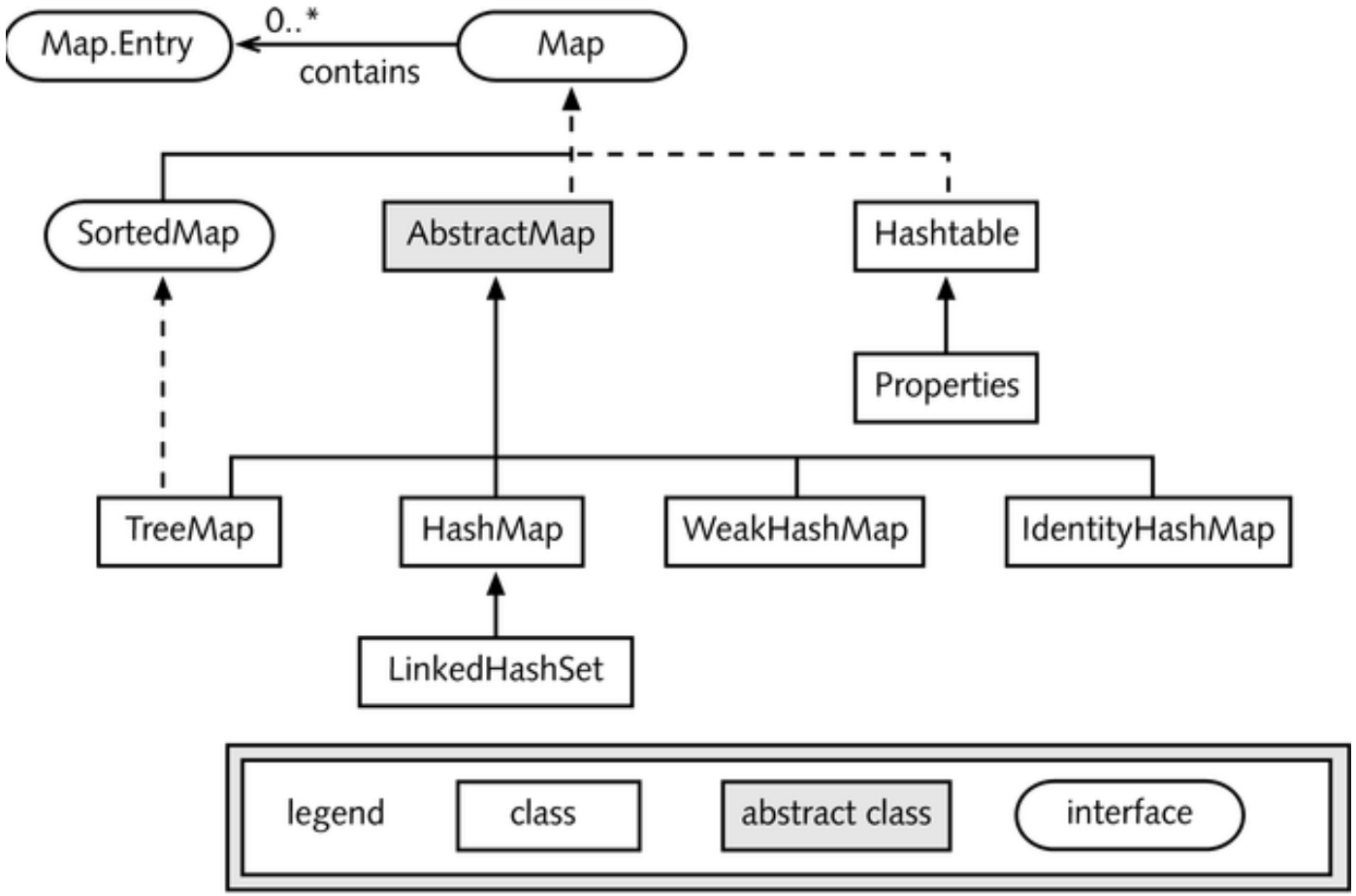
- A map is an abstraction for an aggregation of key-value, or name-value, pairs
- Two interfaces in the collections framework define the behavior of maps: Map and SortedMap
- A third interface, Map.Entry, defines the behavior of elements extracted from Map

Maps (Contd.)

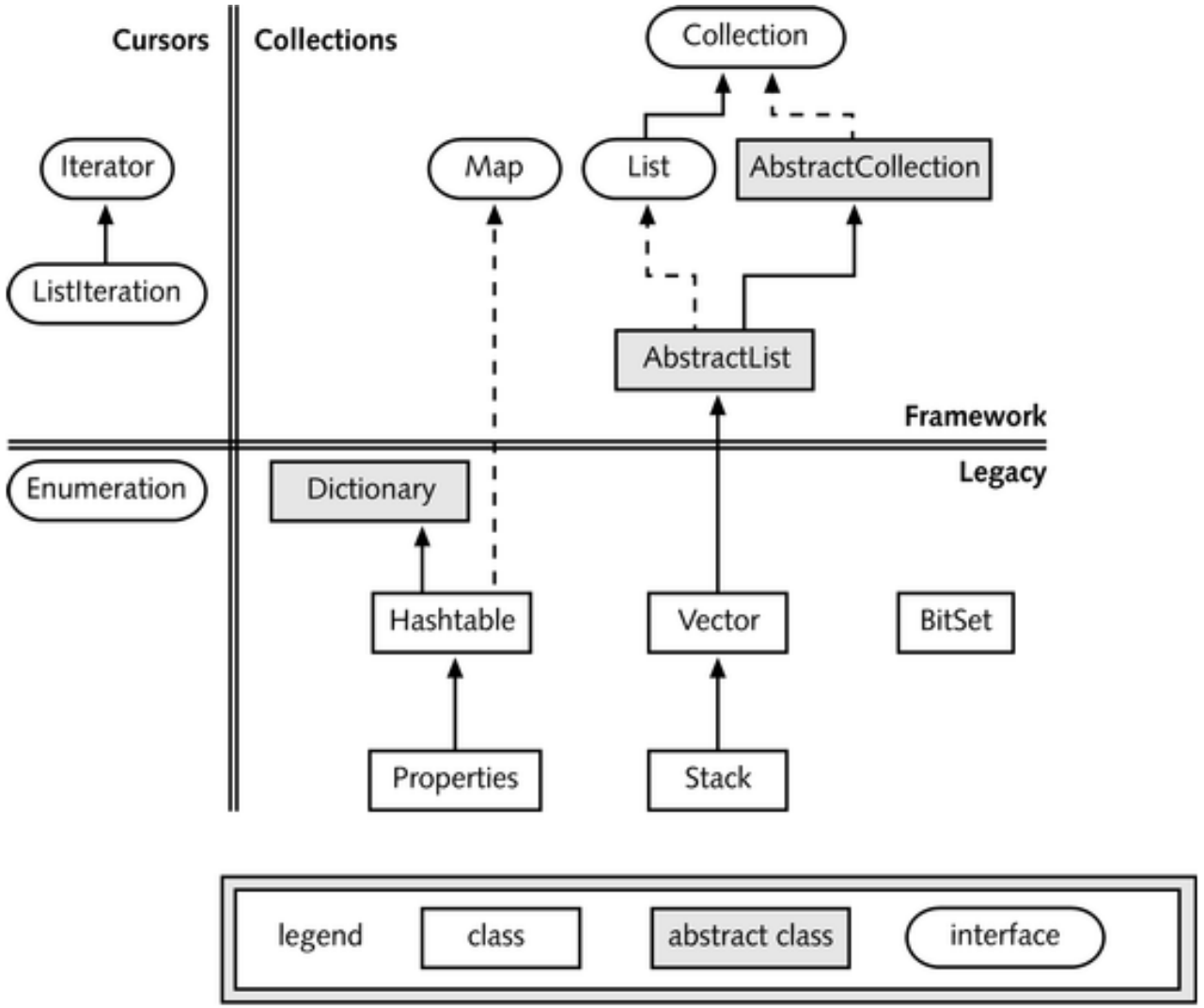
Seven concrete framework classes implement the Map interface:

- HashMap
- IdentityHashMap
- LinkedHashMap
- TreeMap
- WeakHashMap
- Hashtable
- Properties

The Map Types



Legacy Classes and Collections



Legacy Collection Classes

Legacy Class	Purpose
`BitSet`	Use `BitSet` collections to contain sets of bits or true-false flags. The collection is dynamically sized, so you can add elements without worrying about exceeding limits. Simply setting or clearing a value at an index value that is beyond the current size extends the set. When you extend a `BitSet` object, all the added elements are given the default value of `false`. A `BitSet` collection never shrinks.
`Dictionary`	This abstract class defines methods for storing elements of type `Object`, according to key values that are also of type `Object`. `Dictionary` is the superclass of the `Hashtable` class, but is otherwise not available for use. For new collections, implement `Map` and do not extend `Dictionary`.
`Hashtable`	Instances of the `Hashtable` class are hash tables.
`Properties`	The `Properties` class extends `Hashtable` to a collection of key-value pairs, where each key and each value is a `String` object. The key for each item in the `Properties` table is the name of a property.
`Stack`	The `Stack` class has methods for adding and removing objects according to the last-in first-out rule. The `Stack` class extends `Vector` and adds methods for pushing, popping, and peeking into the stack.
`Vector`	An instance of `Vector` is an indexed list of objects, much like an array. Use a `Vector` when you need greater flexibility than arrays provide. The main advantage is that a `Vector` collection can grow and shrink in size as required, but an array has a fixed size. The `Vector` class also has several methods that are not available for arrays. Because `Vector` objects are ordered, they are more efficient than `Hashtable` objects for enumeration purposes.

java.util.Enumeration

The java.util.Enumeration interface defines the methods you can use to traverse the objects in a collection of type Vector, Stack, Hashtable, or Properties

Methods:

- boolean hasMoreElements()
- Object nextElement()

- A BitSet object contains a number of bits, each of which represents a true or false value
- You can use a Hashtable collection for key-value pairs
- The Properties class extends Hashtable and suitable for writing to or reading from I/O streams
- The Vector class supports a dynamically resizable list of object references
- The Stack class provides a collection with first-in last-out or last-in first-out behavior

WeakHashMap

- This is a hashtable-based Map implementation with weak keys.
- An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, so that if the garbage collector gets rid of it for other reasons, then the key can be deleted from the table.
- This prevents objects remaining on the heap just because they are in the hash table.

Generating Random Numbers

- A pseudo-random number generator produces a sequence of values, one at a time, so that resulting data can pass statistical tests of randomness
- The java.util.Random class generates pseudo-random numbers or data of a variety of types
- The random method in the java.lang.Math class generates uniformly distributed pseudo-random double values in the range 0.0 to 1.0
- A random number is one for which the shortest generating program is longer than the number itself. If the program is shorter then the number is pseudo random

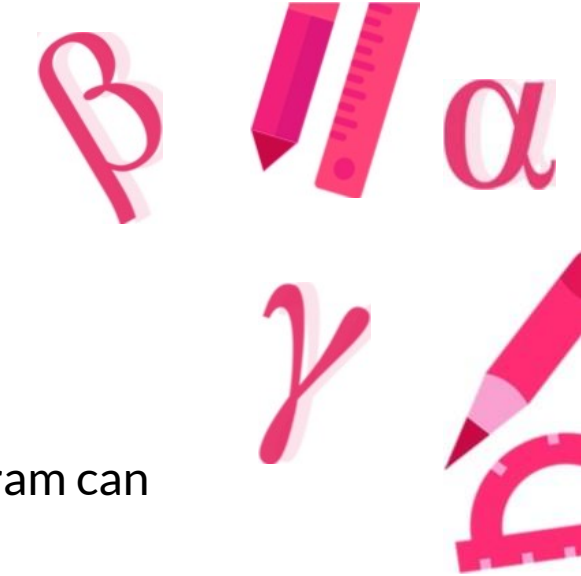
Class Random

```
Random r = new Random(109876L); // seed
int i = r.nextInt();
int j = r.nextInt();
 long l = r.nextLong();
 float f = r.nextFloat();
double d = r.nextDouble();
 double k = r.nextGaussian();
```

The nextGaussian() method returns a pseudo-random, Gaussian distributed, double value with mean 0.0 and standard deviation 1.0.

- A locale stores settings about a language or country—including what alphabet is used, how dates and numbers are written, and other culture-specific aspects of information processing

- Dates, numbers, and monetary values are formatted by default according to the default locale for the implementation of Java



- A resource is a single entity, such as a text message, that your program can access and use
- Use the ResourceBundle class to collect resources into one manageable object
- The resources can reside in a separate file called a properties file or in a class definition created for the purpose of holding resources

A Sample Properties File

```
# RegistrationForm.properties
# This file contains the strings used in the
# Registration Example when run for default
# English US locales.
instructions=please complete the following form
contact.name=Name
address.street=Street
address.city=City
address.region=State
address.code=Zip Code
contact.phone=telephone
# thank-you=Thank-you
messages.missing resource=Could not find resource “{0}”.
```

Using Resource Bundles

The java.util.ResourceBundle abstract class encapsulates resources that are loaded from a property file or a class that extends ListResourceBundle

The getBundle method

- Locates a .class file or a .properties file
- Parses the file
- Creates a ResourceBundle object containing the resource information
- myResourceBundle.getString(“address.street")

Summary

- The collections framework defines interfaces that define the behavior of a variety of data structures: Collection, List, Set, SortedSet, Map, and SortedMap
- Each collection class has an iterator method that provides an Iterator object to traverse the collection one element at a time
- Legacy collection classes Hashtable, Properties, Vector, and Stack (except Bitset) have been retrofitted into the collections framework; they retain old methods and cursor objects of type Enumeration

Summary (Cont.)

- Classes that implement the interface Observer are notified when an object of a class that extends the Observable class changes
- The Random class generates pseudo-random numbers
- Locales are data structures that encapsulate cultural environments
- The StringTokenizer class gives you limited ability to parse strings
- The classes Pattern and Matcher in java.util.regex provide flexible pattern recognition in character sequences using regular expressions

Thank You!