Java Programming

Collection Framework

Module 9

Agenda

1. Introduction to Collection
2. Collection Hierarchy
3. Generics
4. Understanding List, Set and Map

Objectives

At the end of this module, you will be able to:

- Implement various Collection classes and interfaces
- Explore Generics
- Apply Best practices in Collections

Introduction to Collection

Introduction

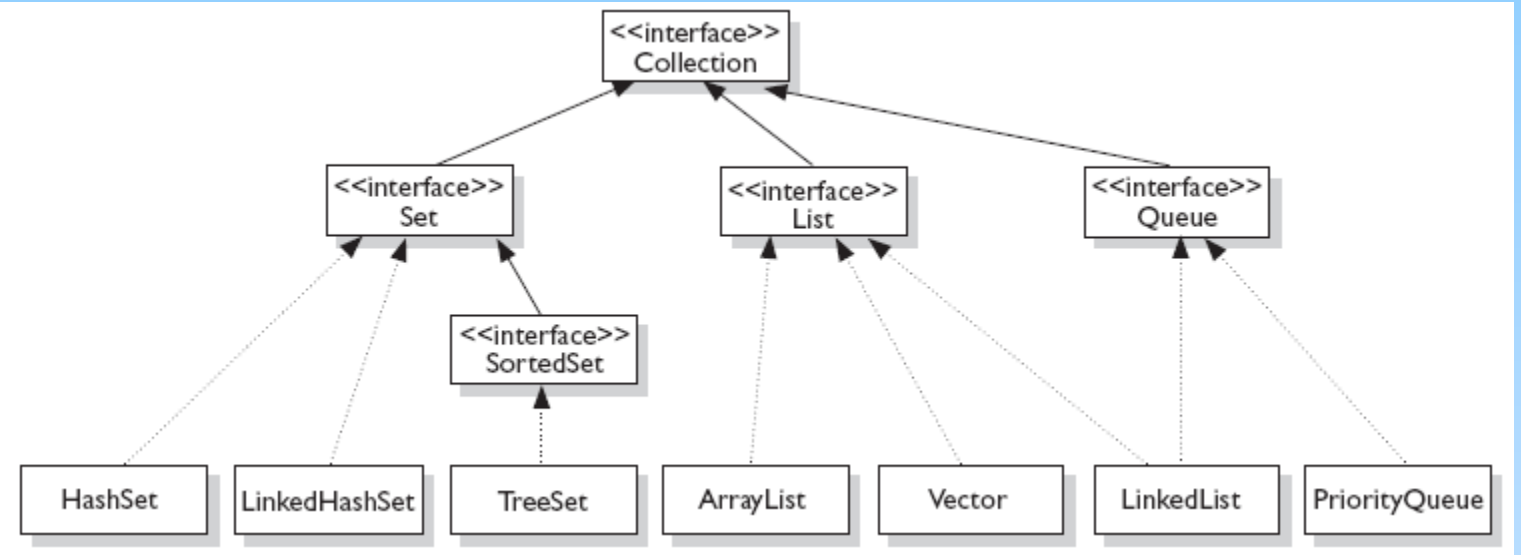
- A Collection is a group of objects
- Collections framework provide a a set of standard utility classes to manage collections
- Collections Framework consists of three parts:
  - Core Interfaces
  - Concrete Implementation
  - Algorithms such as searching and sorting

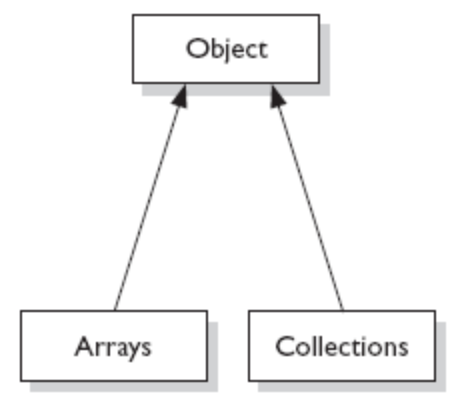
Advantages of Collections

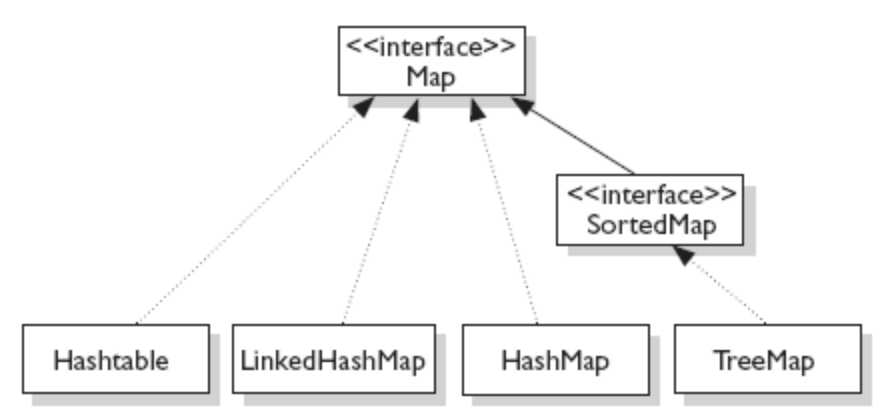
- Reduces programming effort
- Increases performance
- Provides interoperability between unrelated APIs
- Reduces the effort required to learn APIs
- Reduces the effort required to design and implement APIs
- Fosters Software reuse

Collection Hierarchy

Interfaces and their implementation



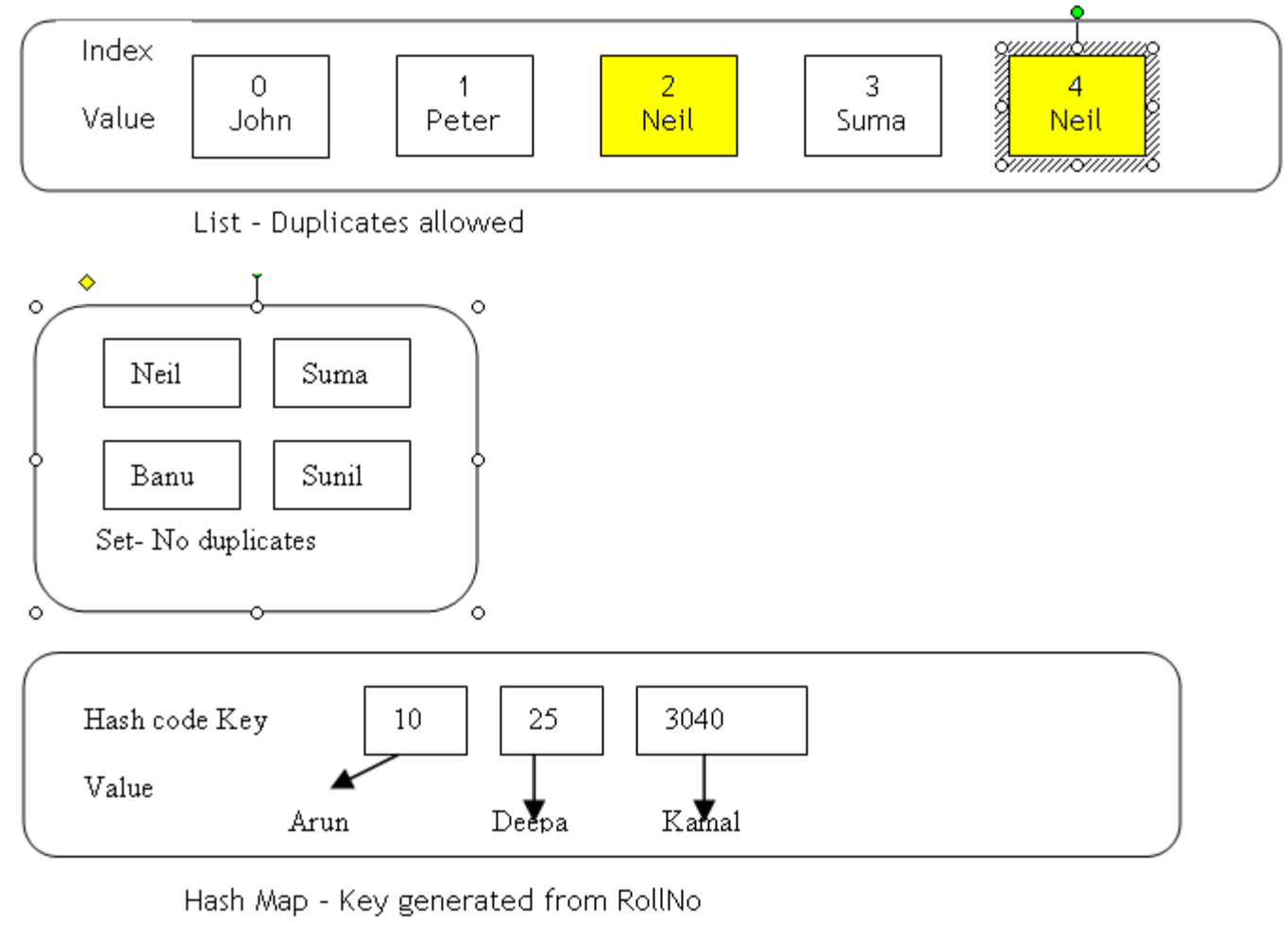




Collection Interfaces

Interfaces	Description
Collection	A basic interface that defines the operations that all the classes that maintain collections of objects typically implement.
Set	Extends the Collection interface for sets that maintain unique element.
SortedSet	Augments the Set interface or Sets that maintain their elements in sorted order.
List	Collections that require position-oriented operations should be created as lists. Duplicates are allowed.
Queue	Things arranged by the order in which they are to be processed.
Map	A basic interface that defines operations that classes that represent mapping of keys to values typically implement.
SortedMap	Extends the Map interface for maps that maintain their mappings in the key order.

Illustration of List, Set and Map



Collection Implementations

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Collection interface methods

Method	Description
`int size();`	Returns number of elements in collection.
`boolean isEmpty();`	Returns true if invoking collection is empty.
`boolean contains(Object element);`	Returns true if element is an element of invoking collection.
`boolean add(Object element);`	Adds element to invoking collection.
`boolean remove(Object element);`	Removes one instance of element from invoking collection
`Iterator iterator();`	Returns an iterator fro the invoking collection
`boolean containsAll(Collection c);`	Returns true if invoking collection contains all elements of *c*; false otherwise.
`boolean addAll(Collection c);`	Adds all elements of *c* to the invoking collection.
`boolean removeAll(Collection c);`	Removes all elements of *c* from the invoking collection
`boolean retainAll(Collection c);`	Removes all elements from the invoking collection except those in *c*.
`void clear();`	Removes all elements from the invoking collection
`Object[] toArray();`	Returns an array that contains all elements stored in the invoking collection
`Object[] toArray(Object a[]);`	Returns an array that contains only those collection elements whose type matches that of *a*.

Quiz

1. Which of the following classes is not implemented from the Collection interface?

a. TreeSet
b. HashTable
c. Vector
d. Linked List

HashTable

2. Which of the following is a class?

a. Collection
b. Collections

Collections

3. Which of the following does not accept duplicate values?

a. ArrayList
b. LinkedList
c. TreeSet
d. Vector

TreeSet

Generics

What are and Why Generics?

- Mechanism by which a single piece of code can manipulate many different data types without explicitly having a separate entity for each data type

Before Generics

```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer x = (Integer) myIntegerList.iterator().next(); // 3
Line no 3 if not properly typecasted will throw runtime exception
```

After Generics

```
List<Integer> myIntegerList = new LinkedList<Integer>(); // 1
myIntegerList.add(new Integer(0)); //2
Integer x = myIntegerList.iterator().next(); // 3
```

What problems does Generics solve?

- Problem: Collection element types
  - Compiler is unable to verify types
  - Assignment must have type casting
  - *ClassCastException* can occur during runtime
- Solution: Generics
  - Tell the compiler type of the collection
  - Let the compiler fill in the cast
  - Example: Compiler will check if you are adding Integer type entry to a String type collection (compile time detection of type mismatch)

Using Generic class

- Instantiate a generic class to create type specific object
- In J2SE 5.0, all collection classes are rewritten to be generic classes

```
Vector<String> vs = new Vector<String>();
   vs.add(new Integer(5)); // Compile error!
   vs.add(new String("hello"));
   String s = vs.get(0); // No casting needed
```

Using Generic class (Contd.).

- Generic class can have multiple type parameters
- Type argument can be a custom type

```
HashMap<String, Mammal> map =
  new HashMap<String, Mammal>();
  map.put(“wombat”, new Mammal("wombat"));
  Mammal w = map.get(“wombat”);
```

Generics and Sub typing

- You can do this

```
Object o = new Integer(5);
```

- You can even do this

```
Object[] or = new Integer[5];
```

- So you would expect to be able to do this

```
ArrayList<Object> ao = new ArrayList<Integer>();
```

But you can’t do it!!

  - This is counter-intuitive at the first glance

Generics and Sub typing (Contd.).

- Why this compile error? It is because if it is allowed, ClassCastException can occur during runtime – this is not type-safe

```
ArrayList<Integer> ai = new ArrayList<Integer>();
ArrayList<Object> ao = ai; // If it is allowed at
  compile time,
ao.add(new Object());
Integer i = ao.get(0); // This would result in
          // runtime ClassCastException
```

- No inheritance relationship between type arguments of a generic class

Generics and Sub typing (Contd.).

- The following code work

```
ArrayList<Integer> ai = new ArrayList<Integer>();
List<Integer> li = new ArrayList<Integer>();
Collection<Integer> ci = new
  ArrayList<Integer>();
Collection<String> cs = new Vector<String>(4);
```

- Inheritance relationship between Generic classes themselves still exist

Generics and Sub typing (Contd.).

- The following code work

```
ArrayList<Number> an = new ArrayList<Number>();
an.add(new Integer(5));
an.add(new Long(1000L));
an.add(new String("hello")); // compile error
```

- The entries maintain inheritance relationship

Why Wildcards?

- To print the contents of the collection following method will not help

```
void printCollection(Collection<Object> c) {
  for (Object o : c)
  System.out.println(o);
  }
```

- It has to be re-written as

```
void printCollection(Collection<?> c) { for
  (Object e : c) { System.out.println(e);
}}
```

Defining Generic class

- Add type parameters in class definition
- Typically use one capital letter for types
- Use anywhere in class that type is required

```
public class Pair<F, S> {
        F first; S second;
        public Pair(F f, S s) {
        first = f; second = s;
        }
        }
```

Defining Generic Classes (Contd.).

```
public class Pair<F, S> {
F first; S second;
public Pair(F f, S s) {
first = f; second = s;
}
public void setFirst(F f){ first = f;}
public F getFirst(){ return first;}
public void setSecond(S s){ second = s; }
public S getSecond(){ return second;}
}
```

Usage of the Generic Classes

```
// Create an instance of Pair <F, S> class.
// Let's call it p1.
Number n1 = new Integer(5);
String s1 = new String("Sun");
Pair<Number,String> p1 =
new Pair<Number,String>(n1, s1);
// Set internal variables of p1.
p1.setFirst(new Long(6L));
p1.setSecond(new String("rises"));
```

Extending the Generic Classes

```
public class PairExtended<F, S, T>
extends Pair<F, S> {
T third;
/** Creates a new instance of PairExtended */
PairExtended(F f, S s, T t){
super(f, s);
third = t;
}
public T getThird(){
return third; } }
```

Usage of the Extended Generic Class

```
// Create an instance of PairExtended<F, S, T>
  class
// with concrete type arguments, <Number, String,
  Integer>
Number n4 = new Long(3000L);
String s4 = new String("james");
Integer i4 = new Integer(7);
PairExtended<Number, String, Integer> pe4
= new PairExtended<Number, String, Integer>(n4, s4,
  i4);
```

Raw Type

- Generic type instantiated with no type arguments
- Pre-J2SE 5.0 classes continue to function over J2SE 5.0 JVM as raw type

```
 // Generic type instantiated with type argument
 List<String> ls = new LinkedList<String>();
// Generic type instantiated with no type
// argument – Raw type
List lraw = new LinkedList();
```

Type Erasure

- Enables Java applications that use generics to maintain binary compatibility with Java libraries and applications that were created before generics
- So generic type information does not exist during runtime
  - During runtime, the class that represents ArrayList<String>, ArrayList<Integer> is the same class that represents ArrayList

Type Erasure Example code: True / False

```
ArrayList<Integer> ai = new ArrayList<Integer>();
ArrayList<String> as = new ArrayList<String>();
Boolean b1 = (ai.getClass() == as.getClass());
System.out.println(" Do ArrayList<Integer> and
  ArrayList<String>
share same class?" + b1);
```

O/P:

```
Do ArrayList<Integer> and ArrayList<String> share same
  class?true
```

Interoperability with pre J2SE 5 code

- For raw type, compiler does not have enough type information (for type checking) so it just generates “unchecked” or “unsafe” warning
- If you ignore them, ClassCastException can still occur during runtime

```
List<String> ls = new LinkedList<String>();
List lraw = ls;
lraw.add(new Integer(4)); // Compiler Warning
String s = ls.iterator().next(); // Runtime error
```

Recommendations

- Do not use raw type whenever possible
- If you have to use raw type or have to use pre-J2SE 5.0 compiled classes or libraries, make sure the “unsafe” compile warnings are really just warnings

AutoBoxing with Collections

- Boxing conversion converts primitive values to objects of corresponding wrapper types

```
// prior to Java 5, Explicit Boxing
int i = 11;
Integer iReference = new Integer(i);
// In Java 5,Automatic Boxing
iReference = i;
```

- Unboxing conversion converts objects of wrapper types to values of corresponding primitive types

```
// prior to Java 5,Explicit Unboxing
int j = iReference.intValue();
j = iReference; // In Java 5, Automatic Unboxing
```

Quiz

1. Which of the following are true regarding Generics?

a. It lets you enforce compile-time safety on collections

b. When using collections, a cast is needed to get elements out of the Collection

c. Both option a and option b are true

d. None of the above are true

Both option a and option b are true

2. Will the following code compile?

```
List<Animal> aList = new ArrayList<Animal>();
```

a. True

b. False

True

Understanding List, Set and Map

The ArrayList Class

- It can grow dynamically
- Array Lists are synchronized
- It provides more powerful insertion and search mechanisms than arrays
- Gives faster Iteration and fast random access
- Ordered Collection (by index), but not Sorted

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

Iterator

- Iterator is an object that enables you to traverse through a collection
- Can be used to remove elements from the collection selectively, if desired

```
public interface Iterator<E>

 {
 boolean hasNext();
 E next();
 void remove();
 }
```

```
ArrayList<Integer> ai=new ArrayList<Integer>();
Iterator i=ai.iterator();
while (i.hasNext())
System.out.println(i.next());
```

ListIterator

- Used for obtaining a iterator for collections that implement List
- ListIterator gives us the ability to access the collection in either forward or backward direction

Advantage of Iterator over for-each method

- for-each construct can also be used for iterating through the Collection
- Use Iterator instead of the for-each construct when you need to:
  - Remove the current element
    - The for-each construct hides the iterator, so you cannot call remove
  - Iterate over multiple collections in parallel

```
for(Object o : oa) {
Fruit d2 = (Fruit)o;
System.out.println(d2.name); }
```

Enhanced for loop

- Iterating over collections looks cluttered

```
void printAll(Collection<emp> e) {
 for (Iterator<emp> i = e.iterator();
  i.hasNext(); )
 System.out.println(i.next()); } }
```

- Using enhanced for loop we can do the same thing as

```
void printAll(Collection<emp> e) {
 for (emp t: e) )
 System.out.println(t); }}
```

- The loop above reads as “for each emp t in e.”

Linked List

- Implements the List interface
- Some Useful Methods
  - void addFirst(Object x)
  - void addLast(Object x)
  - Object getFirst()
  - Object getLast()
  - Object removeFirst()
  - Object removeLast()

The HashSet Class

- No duplicates allowed
- A HashSet is an unsorted, unordered Set
- Can be used when you want a collection with no duplicates and you don't care about the order when you iterate through it
- Uses HashTable for storage

```
Set<Integer> s = new HashSet<Integer>();
ba[0] = s.add(1);
ba[1] = s.add(2);
ba[2] = s.add(3);
```

TreeSet

- No duplicates allowed
- Iterates in sorted order
- Sorted Collection
  - By default elements will be in ascending order
- Not synchronized
  - If more than one thread wants to access it at the same time then it must be synchronized externally

```
TreeSet<String> t1 = new TreeSet<String>();
t1.add("One");
t1.add("Two");
t1.add("Three");
```

The HashMap class

- HashMap uses the hashcode value of an object to determine how the object should be stored in the collection
- Hashcode is used again to help locate the object in the collection
- Gives you an unsorted and unordered Map
- Allows one null key and multiple null values in a collection
- HashMap are not synchronized

```
HashMap<String,Double> hm = new
   HashMap<String,Double>();
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
```

The Hashtable Class

- Part of java.util package
- Implements a hashtable, which maps keys to values
  - Any non-null object can be used as a key or as a value
  - The Objects used as keys must implement the hashcode and the equals method
- Synchronized class

```
Hashtable<String,Double> balance = new
  Hashtable<String,Double>();
balance.put("Arun", new Double(3434.34));
balance.put("Radha", new Double(123.22));
```

Properties

- Extends Hashtable.
- Used to maintain lists of key value pairs in which both the key and the value are Strings
- Useful method

```
//Used to print all the system properties
Properties p=System.getProperties();
p.list(System.out);

//Used to get a system property user.name
System.out.println(p.getProperty("user.name"));
```

TreeMap

- Implements Map interface
- Provides efficient means of storing key/value pairs in sorted order
- Allows rapid retrieval
- Guarantees that its elements will be sorted in ascending key order

The Vector Class

- The java.util.Vector class implements a growable array of Objects
- Same as ArrayList, but Vector methods are synchronized for thread safety
- New java.util.Vector is implemented from List Interface
- Creation of a Vector

  - `Vector v1 = new Vector(); // allows old or new methods`
  - `List v2 = new Vector(); // allows only the new (List) methods.`

Quiz

1.
```
TreeSet map = new TreeSet();
map.add("one");
map.add("two");
map.add("three");
map.add("one");
map.add("four");
Iterator it = map.iterator();
while (it.hasNext() ) {
System.out.print( it.next() + " " );
}
```

four three two one

A. Compilation fails

B. four three two one

C. one two three four

D. four one three two

Quiz (Contd.).

2.
```
public static void before() {
Set set = new TreeSet();
set.add("2");
set.add(3);
set.add("1");
Iterator it = set.iterator();
while (it.hasNext())
System.out.print(it.next() + " ");
}
```

The before() method will not compile

Which of the following statements are true?

A. The before() method will print 1 2

B. The before() method will print 1 2 3

C. The before() method will not compile.

D. The before() method will throw an exception at runtime.

Summary

- In this module, you were able to:
  - Implement various Collection classes and interfaces
  - Explore Generics
  - Apply Best practices in Collections

References

1. Oracle Sun Developer Network (2010). *Introduction to the Collections Framework*. Retrieved on April 9, 2012, from, http://java.sun.com/developer/onlineTraining/collections/Collection.html

2. Campione, M and others. *The Java Tutorial: a Short Course on the Basics. New Delhi:* Addison-Wesley, 2001.

3. Schildt, H. Java: The Complete Reference. J2SETM. Ed 5. New Delhi: McGraw Hill-Osborne, 2005.

Thank You