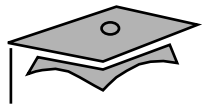


Module 9

Collections and Generics Framework



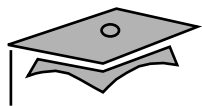
Objectives

- Describe the Collections
- 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

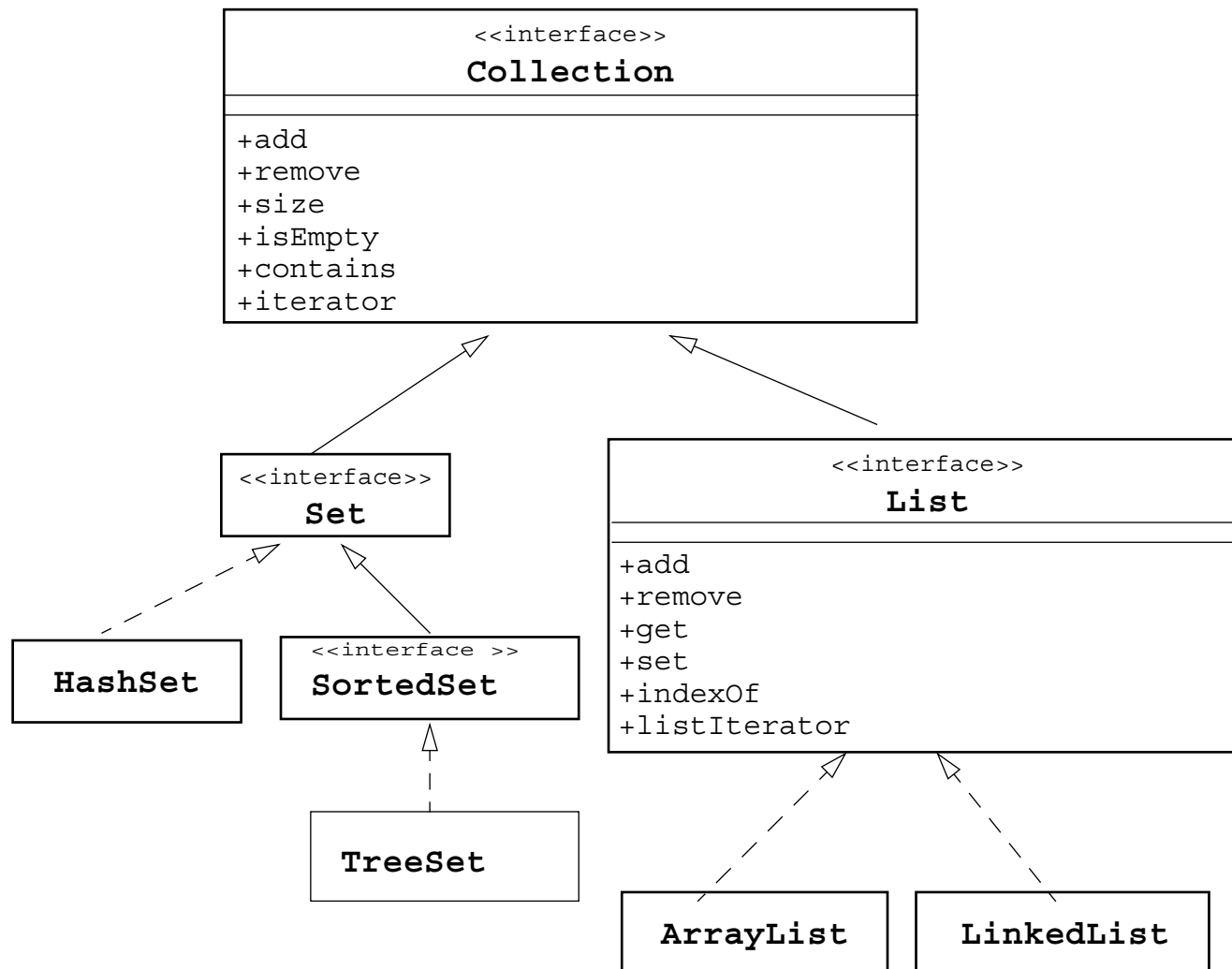


The Collections API

- A *collection* is a single object managing a group of objects known as its elements.
- The Collections API contains interfaces that group objects as one of the following:
 - `Collection` – A group of objects called 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.



The Collections API

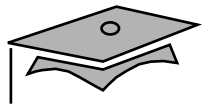




Collection Implementations

There are several general purpose implementations of the core interfaces (Set, List, Deque and Map)

| | 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 |

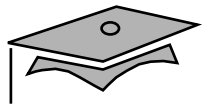


A Set Example

```
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 is:

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



A List Example

```
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 }
```

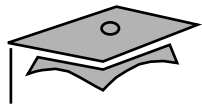
The output generated from this program is:

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

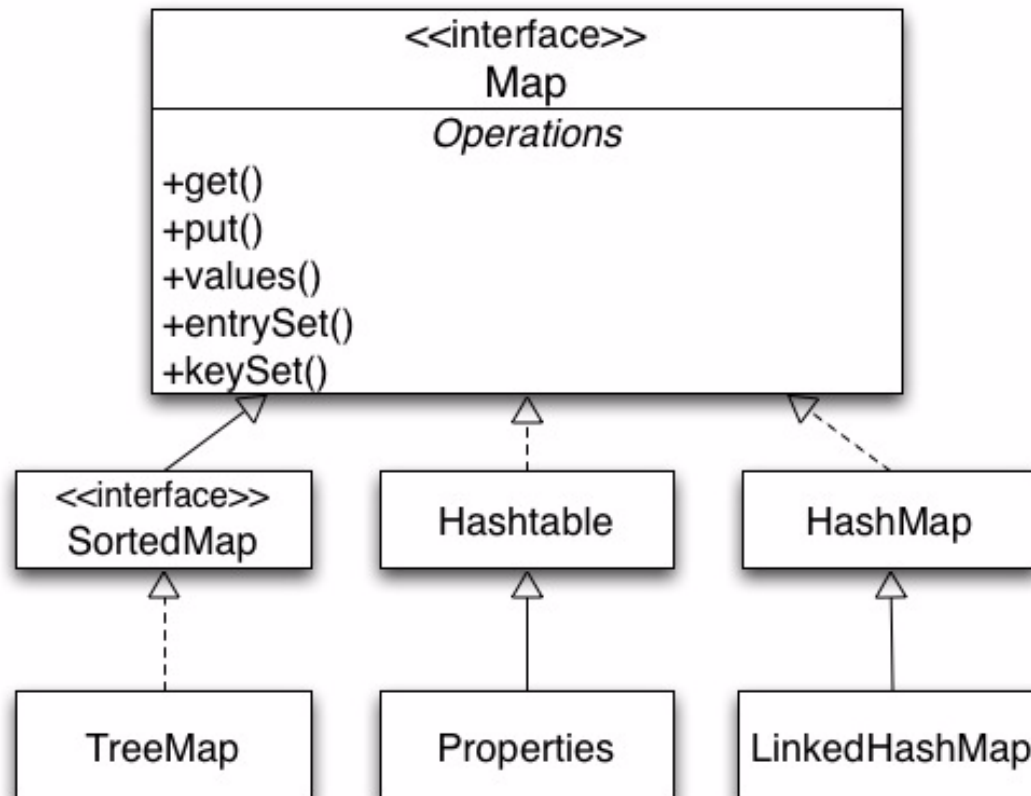


The Map Interface

- Maps are sometimes called associative arrays
- A Map object describes mappings from keys to values:
 - Duplicate keys are not allowed
 - One-to-many mappings from keys to values is not permitted
- The contents of the Map interface can be viewed and manipulated as collections
 - `entrySet` – Returns a Set of all the key-value pairs.
 - `keySet` – Returns a Set of all the keys in the map.
 - `values` – Returns a Collection of all values in the map.



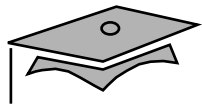
The Map Interface API





A Map Example

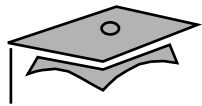
```
1  import java.util.*;
2  public class MapExample {
3      public static void main(String args[]) {
4          Map map = new HashMap();
5          map.put("one", "1st");
6          map.put("second", new Integer(2));
7          map.put("third", "3rd");
8          // Overwrites the previous assignment
9          map.put("third", "III");
10         // Returns set view of keys
11         Set set1 = map.keySet();
12         // Returns Collection view of values
13         Collection collection = map.values();
14         // Returns set view of key value mappings
15         Set set2 = map.entrySet();
16         System.out.println(set1 + "\n" + collection + "\n" + set2);
17     }
18 }
```



A Map Example

Output generated from the MapExample program:

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



Legacy Collection Classes

Collections in the JDK include:

- The `Vector` class, which implements the `List` interface.
- The `Stack` class, which is a subclass of the `Vector` class and supports the `push`, `pop`, and `peek` methods.
- The `Hashtable` class, which implements the `Map` interface.
- 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` interface is incompatible with, the `Iterator` 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 specifies order relation. It can also be used to override natural ordering.
- Both interfaces are useful for sorting collections.



The Comparable Interface

Imparts natural ordering to classes that implement it:

- Used for sorting
- The `compareTo` method should be implemented to make any class comparable:
 - `int compareTo(Object o)` method
- The `String`, `Date`, and `Integer` classes implement the `Comparable` interface
- You can sort the `List` elements containing objects that implement the `Comparable` interface



The Comparable Interface

- While sorting, the `List` elements follow the natural ordering of the element types
 - `String` elements – Alphabetical order
 - `Date` elements – Chronological order
 - `Integer` elements – Numerical order



Example of the Comparable Interface

```
1  import java.util.*;
2  class Student implements Comparable {
3      String firstName, lastName;
4      int studentID=0;
5      double GPA=0.0;
6      public Student(String firstName, String lastName, int studentID,
7          double GPA) {
8          if (firstName == null || lastName == null || studentID == 0
9              || GPA == 0.0) {throw new IllegalArgumentException();}
10         this.firstName = firstName;
11         this.lastName = lastName;
12         this.studentID = studentID;
13         this.GPA = GPA;
14     }
15     public String firstName() { return firstName; }
16     public String lastName() { return lastName; }
17     public int studentID() { return studentID; }
18     public double GPA() { return GPA; }
```



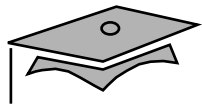

Example of the Comparable Interface

```
19    // Implement compareTo method.
20    public int compareTo(Object o) {
21        double f = GPA-((Student)o).GPA;
22        if (f == 0.0)
23            return 0;    // 0 signifies equals
24        else if (f<0.0)
25            return -1;    // negative value signifies less than or before
26        else
27            return 1;    // positive value signifies more than or after
28    }
29 }
```



Example of the Comparable Interface

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



Example of the Comparable Interface

Generated Output:

```
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
```



The Comparator Interface

- Represents an order relation
- Used for sorting
- Enables sorting in an order different from the natural order
- Used for objects that do not implement the Comparable interface
- Can be passed to a sort method

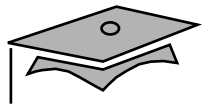
You need the compare method to implement the Comparator interface:

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



Example of the Comparator Interface

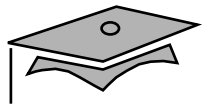
```
1  class Student {
2      String firstName, lastName;
3      int studentID=0;
4      double GPA=0.0;
5      public Student(String firstName, String lastName,
6          int studentID, double GPA) {
7          if (firstName == null || lastName == null || studentID == 0 ||
8              GPA == 0.0) throw new NullPointerException();
9          this.firstName = firstName;
10         this.lastName = lastName;
11         this.studentID = studentID;
12         this.GPA = GPA;
13     }
14     public String firstName() { return firstName; }
15     public String lastName() { return lastName; }
16     public int studentID() { return studentID; }
17     public double GPA() { return GPA; }
18 }
```



Example of the Comparator Interface

```
1  import java.util.*;
2  public class NameComp implements Comparator {
3      public int compare(Object o1, Object o2) {
4          return
5              (((Student)o1).firstName.compareTo(((Student)o2).firstName));
6      }
7  }
```

```
1  import java.util.*;
2  public class GradeComp implements Comparator {
3      public int compare(Object o1, Object o2) {
4          if (((Student)o1).GPA == ((Student)o2).GPA)
5              return 0;
6          else if (((Student)o1).GPA < ((Student)o2).GPA)
7              return -1;
8          else
9              return 1;
10     }
11 }
```



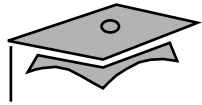
Example of the Comparator Interface

```
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.println("Name = %s %s ID = %d GPA = %.1f\n",
15                 s.firstName(), s.lastName(), s.studentID(), s.GPA());
16         }
17     }
18 }
```



Example of the Comparator Interface

```
Name = Jim Max ID = 0 GPA = 3.6  
Name = John Lynn ID = 0 GPA = 2.8  
Name = Kelly Grant ID = 0 GPA = 2.3  
Name = Mike Hauffmann ID = 0 GPA = 4.0
```

Generics

Generics are described as follows:

- Provide compile-time type safety
- Eliminate the need for casts
- Provide the ability to create compiler-checked homogeneous collections



Generics

Using non-generic collections:

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

Using generic collections:

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



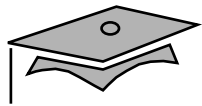
Generic Set Example

```
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         set.add("second");
11     // Duplicate, not added
12         System.out.println(set);
13     }
14 }
```



Generic Map Example

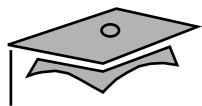
```
1  import java.util.*;
2
3  public class MapPlayerRepository {
4      HashMap<String, String> players;
5
6      public MapPlayerRepository() {
7          players = new HashMap<String, String> ();
8      }
9
10     public String get(String position) {
11         String player = players.get(position);
12         return player;
13     }
14
15     public void put(String position, String name) {
16         players.put(position, name);
17     }
```



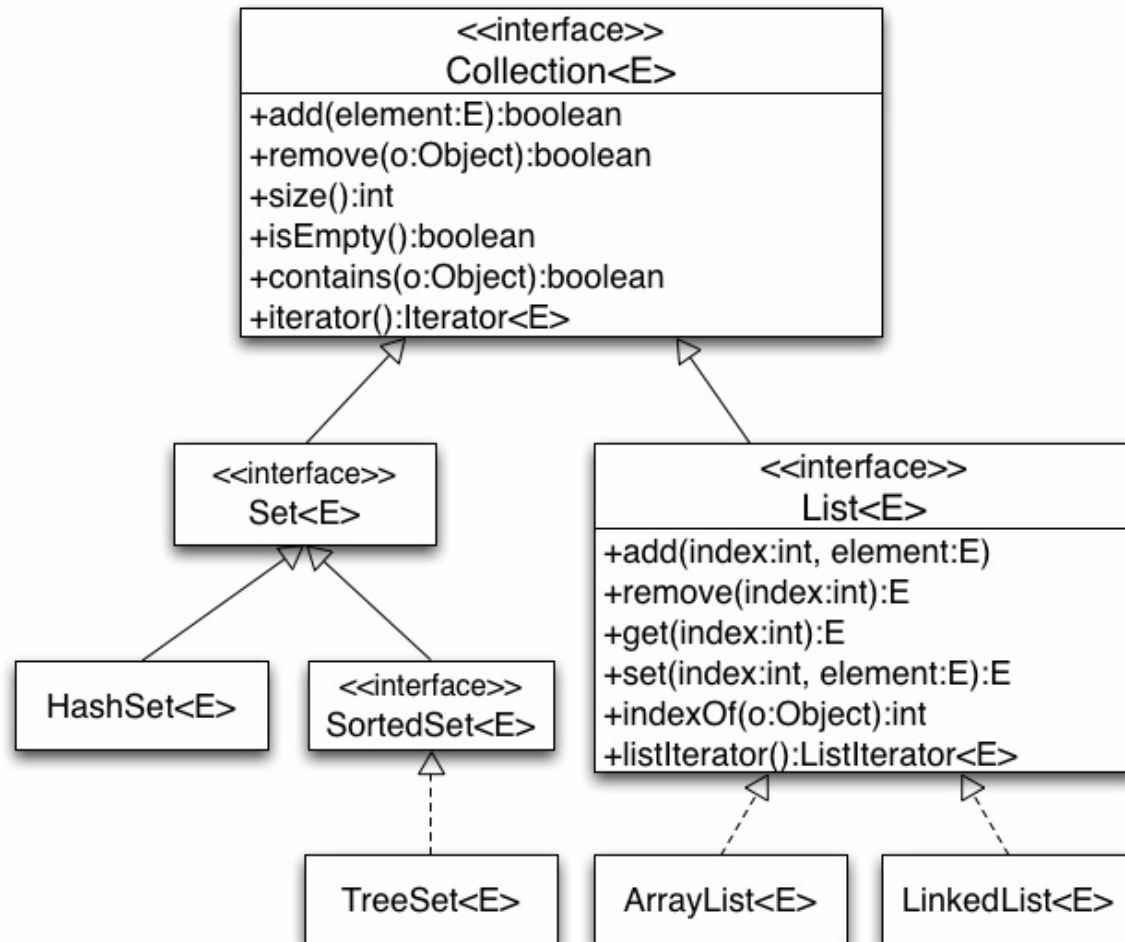
Generics: Examining Type Parameters

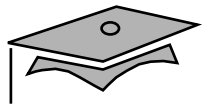
Shows how to use type parameters

| Category | Non Generic Class | Generic Class |
|-------------------------------|---|--|
| Class declaration | <code>public class ArrayList extends AbstractList implements List</code> | <code>public class ArrayList<E> extends AbstractList<E> implements List <E></code> |
| Constructor declaration | <code>public ArrayList (int capacity);</code> | <code>public ArrayList<E> (int capacity);</code> |
| Method declaration | <code>public void add((Object o) public Object get(int index)</code> | <code>public void add(E o) public E get(int index)</code> |
| Variable declaration examples | <code>ArrayList list1; ArrayList list2;</code> | <code>ArrayList <String> list1; ArrayList <Date> list2;</code> |
| Instance declaration examples | <code>list1 = new ArrayList(10); list2 = new ArrayList(10);</code> | <code>list1= new ArrayList<String> (10); list2= new ArrayList<Date> (10);</code> |

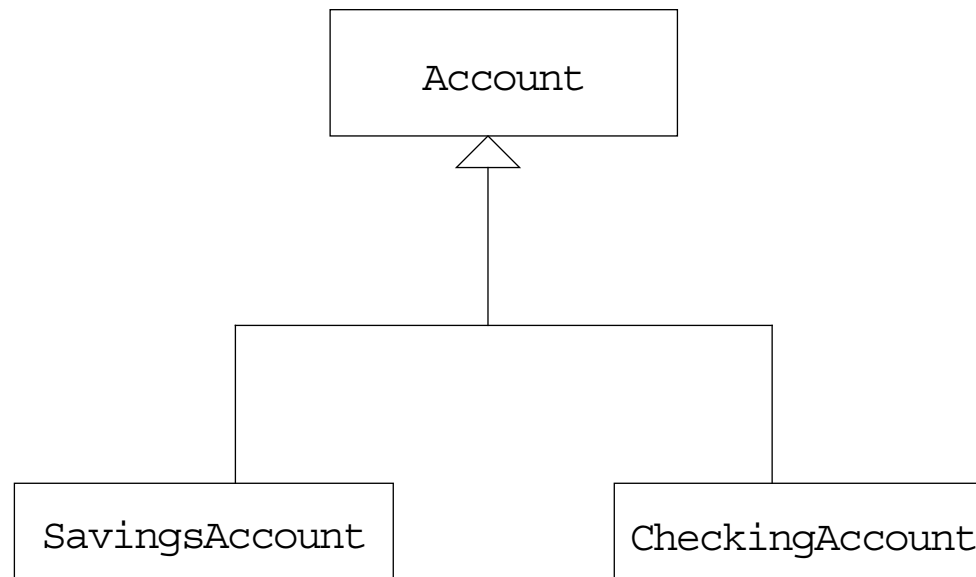


Generic Collections API





Wild Card Type Parameters





The Type-Safety Guarantee

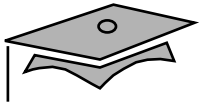
```
1  public class TestTypeSafety {
2
3      public static void main(String[] args) {
4          List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
5
6          lc.add(new CheckingAccount("Fred")); // OK
7          lc.add(new SavingsAccount("Fred")); // Compile error!
8
9          // therefore...
10         CheckingAccount ca = lc.get(0);      // Safe, no cast required
11     }
12 }
```




The Invariance Challenge

```
7      List<Account> la;  
8      List<CheckingAccount> lc = new ArrayList<CheckingAccount>();  
9      List<SavingsAccount> ls = new ArrayList<SavingsAccount>();  
10  
11     //if the following were possible...  
12     la = lc;  
13     la.add(new CheckingAccount("Fred"));  
14  
15     //then the following must also be possible...  
16     la = ls;  
17     la.add(new CheckingAccount("Fred"));  
18  
19     //so...  
20     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>`.



The Covariance Response

```
6 public static void printNames(List <? extends Account> lea) {
7     for (int i=0; i < lea.size(); i++) {
8         System.out.println(lea.get(i).getName());
9     }
10 }
11
12 public static void main(String[] args) {
13     List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
14     List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
15
16     printNames(lc);
17     printNames(ls);
18
19     //but...
20     List<? extends Object> leo = lc; //OK
21     leo.add(new CheckingAccount("Fred")); //Compile error!
22 }
23 }
```



Generics: Refactoring Existing Non- Generic Code

```
1  import java.util.*;
2  public class GenericsWarning {
3      public static void main(String[] args) {
4          List list = new ArrayList();
5          list.add(0, new Integer(42));
6          int total = ((Integer)list.get(0)).intValue();
7      }
8  }
```

javac GenericsWarning.java

Note: GenericsWarning.java uses unchecked or unsafe operations.

Note: Recompile with `-Xlint:unchecked` for details.

javac -Xlint:unchecked GenericsWarning.java

GenericsWarning.java:7: warning: [unchecked] unchecked call to add(int,E)
as a member of the raw type java.util.ArrayList

```
    list.add(0, new Integer(42));
```

^

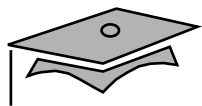
1 warning



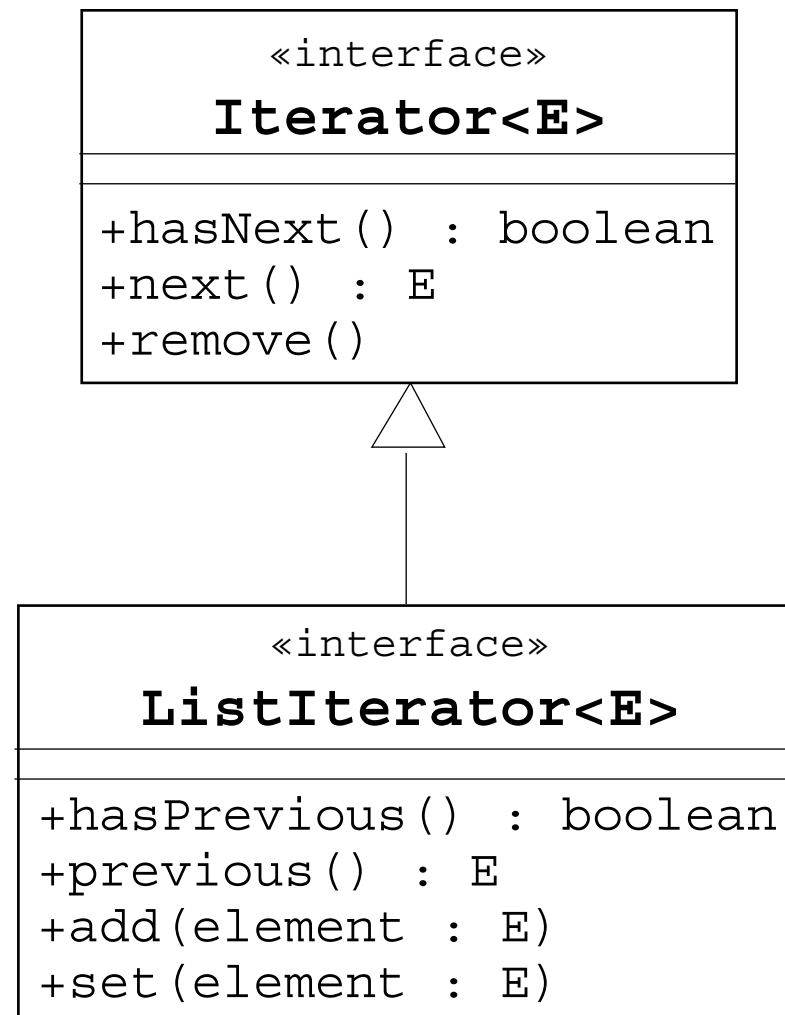
Iterators

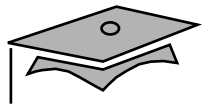
- Iteration is the process of retrieving every element in a collection.
- The basic Iterator interface allows you to scan forward through any collection.
- A List object supports the ListIterator, which allows you to scan the list backwards and insert or modify elements.

```
1 List<Student> list = new ArrayList<Student>();  
2 // add some elements  
3 Iterator<Student> elements = list.iterator();  
4 while (elements.hasNext()) {  
5     System.out.println(elements.next());  
6 }
```



Generic Iterator Interfaces





The Enhanced `for` Loop

The enhanced `for` loop has the following characteristics:

- Simplified iteration over collections
- Much shorter, clearer, and safer
- Effective for arrays
- Simpler when using nested loops
- Iterator disadvantages removed

Iterators are error prone:

- Iterator variables occur three times per loop.
- This provides the opportunity for code to go wrong.



The Enhanced for Loop

An enhanced for loop can look like the following:

- Using the iterator with a traditional for loop:

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

- Iterating using an enhanced for loop in collections:

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



The Enhanced for Loop

- Nested enhanced for loops:

```
1 List<Subject> subjects=...;
2 List<Teacher> teachers=...;
3 List<Course> courseList = ArrayList<Course>();
4 for (Subject subj: subjects) {
5     for (Teacher tchr: teachers) {
6         courseList.add(new Course(subj, tchr));
7     }
8 }
```