Module 9

Collections and Generics Framework

Objectives

Upon completion of this module, you should be able to:

- Describe the general purpose implementations of the core interfaces in the Collections framework
- Examine the `Map` interface
- Examine the legacy collection classes
- Create natural and custom ordering by implementing the `Comparable` and `Comparator` interfaces
- Use generic collections
- Use type parameters in generic classes
- Refactor existing non-generic code
- Write a program to iterate over a collection
- Examine the enhanced `for` loop

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.

The Collections API

A *collection* is a single object managing a group of objects. The objects in the collection are called *elements*. Typically, collections deal with many types of objects, all of which are of a particular kind (that is, they all descend from a common parent type).

The Collections API contains interfaces that group objects as one of the following:

- `Collection` – A group of objects known as elements; implementations determine whether there is specific ordering and whether duplicates are permitted
- `Set` – An unordered collection; no duplicates are permitted
- `List` – An ordered collection; duplicates are permitted

Before the release of the Java SE 5.0 platform, collections maintained references to objects of type `Object`. This enables any object to be stored in the collection. It also necessitates the use of correct casting before you can use the object, after retrieving it from the collection. However, with the Java SE 5.0 platform and onwards, you can use generic collection features to specify the object type to be stored in a collection. This avoids the need to cast the object on retrieval. Generic collections are discussed in more detail in “Generics” on page 9-17.

Figure 9-1 shows a UML diagram of the primary interfaces and implementation classes of the Collections API.

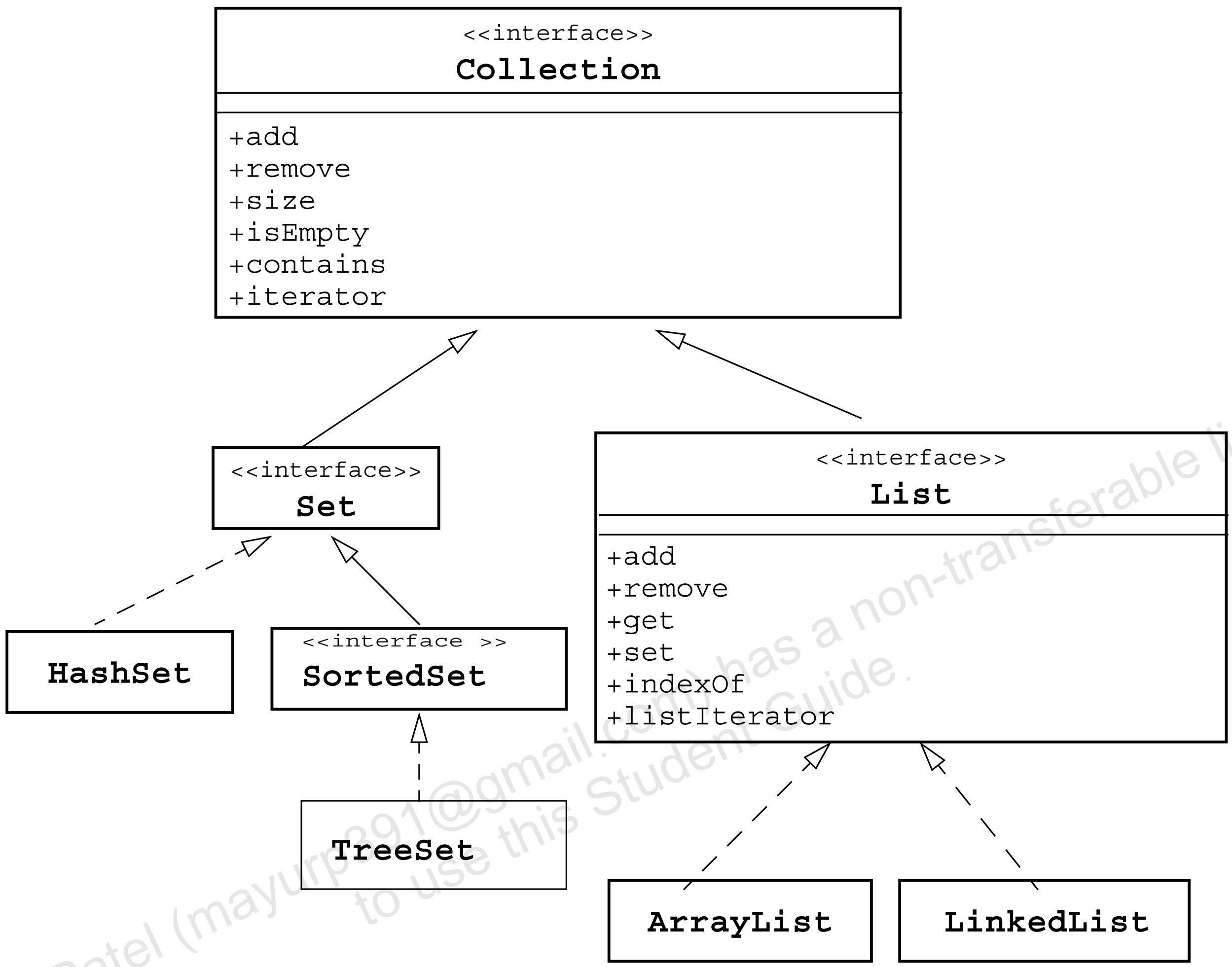

Figure 9-1 The Collection Interface and Class Hierarchy

The `HashSet` is one example of a class that supplies an implementation of the `Set` interface. The `SortedSet` interface extends the `Set` interface. The classes that implement `SortedSet` impose total ordering on its elements. `TreeSet` implements the `SortedSet` interface.The `ArrayList` and `LinkedList` classes supply an implementation of the `List` interface.

Note – This discussion of the Collections API is a simplification of the complete API (which includes many more methods, more interfaces, and several intermediate abstract classes). For more information, read *Introduction to the Collections Framework* at the following URL:
`http://developer.java.sun.com/developer/onlineTraining/collections/`

Collection Implementations

There are several general purpose implementations of the core interfaces (`Set`, `List`, `Map` and `Deque`) in the Collections framework. Table 9-1 shows some of the concrete implementations of these interfaces.

Table 9-1 General Purpose Collection Implementations

	Hash Table	**Resizable Array**	**Balanced Tree**	**Linked List**	**Hash Table + Linked List**
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

The following subsections show examples of the use of `HashSet` and `ArrayList`.

A `Set` Example

In the following example shown in Code 9-1, the program declares a variable (`set`) of type `Set` that is initialized to a new `HashSet` object. It then adds a few elements and prints the `set` to standard output. Lines 10 and 11 in Code 9-1 attempt to add duplicate values to `set`. Because duplicate values cannot be added to a `Set`, the `add` methods return `false`.

Code 9-1 `SetExample` Program

```
1   import java.util.*;
2   public class SetExample {
3     public static void main(String[] args) {
4      Set set = new HashSet();
5      set.add("one");
6      set.add("second");
7      set.add("3rd");
8      set.add(new Integer(4));
9      set.add(new Float(5.0F));
10     set.add("second");        // duplicate, not added
11     set.add(new Integer(4)); // duplicate, not added
12     System.out.println(set);
13    }
14  }
```

The output generated from this program might be:

```
[one, second, 5.0, 3rd, 4]
```

You should note that the order of the elements is not the same as the order in which they were added.

Note – In Line 13, the program prints the `set` object to standard output. This works because the `HashSet` class overrides the inherited `toString` method and creates a comma-separated sequence of the items delimited by the open and close braces.

A `List` Example

In the following example shown in Code 9-2, the program declares a variable (`list`) of type `List` that is assigned to a new `ArrayList` object. It then adds a few elements and prints the list to standard output. Because lists allow duplicates, the `add` methods in lines 10 and 11 in Code 9-2 return `true`.

Code 9-2 `ListExample` Program

```
1   import java.util.*;
2   public class ListExample {
3     public static void main(String[] args) {
4      List list = new ArrayList();
5      list.add("one");
6      list.add("second");
7      list.add("3rd");
8      list.add(new Integer(4));
9      list.add(new Float(5.0F));
10     list.add("second");        // duplicate, is added
11     list.add(new Integer(4));  // duplicate, is added
12     System.out.println(list);
13    }
14  }
```

The output generated from this program is:

```
[one, second, 3rd, 4, 5.0, second, 4]
```

The order of the elements is the order in which they were added.

Note – The Collections API contains many useful, concrete implementations of the `Collection`, `Set`, and `List` interfaces. You are encouraged to check the API documentation and become familiar with the implementations. Some even implement hybrid behavior, such as the `LinkedHashMap`, which uses hashing to implement fast searches, and also maintains a doubly linked list internally so that it can return objects from an iterator in a meaningful order.

The Map Interface

Maps are sometimes called associative arrays. A Map object describes mappings from keys to values. By definition, a Map object does not allow duplicate keys and a key can map to one value at most.

The Map interface provides three methods that allow map contents to be viewed as collections

- entrySet – Returns a Set that contains all the key value pairs.
- keySet – Returns a Set of all the keys in the map.
- values – Returns a Collection containing all the values contained in the map.

Figure 9-2 shows the API for the Map interface, its sub-interfaces and a few of the more well known implementing classes.

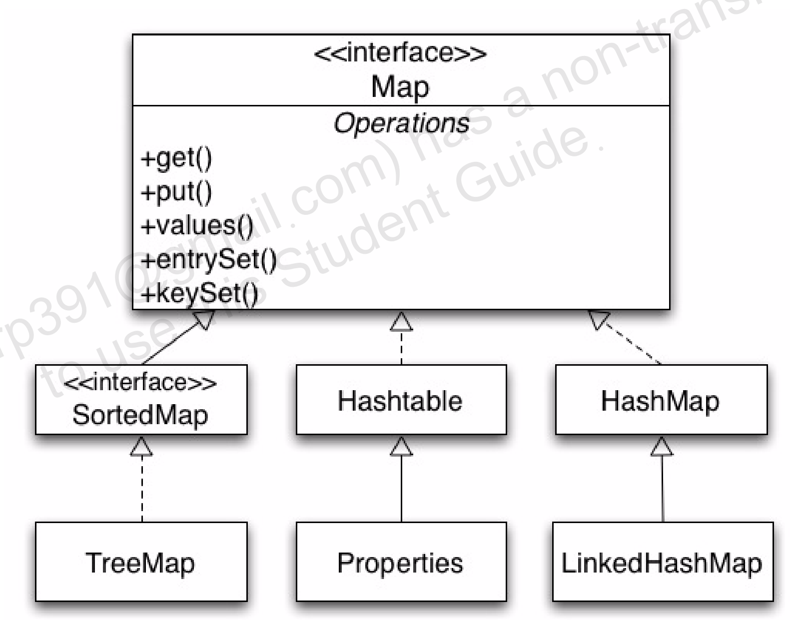

Figure 9-2 The Map Interlace API

The Map interface does not extend the Collection interface because it represents mappings and not a collection of objects. The SortedMap interface extends the Map interface. Some of the classes that implement the Map interface are HashMap, TreeMap, IdentityHashMap, and WeakHashMap. The order presented by the iterators of these map collection implementations is specific to each iterator.

A Map example

In the example shown in Code 9-3, the program declares a variable `map` of type `Map`, and assigns it to a new `HashMap` object. It then adds a few elements to the map by using the `put` operation. To prove that duplicate keys are not allowed in a map, the program attempts to add a new value using an existing key. This results in the previously added value for the key being replaced by the new value. The program later uses the collection view operations `keySet`, `values`, and `entrySet` for retrieving the contents of the map.

Code 9-3 `MapExample` Program

```
import java.util.*;
public class MapExample {
   public static void main(String args[]) {
      Map map = new HashMap();
      map.put("one","1st");
      map.put("second", new Integer(2));
      map.put("third","3rd");
      // Overwrites the previous assignment
      map.put("third","III");
      // Returns set view of keys
      Set set1 = map.keySet();
      // Returns Collection view of values
      Collection collection = map.values();
      // Returns set view of key value mappings
      Set set2 = map.entrySet();
       System.out.println(set1 + "\n" + collection + "\n" + set2);
   }
}
```

The generated output from the program is:

```
[second, one, third]
[2, 1st, III]
[second=2, one=1st, third=III]
```

Legacy Collection Classes

The collection classes in the JDK version 1.0 and JDK version 1.1 still exist in the current JDK with the same interface, but they have been retooled to interact with the new Collections API.

- The `Vector` class implements the `List` interface.
- The `Stack` class is an extension of `Vector` that adds the typical stack operations: `push`, `pop`, and `peek`.
- The `Hashtable` is an implementation of `Map`.
- The `Properties` class is an extension of `Hashtable` that only uses `Strings` for keys and values.
- Each of these collections has an `elements` method that returns an `Enumeration` object. The `Enumeration` is an interface similar to, but incompatible with, the `Iterator` interface. For example, `hasNext` is replaced by `hasMoreElements` in the `Enumeration` interface.

Ordering Collections

The `Comparable` and `Comparator` interfaces are useful for ordering collections. The `Comparable` interface imparts natural ordering to classes that implement it. The `Comparator` interface is used to specify order relation. It can also be used to override natural ordering. These interfaces are useful for sorting the elements in a collection.

The `Comparable` Interface

The `Comparable` interface is a member of the `java.lang` package. When you declare a class, the JVM implementation has no means of determining the order you intend for objects of that class. You can, by implementing the `Comparable` interface, provide order to the objects of any class. You can sort collections that contain objects of classes that implement the `Comparable` interface.

Examples of some Java classes that implement the `Comparable` interface, are `Byte`, `Long`, `String`, `Date`, and `Float`. Of these, the numeric classes use a numeric implementation; the `String` class uses an alphabetic implementation and the `Date` class uses a chronological implementation. Passing an `ArrayList` containing `String` type elements to the static `sort` method of the `Collections` class returns a list sorted in alphabetical order. A list containing `Date` type elements sorts in chronological order and a list containing `Integer` type elements sorts in numerical order.

To write custom `Comparable` types, you need to implement the `compareTo` method of the `Comparable` interface. Code 9-4 shows how to implement the `Comparable` interface. The `Student` class implements the `Comparable` interface so that the objects of this class can be compared to each other based on grade point average (GPA).

Code 9-4 Example `Comparable` Interface Implementation

```
class Student implements Comparable {
  String firstName, lastName;
  int studentID=0;
  double GPA=0.0;
  public Student(String firstName, String lastName, int studentID,
    double GPA) {
    if (firstName == null || lastName == null || studentID == 0
      || GPA == 0.0) {throw new IllegalArgumentException();}
    this.firstName = firstName;
    this.lastName = lastName;
```

```
    this.studentID = studentID;
     this.GPA = GPA;
  }
  public String firstName() { return firstName; }
  public String lastName()  { return lastName;  }
  public int studentID() { return studentID; }
  public double GPA() { return GPA; }
  // Implement compareTo method.
  public int compareTo(Object o) {
    double f = GPA-((Student)o).GPA;
    if (f == 0.0)
      return 0;    // 0 signifies equals
    else if (f<0.0)
      return -1;   // negative value signifies less than or before
    else
      return 1;    // positive value signifies more than or after
  }
}
```

The `StudentList` program in Code 9-5 tests the `Comparable` interface implementation created in the Code 9-4 on page 9-11. The `StudentList` program creates four `Student` objects and prints them. Because the `Student` objects are added to the `TreeSet`, which is a sorted set, the objects are sorted.

Code 9-5 `StudentList` Program

```
import java.util.*;
public class ComparableTest {
  public static void main(String[] args) {
    TreeSet studentSet = new TreeSet();
    studentSet.add(new Student("Mike", "Hauffmann",101,4.0));
    studentSet.add(new Student("John", "Lynn",102,2.8 ));
    studentSet.add(new Student("Jim", "Max",103, 3.6));
    studentSet.add(new Student("Kelly", "Grant",104,2.3));
    Object[] studentArray = studentSet.toArray();
    Student s;
    for(Object obj : studentArray){
      s = (Student) obj;
      System.out.printf("Name = %s %s ID = %d GPA = %.1f\n",
        s.firstName(), s.lastName(), s.studentID(), s.GPA());
    }
  }
}
```

The output generated by the `StudentList` program is

```
Name = Kelly Grant ID = 104 GPA = 2.3
Name = John Lynn ID = 102 GPA = 2.8
Name = Jim Max ID = 103 GPA = 3.6
Name = Mike Hauffmann ID = 101 GPA = 4.0
```

Observe that the students are compared based on the GPA. This happens because while ordering the elements in the `TreeSet` collection, the `TreeSet` looks if the objects have their natural order, and in this case, uses the `compareTo` method to compare the objects.

Some collections, such as `TreeSet`, are sorted. The `TreeSet` implementation needs to know how to order elements. If the elements have a natural order, `TreeSet` uses the natural order. Otherwise, you have to assist it. For example, the `TreeSet` class has the following constructor that takes `Comparator` as a parameter.

```
TreeSet(Comparator comparator)
```

This constructor creates a new, empty tree set, sorted according to the specified `Comparator`. The following section provides a detailed discussion of the use of the `Comparator` interface.

The `Comparator` Interface

The `Comparator` interface provides greater flexibility with ordering. For example, if you consider the `Student` class described previously, the sorting of students was restricted to sorting on GPAs. It was not possible to sort the student based on first name or some other criteria. This section demonstrates how sorting flexibility can be enhanced using the `Comparator` interface.

The `Comparator` interface is a member of the `java.util` package. It is used to compare objects in the custom order instead of the natural order. For example, it can be used to sort objects in an order other than the natural order. It is also used to sort objects that do not implement the `Comparable` interface.

To write a custom `Comparator`, you need to provide an implementation for the `compare` method in the interface:

```
int compare(Object o1, Object o2)
```

This method compares two arguments for order and returns a negative integer if the first argument is less than second, returns zero if both are equal, and returns a positive integer if the first argument is greater than the second. Code 9-6 shows the version of the `Student` class.

Code 9-6 `Student` Class

```
class Student {
  String firstName, lastName;
  int studentID=0;
  double GPA=0.0;
  public Student(String firstName, String lastName,
    int StudentID, double GPA) {
    if (firstName == null || lastName == null || StudentID == 0 ||
      GPA == 0.0) throw new IllegalArgumentException();
      this.firstName = firstName;
      this.lastName = lastName;
      this.studentID = studentID;
      this.GPA = GPA;
  }
  public String firstName() { return firstName; }
  public String lastName()  { return lastName;  }
  public int studentID() { return studentID; }
  public double GPA() { return GPA; }
}
```

Several classes can be created to compare the students based on `firstName`, or `lastName`, or `studentID`, or `GPA`. The class `NameComp` in Code 9-7 implements the `Comparator` interface to compare students based on `firstName`.

Code 9-7 Example `Comparator` Interface Implementation

```
import java.util.*;
public class NameComp implements Comparator {
  public int compare(Object o1, Object o2) {
    return
      (((Student)o1).firstName.compareTo(((Student)o2).firstName));
  }
}
```

The class `GradeComp` in Code 9-8 on page 9-15 implements the `Comparator` interface to compare students based on their `GPA`s.

Code 9-8 Another Example Comparator Interface Implementation.

```
import java.util.*;
public class GradeComp implements Comparator {
  public int compare(Object o1, Object o2) {
    if (((Student)o1).GPA == ((Student)o2).GPA)
      return 0;
    else if (((Student)o1).GPA < ((Student)o2).GPA)
      return -1;
    else
      return 1;
  }
}
```

The ComparatorTest class in Code 9-9 tests the NameComp comparator. In line 4 notice that the name comparator is passed as a parameter to TreeSet

Code 9-9 ComparatorTest Program.

```
1   import java.util.*;
2   public class ComparatorTest {
3     public static void main(String[] args) {
4       Comparator c = new NameComp();
5       TreeSet studentSet = new TreeSet(c);
6       studentSet.add(new Student("Mike", "Hauffmann",101,4.0));
7       studentSet.add(new Student("John", "Lynn",102,2.8 ));
8       studentSet.add(new Student("Jim", "Max",103, 3.6));
9       studentSet.add(new Student("Kelly", "Grant",104,2.3));
10      Object[] studentArray = studentSet.toArray();
11      Student s;
12      for(Object obj : studentArray){
13        s = (Student) obj;
14        System.out.printf("Name = %s %s ID = %d GPA = %.1f\n",
15          s.firstName(), s.lastName(), s.studentID(), s.GPA());
16      }
17    }
18  }
```

The output generated by this program is:

```
Name = Jim Max ID = 103 GPA = 3.6
Name = John Lynn ID = 102 GPA = 2.8
Name = Kelly Grant ID = 104 GPA = 2.3
Name = Mike Hauffmann ID = 101 GPA = 4.0
```

If Code 9-9 was modified to use a `GradeComp` object, students would be sorted based on the GPA.

Generics

Collection classes use the `Object` type to permit different input and return types. You need to cast down explicitly to retrieve the object you need. This is not type-safe.

Although the existing collections framework does support homogeneous collections (that is collections of one type of object, for example `Date` objects), there was no mechanism to prevent other object types from being inserted into the collection. Also a retrieval almost always required a cast.

The solution for this problem is to make use of *generics* functionality. This was introduced in the Java SE 5.0 platform. It provides information for the compiler about the type of collection used. Hence, type checking is resolved automatically at run time. This eliminates the explicit casting of the data types to be used in the collection. With the addition of autoboxing of primitive types, you can use generics to write simpler and more understandable code.

Before generics, code might look like the following in Code 9-10:

Code 9-10 Using Non-generic Collections

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

In Code 9-10, you need an `Integer` wrapper class for typecasting while retrieving the integer value from the `list`. At runtime, the program needs the type checking for the `list`.

With the application of generics, the `ArrayList` should be declared as `ArrayList<Integer>`, to inform the compiler of the type of collection to be used. When retrieving the value, there is no need for an `Integer` wrapper class. The use of generics for the original code in Code 9-10 is shown in Code 9-11:

Code 9-11 Using Generic Collections

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

The autoboxing facility fits well with the Generics API. Using autoboxing, the code example could be written as shown in Code 9-12.

Code 9-12 Using Autoboxing With Collections

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Code 9-12 uses the generic declaration and instantiation to declare and instantiate an `ArrayList` instance of `Integer` objects. Consequently, the addition of a non-`Integer` type to the array list generates a compilation error.

Note – Generics are enabled by default since the Java SE 5.0 platform. You can disable generics by using the `-source 1.4` option on the `javac` command.

Generic `Set` Example

In the following example, the program declares a variable (`set`) of type `Set<String>` and assigns it to a new `HashSet<String>` object. It then adds a few `String` elements and prints the set to standard output. Line 8 causes a compilation error (because an Integer is not a String). Code 9-13 shows the `Set` implementation using generics. This can be compared to the non-generic `Set` implementation shown in Code 9-1 on page 9-5.

Code 9-13 Generic `Set` Example

```
1   import java.util.*;
2   public class GenSetExample {
3    public static void main(String[] args) {
4       Set<String> set = new HashSet<String>();
5       set.add("one");
6       set.add("second");
7       set.add("3rd");
8       // This line generates compile error
9       set.add(new Integer(4));
10      // Duplicate, not added
11      set.add("second");
12      System.out.println(set);
13    }
14  }
```

Generic Map Example

The `MapPlayerRepository` class in Code 9-14 shows a more practical way to use generic collections. The program creates a repository of players. It declares a variable (`players`) of type `HashMap<String, String>` which stores players based on their position and name. It defines two methods `put()` and `get()` to add the elements to the map repository and to retrieve the elements from the map repository respectively.

Code 9-14 Generic Map Implementation

```
import java.util.*;

public class MapPlayerRepository {
  HashMap<String, String> players;

  public MapPlayerRepository() {
    players = new HashMap<String, String> ();
  }

  public String get(String position) {
    String player = players.get(position);
    return player;
  }

  public void put(String position, String name) {
    players.put(position, name);
  }

  public static void main(String[] args){
    MapPlayerRepository dreamteam = new MapPlayerRepository();

    dreamteam.put("forward", "henry");
    dreamteam.put("rightwing", "ronaldo");
    dreamteam.put("goalkeeper", "cech");
    System.out.println("Forward is " + dreamteam.get("forward"));
    System.out.println("Right wing is "+ dreamteam.get("rightwing"));
    System.out.println("Goalkeeper is "+dreamteam.get("goalkeeper"));
  }
}
```

The program produces the following output.

```
Forward is henry
Right wing is ronaldo
Goalkeeper is cech
```

Generics: Examining Type Parameters

This section examines the use of type parameters in (the class, constructor and method declarations of) generic classes. Table 9-2 compares the non-generic version (pre-Java SE 5.0 platform) and the generic version (since the Java SE 5.0 platform) of the ArrayList class.

Table 9-2 Comparing the non Generic and Generic ArrayList Classes

Category	**Non Generic Class**	**Generic Class**
Class declaration	public class ArrayList extends AbstractList implements List	public class ArrayList<E> extends AbstractList<E> implements List <E>
Constructor declaration	public ArrayList (int capacity)	public ArrayList (int capacity)
Method declaration	public void add(Object o) public Object get(int index)	public void add(E o) public E get(int index)
Variable declaration examples	ArrayList list1; ArrayList list2;	ArrayList <String> a3; ArrayList <Date> a4;

Instance declaration examples	list1 = new ArrayList(10);
	
	list2 = new ArrayList(10);
		a3= new ArrayList<String> (10);
		
		a4= new ArrayList<Date> (10);

The major difference is the introduction of the type parameter E. In the following ArrayList declaration and instantiation, the String type substitutes the parametric type variable E.

```
ArrayList <String> a3 = new ArrayList <String> (10);
```

In the following ArrayList declaration and instantiation, the Date type substitutes the parametric type variable E.

```
ArrayList <Date> a3 = new ArrayList <Date> (10);
```

Figure 9-3 shows a UML diagram of the primary interfaces and implementation classes of the generic Collections API.

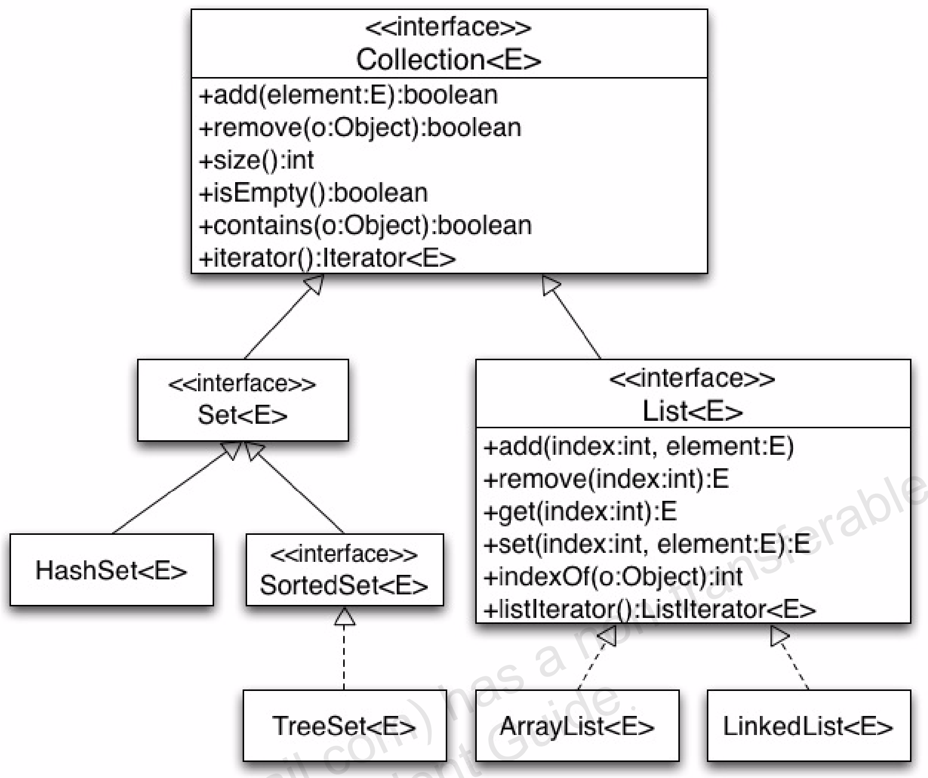

Figure 9-3 Generic Collections API

Note – The type variable `E` in each interface declaration stands for the type of the elements in the collection.

It is common practice to use upper case letters for type parameters. Java libraries use:

- E for element type of collection
- K and V for key-value pairs
- T for all the other types

Wild Card Type Parameters

The discussions contained in this section use the inheritance hierarchy of the `Account` class shown in Figure 9-4.

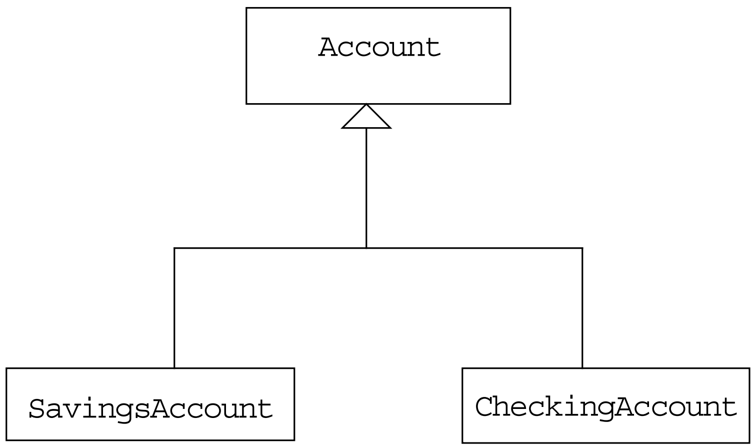

Figure 9-4 The `Account` Class and its Subclasses

This section introduces wildcard type parameters.

The Type-Safety Guarantee

Examine the code shown in Code 9-15.

Code 9-15 Type-Safety Discussion Code

```
import com.mybank.domain.*;
import java.util.*;

public class TestTypeSafety {

  public static void main(String[] args) {
    List<CheckingAccount> lc = new ArrayList<CheckingAccount>();

    lc.add(new CheckingAccount("Fred")); // OK
    lc.add(new SavingsAccount("Fred"));  // Compile error!

    // therefore...
    CheckingAccount ca = lc.get(0);      // Safe, no cast required
  }
}
```

If generic collections mean that inappropriate object types can never be added, then the guarantee is that objects retrieved from the collection can be directly and safely assigned to variables of the same type as the actual type parameter.

The Invariance Challenge

Consider the code contained in Code 9-16.

Code 9-16 Invariance of Assigning Collections of Different Types

```
import com.mybank.domain.*;
import java.util.*;

public class TestInvariance {

  public static void main(String[] args) {
    List<Account> la;
    List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
    List<SavingsAccount> ls = new ArrayList<SavingsAccount>();

     //if the following were possible...
    la = lc;
    la.add(new CheckingAccount("Fred"));

     //then the following must also be possible...
    la = ls;
    la.add(new CheckingAccount("Fred"));

     //so...
    SavingsAccount sa = ls.get(0); //aarrgghh!!
  }
}
```

In fact, `la=lc;` is illegal, so, even though a `CheckingAccount` is an `Account`, an `ArrayList<CheckingAccount>` is not an `ArrayList<Account>`.

For the type-safety guarantee always to be valid, it must be impossible to assign a collection of one type to a collection of a different type, even if the second type is a subclass of the first type.

This is at odds with traditional polymorphism and would appear at first glance to make generic collections somewhat inflexible.

The Covariance Response

Consider the code shown in Code 9-17.

Code 9-17 Using Covarient Types

```
import com.mybank.domain.*;
import java.util.*;

public class TestCovariance {

  public static void printNames(List <? extends Account> lea) {
     for (int i=0; i < lea.size(); i++) {
        System.out.println(lea.get(i).getName());
     }
  }

  public static void main(String[] args) {
    List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
    List<SavingsAccount> ls = new ArrayList<SavingsAccount>();

    printNames(lc);
    printNames(ls);

    //but...
    List<? extends Object> leo = lc; //OK
    leo.add(new CheckingAccount("Fred"));//Compile error!
  }
}
```

Wildcards allow a degree of flexibility when working with generic collections. The `printNames` method is declared with an argument that includes a wildcard. The wildcard '?' in `List <? extends Account>` might be interpreted as "any kind of list of unknown elements that are of type `Account` or a subclass of `Account`." The upper bound (`Account`) means that the elements in the collection can safely be assigned to an `Account` variable. Thus, the two collections of `Account` subtypes can both be passed to the `printNames` method.

This covariance response is designed to be read from, rather than be written to. Because of the invariance principle, it is illegal to add to a collection that uses a wildcard with the extends keyword.

Generics: Refactoring Existing Non-Generic Code

With generic collections, you can specify generic types without type arguments, which are called raw types. This feature is allowed to provide compatibility with the non-generic code.

At compile time, all the generic information from the generic code is removed, leaving behind a raw type. This enables interoperability with the legacy code as the class files generated by the generic code and the legacy code would be the same. At runtime, `ArrayList<String>` and `ArrayList<Integer>` get translated into `ArrayList`, which is a raw type.

Using the new Java SE 5.0 or later compiler on older, non-generic code, generates a warning. Code 9-18 illustrates a class that generates the compile-time warning.

Code 9-18 A Class That Issues a Warning

```
import java.util.*;
public class GenericsWarning {
  public static void main(String[] args) {
    List list = new ArrayList();
    list.add(0, new Integer(42));
    int total = (Integer) list.get(0);
  }
}
```

If you compile the `GenericsWarning` class using the following command:

`javac GenericsWarning.java`

you observe the following warning:

```
Note: GenericsWarning.java uses unchecked or unsafe
operations.
Note: Recompile with -Xlint:unchecked for details.
```

Alternatively, if you compile the class using the following command:

`javac -Xlint:unchecked GenericsWarning.java`

you observe the warning:

```
GenericsWarning.java:5: warning: [unchecked] unchecked call
to add(int,E) as a member of the raw type java.util.List
list.add(0, new Integer(42));
```

```
              ^
1 warning
```

Although the class compiles fine and the warnings can be ignored, you should heed these warnings and modify your code to be generics-friendly. To resolve this warning in the GenericsWarning class, you need to change line 4 to read:

```
List<Integer> list = new ArrayList<Integer>();
```

Iterators

You can scan (iterate over) a collection using an iterator. The basic `Iterator` interface enables you to scan forward through any collection. In the case of an iteration over a set, the order is non-deterministic. The order of an iteration over a list moves forward through the list elements. A `List` object also supports a `ListIterator`, which permits you to scan the list backwards and insert or modify list elements.

Note – The order of the set is deterministic if the set is an instance of some class that guarantees the order. For example if the set is an instance of `TreeSet`, which implements `SortedSet`, the order of the set is deterministic.

Code 9-19 demonstrates the use of an iterator:

Code 9-19 Using Iterators

```
List list<Student> = new ArrayList<Student>();
// add some elements
Iterator<Student> elements = list.iterator();
while (elements.hasNext()) {
  System.out.println(elements.next());
}
```

Figure 9-5 shows a UML diagram of the generic `Iterator` interfaces for the Collections API .

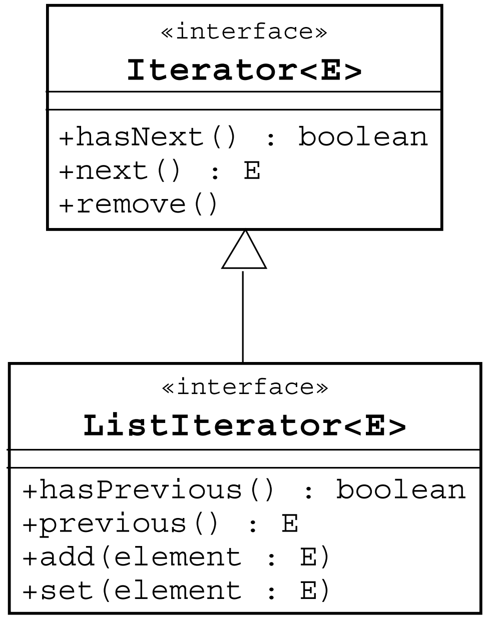

Figure 9-5 UML Diagram: Generic Iterator Interfaces

The `remove` method enables the code to remove the current item in the iteration (the item returned by the most recent next or previous method). If removal is not supported by the underlying collection, then an `UnsupportedOperationException` is thrown.

While using a `ListIterator`, it is common to move through the list in only one direction: either forward using next or backward using previous. If you use previous immediately after next, then you receive the same element; likewise, for calling next after previous.

The `set` method changes the element of the collection currently referenced by the iterator's cursor. The `add` method inserts the new element into the collection immediately before the iterator's cursor. Therefore, if you call `previous` after an `add`, then it returns the newly added element. However, a call to `next` is not affected. If setting or adding is not supported by the underlying collection, then an `UnsupportedOperationException` is thrown.

The Enhanced for Loop

Code 9-20 illustrates the use of an iterator in combination with a traditional for loop to iterate over a collection.

Code 9-20 Using an Iterator With Traditional for Loop

```
public void deleteAll(Collection<NameList> c) {
  for (Iterator<NameList> i=c.iterator(); i.hasNext();)
{
    NameList nl = i.next();
    nl.deleteItem();
  }
}
```

In Code 9-20, the method deleteAll uses variable i three times in the for loop. This provides opportunity for coding errors to be introduced.

Alternatively, you can iterate through a collection by using an enhanced for loop. The enhanced for loop makes traversing through a collection simple, understandable, and safe. The enhanced for loop eliminates the usage of separate iterator methods and minimizes the number of occurrences of the iterator variable. Code 9-21 illustrates the method deleteAll with the enhanced for loop.

Code 9-21 Iterating Using the Enhanced for Loop in Collections

```
public void deleteAll(Collection<NameList> c) {
  for (NameList nl: c) {
    nl.deleteItem();
  }
}
```

The functionality of the enhanced for loop, makes nested for loops traversal simpler and easier to understand in comparison with the traditional for loop. One reason for this is, using enhanced for loops reduces the ambiguity of the variables used in the nested loops. Code 9-22 contains an example of nested enhanced for loops.

Code 9-22 Nesting Enhanced for Loops

```
List<Subject> subjects=...;
List<Teacher> teachers=...;
List<Course> courseList = new ArrayList<Course>();
for (Subject subj: subjects) {
  for (Teacher tchr: teachers) {
    courseList.add(new Course(subj, tchr));
  }
}
```