WSA | Forward looking IT finishing school

Java Programming Language SE – 6

Module 9:Collections and Generics Framework

ORACLE®

Certified Professional

Java SE 6 Programmer

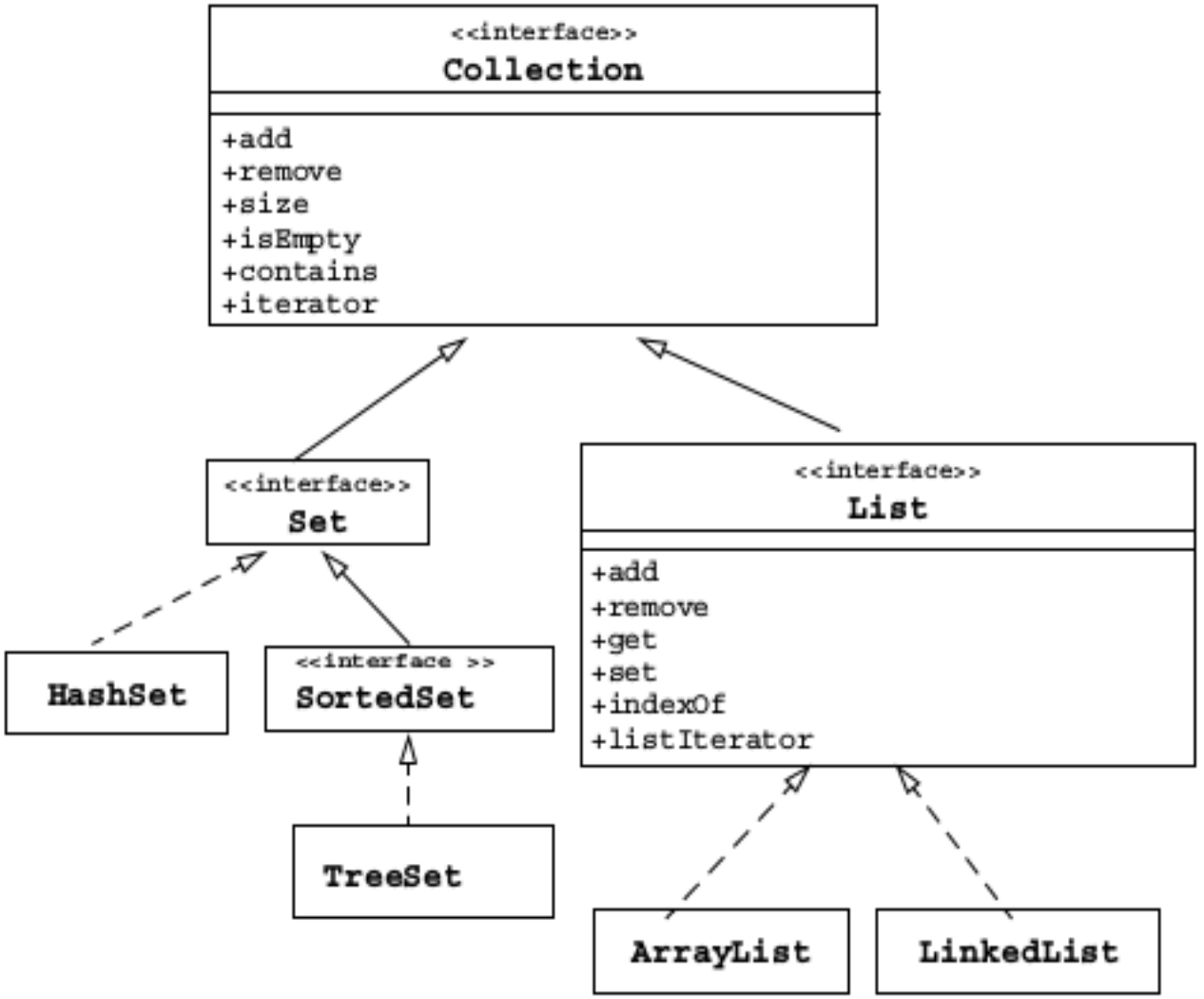
Objectives

- 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



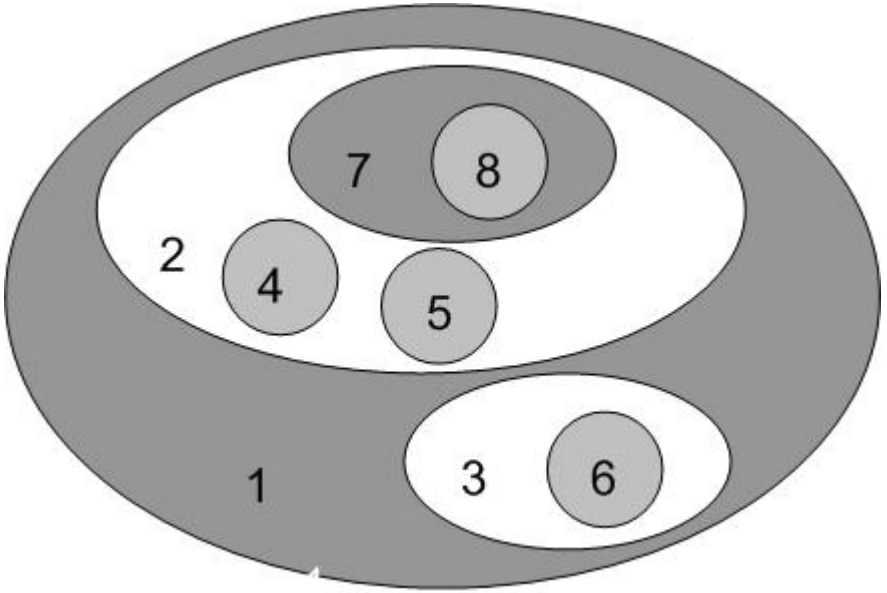
Collection Implementations

- 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

```
import java.util.*;
public class SetExample {
public static void main(String[] args) {
Set set = new HashSet();
set.add("one");
set.add("second");
set.add("3rd");
set.add(new Integer(4));
set.add(new Float(5.0F));
// duplicate, not added
set.add("second");// duplicate, not added
set.add(new Integer(4)); // duplicate, not added
System.out.println(set);
}
```



A Set Example

The output generated from this program is:

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

A List Example

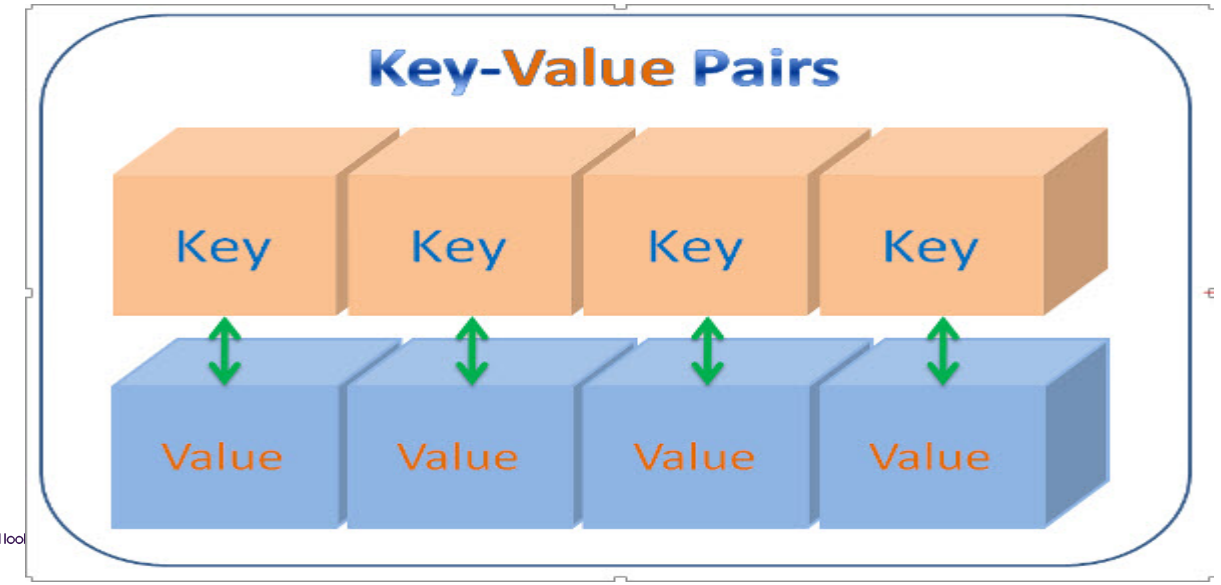
```
import java.util.*
public class ListExample {
public static void main(String[] args) {
List list = new ArrayList();
list.add("one");
list.add("second");
list.add("3rd");
list.add(new Integer(4));
list.add(new Float(5.0F));
// duplicate, is added
list.add("second");//duplicate, is added
list.add(new Integer(4)); // duplicate, is added
System.out.println(list);
}}
```

A List Example

- The output generated from this program is:
- [one, second, 3rd, 4, 5.0, second, 4]

The Map Interface

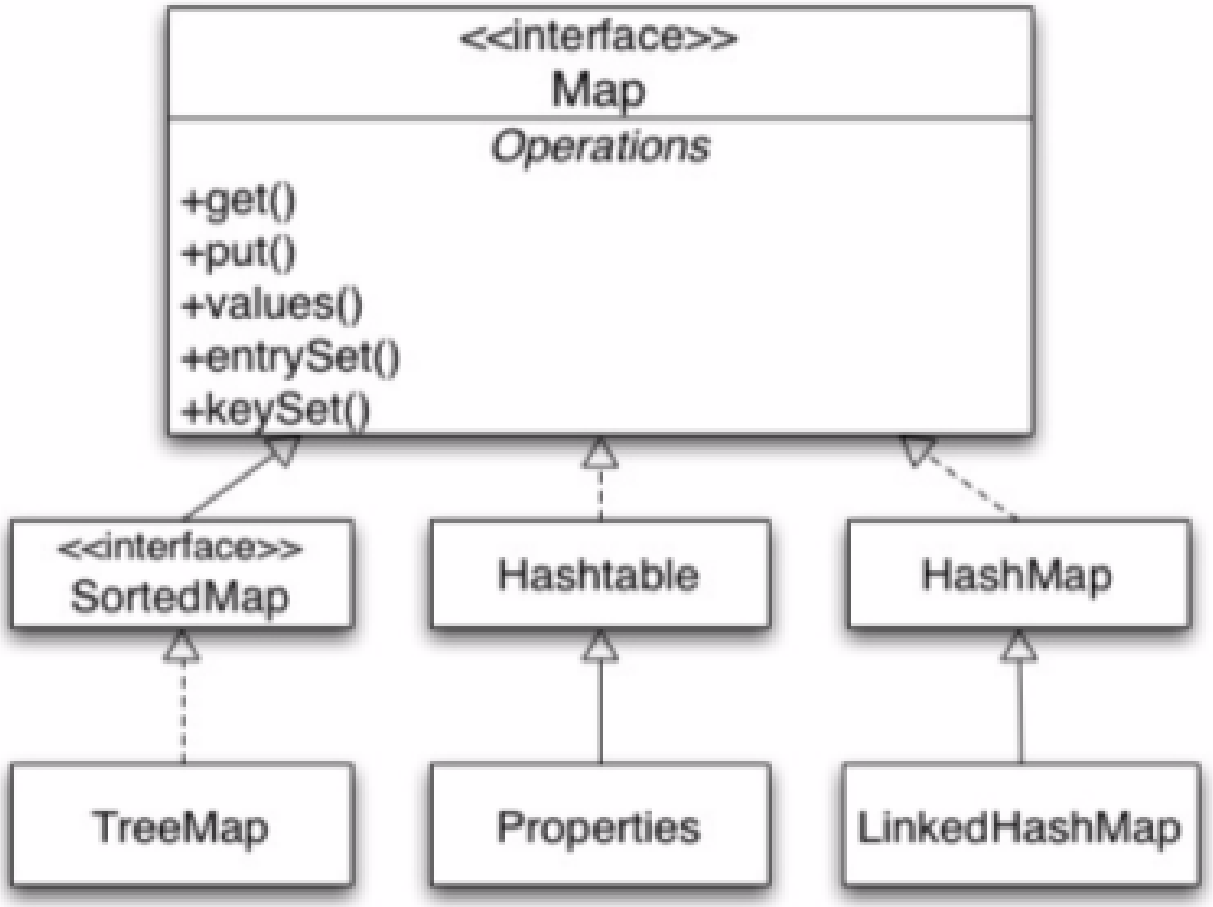
- Maps are sometimes called associative arrays
- A Map object describes mappings from keys to values:
- Duplicate keys are not allowed
- One-to-many mappings from keys to values is not permitted



The Map Interface

- 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

```
public static void main(String args[]) {
Map map = new HashMap();
map.put("one","1st");
map.put("second", new Integer(2));
map.put("third","3rd");
map.put("third","III");
Set set1 = map.keySet();
Collection collection = map.values();
Set set2 = map.entrySet();
System.out.println(set1 + "\n" + collection + "\n" + set2);
}
```

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

```
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;
}
```

Example of the Comparable Interface

```
public String firstName() { return firstName; }
public String lastName() { return lastName; }
public int studentID() { return studentID; }
public double GPA() { return GPA; }
public int compareTo(Object o) {
double f = GPA-((Student)o).GPA;
if (f == 0.0)
return 0;
else if (f<0.0)
return -1;
else
return 1;
}}
```

Example of the Comparable Interface

```
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());
}}
```

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

```
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 NullPointerException();
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; }
}
```

Example of the Comparator Interface

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

Example of the Comparator Interface

```
public class ComparatorTest {
public static void main(String[] args) {
Comparator c = new NameComp();
TreeSet studentSet = new TreeSet(c);
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.println("Name = %s %s ID = %d GPA = %.1f\n",
s.firstName(), s.lastName(), s.studentID(), s.GPA());
}}
```

Example of the Comparator Interface

The output:

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

```
import java.util.*;
public class GenSetExample {
public static void main(String[] args) {
Set<String> set = new HashSet<String>();
set.add("one");
set.add("second");
set.add("3rd");
set.add(new Integer(4));
set.add("second");
System.out.println(set);
}}
```

Generic Map Example

```
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);
}
```

Generics: Examining Type Parameters

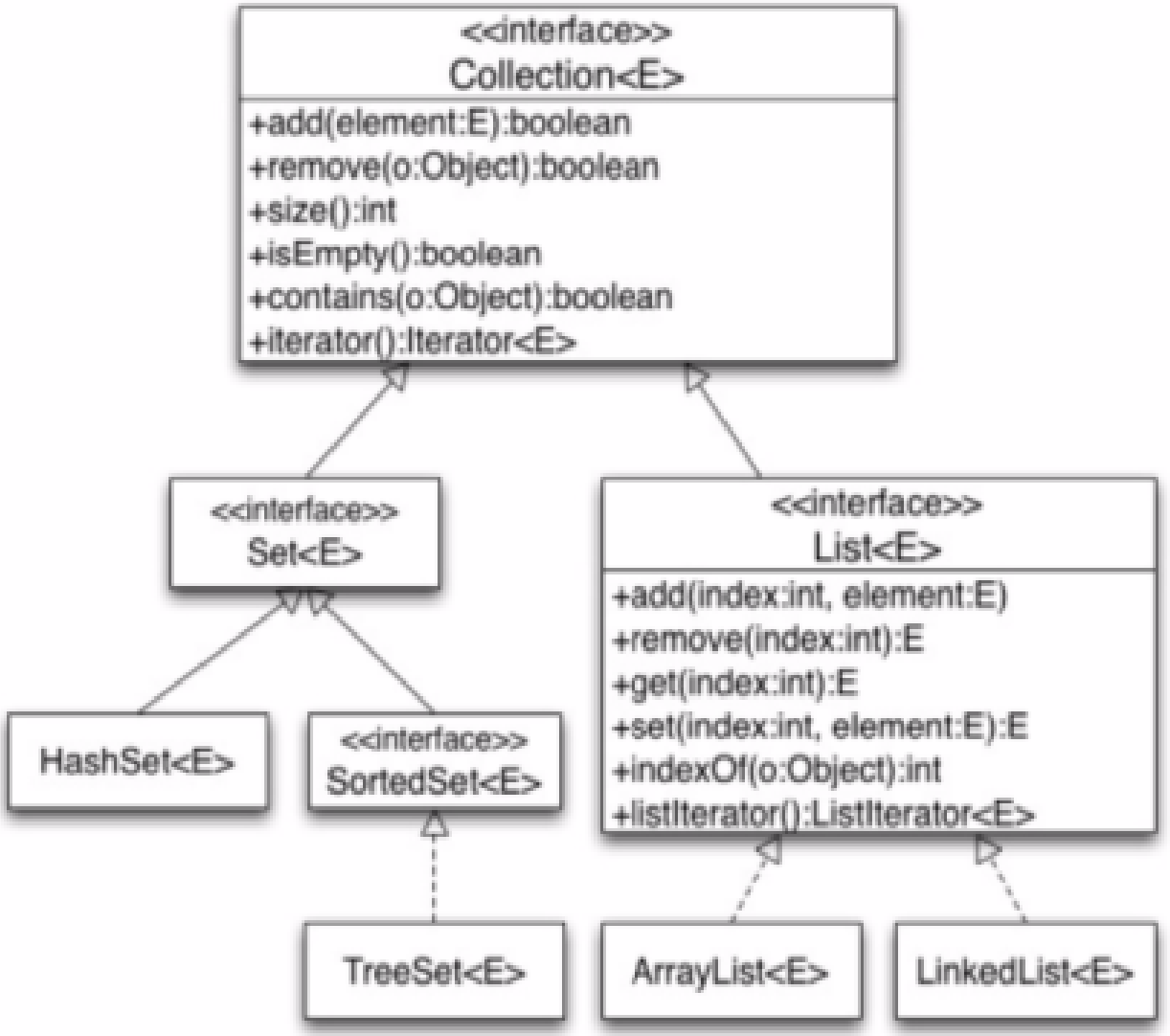
Category	Non Generic Class	Generic Class
Class declaration	`public class ArrayList extends AbstractList implements List`	`public class ArrayList<E> extends AbstractList<E> implements List <E>`
Constructor declaration	`public ArrayList (int capacity);`	`public ArrayList<E> (int capacity);`

Method declaration	`public void add( (Object o)`
	
	`public Object get(int index)`
		`public void add(E o)`
		
		`public E get(int index)`

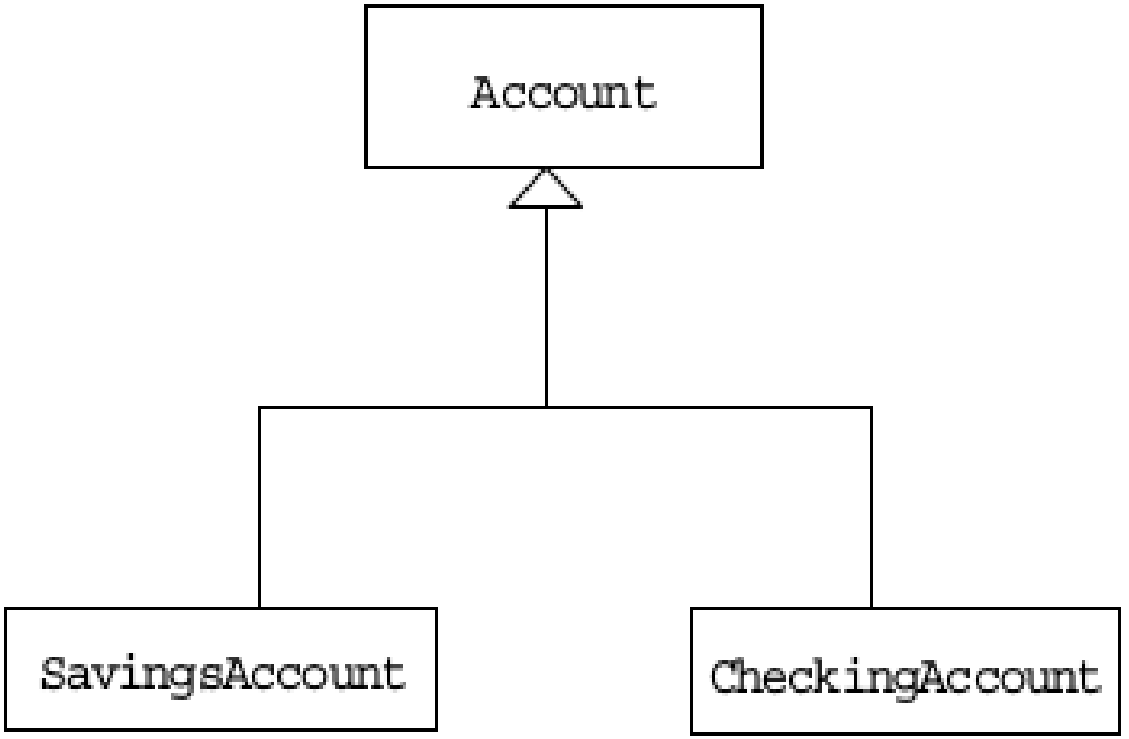
Variable declaration examples	`ArrayList list1;`
	
	`ArrayList list2;`
		`ArrayList <String> list1;`
		
		`ArrayList <Date> list2;`

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

Generic Collections API



Wild Card Type Parameters



The Type-Safety Guarantee

```
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!
CheckingAccount ca = lc.get(0);
}
}
```

The Invariance Challenge

```
List<Account> la;
List<CheckingAccount> lc = new
ArrayList<CheckingAccount>();
List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
la = lc;
la.add(new CheckingAccount("Fred"));
la = ls;
la.add(new CheckingAccount("Fred"));
SavingsAccount sa = ls.get(0); //aarrgghh!!
```

The Invariance Challenge

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

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

Generics: Refactoring Existing Non-Generic Code

```
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).intValue();
}}
```

Generics: Refactoring
Existing Non-Generic Code

- 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));
  ```

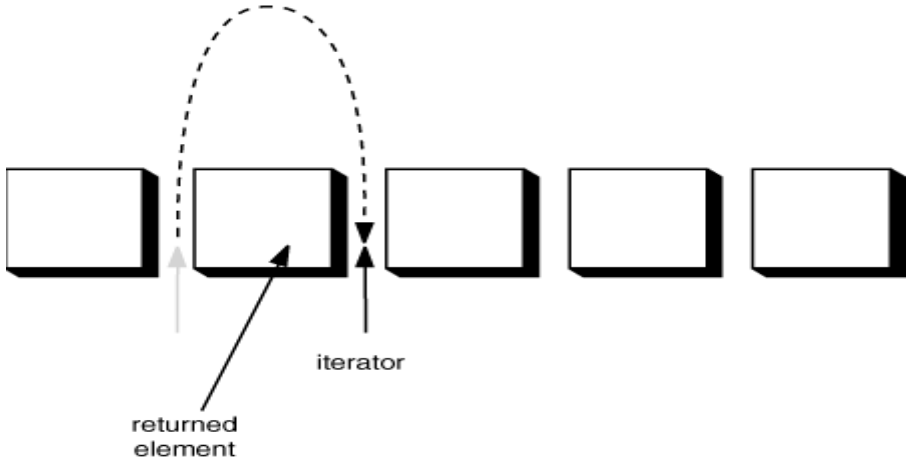
  1 warning

Iterators

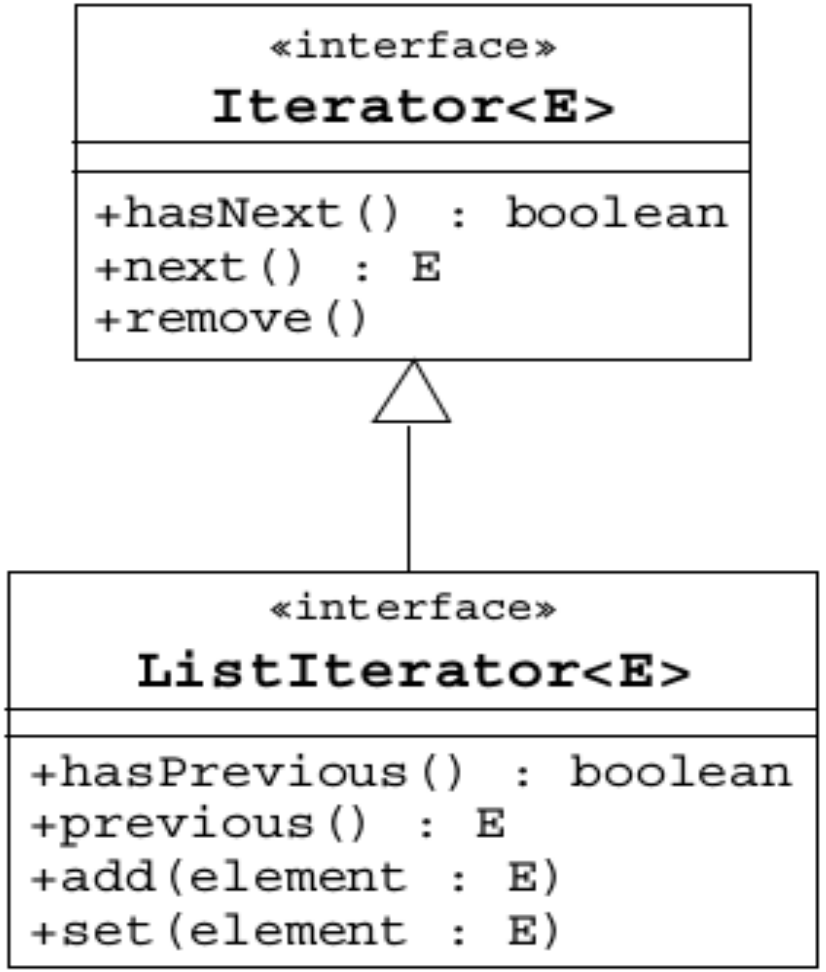
- Iteration is the process of retrieving every element in a collection.
- The basic Iterator interface allows you to scan forward through any collection.
- A List object supports the ListIterator, which allows you to scan the list backwards and insert or modify elements.

Iterators

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



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();
}}
```

The Enhanced for Loop

- *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:*

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

Web Stack Academy (P) Ltd

#83, Farah Towers,

1st floor,MG Road,

Bangalore – 560001

M: +91-80-4128 9576

T: +91-98862 69112

E: info@www.webstackacademy.com