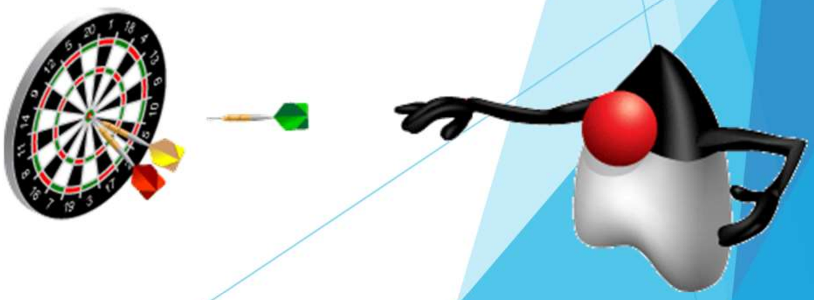
Java Collections

Objectives

At the end of this session, you will be able to

- Explore the java.util package
- Use the collection framework
- Understand the utility classes
- Use StringTokenizer
- Understand and use Generics



Agenda

- Collection Framework
- Interfaces and classes in java.util package
- Utility classes
- StringTokenizer
- Generics
- Enhanced “foreach” loop

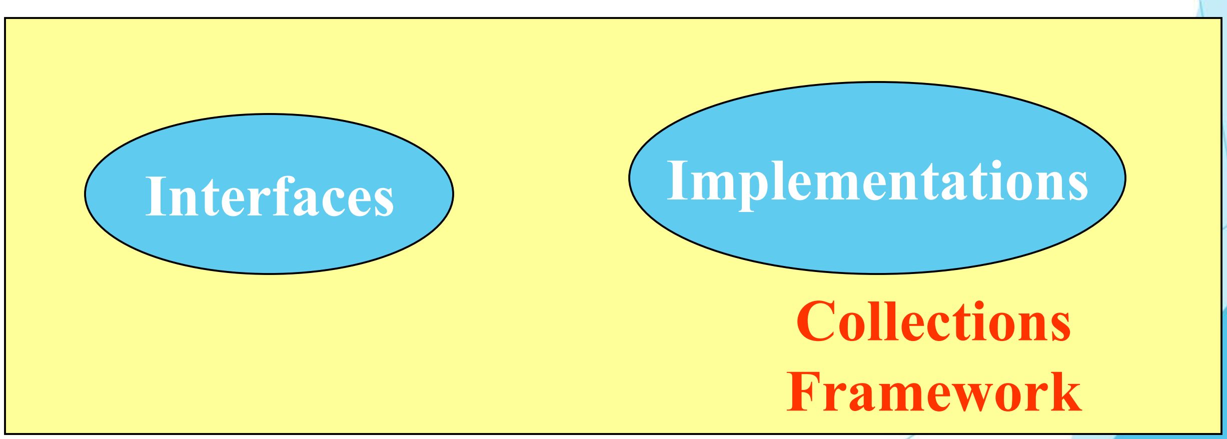
Collections Framework

Collections Framework

- A collection (often called a container) is a group of objects treated as a single entity.

- The collections framework is a unified architecture for representing and manipulating these collections

- The framework consists of:
  - Interfaces
  - Implementations



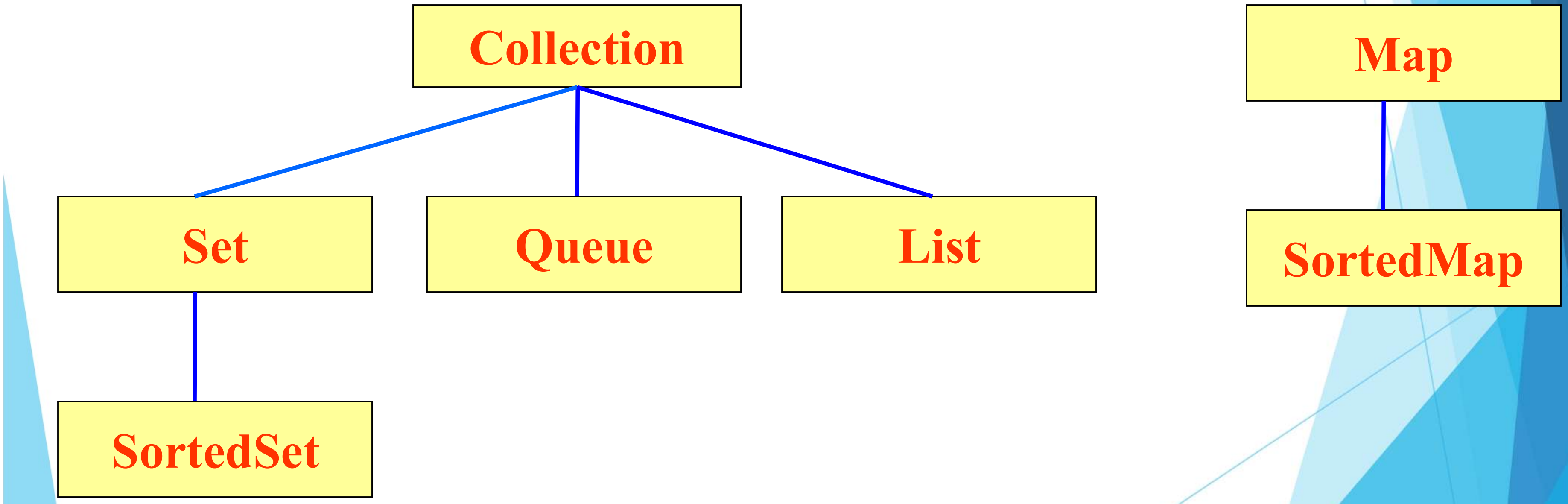
Collections Framework

- Interfaces
  - Can be considered as a contract, upon implementing which a class agrees to become a certain specific collection type.
- Implementations
  - Concrete implementations of the interfaces.
  - Can also be treated as reusable data structures.
  - Are the algorithms / methods that perform the manipulations on these collections.

Collections Framework -Interfaces

Interfaces

- The collections framework has seven core interfaces, with *Collection* interface at the root of the collection hierarchy.
- There are other utility interfaces, viz. Comparator, Iterator, & Enumerator.



Core Interfaces

- **Collection**
  - is the root of collections hierarchy.
  - is the most generalized interface for maintaining collections. It is not directly implemented.
- **List**
  - Extends Collection interface.
  - maintains a sequence of elements.
  - user has precise control over the elements by their indexes.
  - can contain duplicate elements.
  - implemented by ArrayList, Vector, LinkedList.

Core Interfaces

- **Set**
  - models the mathematical set.
  - cannot have duplicate elements.
  - is implemented by HashSet, LinkedHashSet.
- **SortedSet**
  - extends the Set interface.
  - contains elements in ascending order.
  - is implemented by TreeSet.

Core Interfaces

- Map
  - does not extend *Collection* Interface.
  - maps keys to values objects.
  - each key is unique.
  - each key can have atmost one value.
  - is implemented by HashMap, HashTable, LinkedHashMap.
- SortedMap
  - maintains keys in sorted order.
  - is implemented by TreeMap.

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt

List Interface

Important methods in List interface

- boolean add(Object obj), void add(int index,Object obj)
- void clear()
- Object get (int index)
- boolean isEmpty()
- Object remove (int index), boolean remove(Object object)
- Object set(int index, E element)
- int size()
- Object toArray()

Set Interface

Important methods in Set interface

- boolean add (Object o)
- void clear()
- boolean contains (Obeject o)
- Iterator iterator()
- boolean remove(Object o)
- Comparator comparator()

Map Interface

Important methods in Map interface

- Object get(Object Key)
- Object put(Object Key, Object Value)
- Object remove(Object Key)
- int hashCode()
- boolean containsKey(object key)
- boolean containsKey(object value)
- Collection values()

Other Utility Interfaces

- **Comparable**
  - Implemented by all elements in SortedSet and all keys in SortedMap
  - Defines "natural order" for that object class; provides only one sort order
  - Method to be override
    - **public int compareTo(Object o);**
- **Comparator**
  - Meant to be implemented to sort instances of third party classes
  - Allows us to sort a collection in different ways
  - Method to override
    - public int compare(Object a, Object b)

Other Utility Interfaces

- **Enumeration**
  - An Enumeration object generates a series of elements, one at a time.
  - Methods
    - boolean hasMoreElements()
    - Object nextElement()
- **Iterator**
  - provides a way to iterate through any collection.
  - Methods:
    - `boolean hasNext()`
      - returns true if iteration has any more elements
    - `Object next()`
      - returns the next element in the Collection
    - `void remove()`
      - removes the last element returned by the iterator

Collections Framework -Implementations

Implementation

- Implementations are nothing but objects used to store collections.
- Implementations implement one of the core Interfaces to hold collections of specific type.

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

- Classes which implement the core interfaces must provide two constructors:
- A default, no-argument constructor, which creates an empty collection, and
- A constructor which takes a Collection as an argument and creates a new collection with the same elements as the specified collection.

Collections Framework -Algorithms

Algorithms

- The algorithms provided by the collections framework are reusable pieces of functionality.

- These algorithms are static and come from the Collections class.

- The polymorphic nature of the Algorithms enables their operation on a variety of classes, implementing a common interface.

Algorithms

- Sorting
  - reorders (a List) in ascending order
- Searching
  - searches for a specific element using binary search
- Shuffling
  - does the opposite of sorting by destroying the order
- Data Manipulation
  - reverse (reverses the order of elements),
  - fill (reinitializes the List to a specific value),
  - copy (copies elements of one List into another),
  - swap (swaps the elements at specified positions), and
  - addAll (adds specified elements or an array to a Collection).
- Finding Extreme Values
  - returns the min/max values in a Collection.

Collections Framework Important Classes

ArrayList

- An ArrayList is a resizable array implementation of List, thus it's size may not be known beforehand.
- It supports random access of its elements; any element can be accessed in constant time, given only the index of the element.
- ArrayList can hold only objects; however, autoboxing which makes it appear that ArrayList can hold primitives too.
- Constructors
  - ArrayList ()
  - ArrayList (Collection c)
  - ArrayList (int intialSize)

Vector

- Vector is similar to ArrayList, however it is thread safe as its methods are synchronized.
- Vectors are little slow as compared to ArrayList.
- A vector optimizes storage management by maintaining *capacity* and *capacityIncrement*.
- Constructors
  - **`Vector ()`**
  - **`Vector (Collection c)`**
  - **`Vector (int initialCapacity)`**
  - **`Vector (int initialCapacity, int capacityIncrement)`**

Hashtable

- Hashtable, implement *Map interface*, and are associative arrays with key-value pairings.
- The keys are unique but unordered.
- Both key and value should be objects; primitives should be wrapped before they can be used.
- Constructors
  - **`Hashtable ()`**
  - **`Hashtable(int initialCapacity)`**
  - **`Hashtable(int initialCapacity, float loadFactor)`**
  - **`Hashtable(Map map)`**

HashMap

- A HashMap is similar to Hashtable, except that:
  - it can store nulls (both keys and values)
  - it's methods are not synchronized.
- The keys are unique but unordered.
- Constructors
  - `HashMap()`
  - `HashMap(int initialCapacity)`
  - `HashMap(int initialCapacity, float loadFactor)`
  - `HashMap(Map map)`

TreeMap

- TreeMap, implements *SortedMap* interface, and are also associative arrays having key-value pair.
- A TreeMap consists of unique keys in sorted (ascending) order.
- Constructors
  - **`TreeMap()`**
  - **`TreeMap(Comparator c)`**
  - **`TreeMap(Map m)`**
  - **`TreeMap(SortedMap m)`**

HashSet

- A HashSet implements *Set* interface.
- It holds unique elements but unordered.
- Its methods are unsynchronized.
- Constructors
  - **`HashSet()`**
  - **`HashSet(Collection c)`**
  - **`HashSet(int initialCapacity)`**
  - **`HashSet(int initialCapacity, float loadFactor)`**

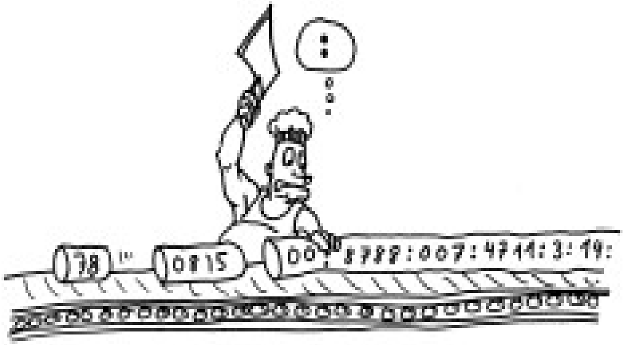
TreeSet

- A TreeSet implements *Set interface*.
- It has unique and ordered elements.
- Its methods are unsynchronized.
- <u>Constructors</u>
  - **`TreeSet()`**
  - **`TreeSet(Collection c)`**
  - **`TreeSet(Comparator c)`**
  - **`TreeSet(SortedSet s)`**

String Tokenizer

String Tokenizer

- Text processing often requires a *string* to be *parsed* into *tokens*.
- The StringTokenizer class allows a string to be broken down into tokens, and is hence called *lexical analyzer*.
- It implements the Enumeration interface, and thus can enumerate individ ed by delimiters.



String Tokenizer

- StringTokenizer Constructors
  - **`StringTokenizer (String str)`**

    String str is tokenized using the default delimiters (*space*,*tab*, *newline* and *carriage return*).
  - **`StringTokenizer (String str, String delim)`**

    The delimiters are specified as the second argument.
  - **`StringTokenizer (String str, String delim, boolean retDelims)`**

    If retDelims is set to true, the delimiters are also return along with the tokens.

String Tokenizer

- Important Methods
  - **`int countTokens()`**

    returns the number of tokens remaining
  - **`boolean hasMoreTokens() or boolean hasMoreElements`**

    returns true if anymore tokens are remaining
  - **`String nextToken()`**

    returns the next token
  - **`Object nextElement()`**

    returns the next token as an *Object*

Try it out

Q1. Collection is a ______ (class/interface) and Collections is a ________ (class/interface)

Q2. ________ provides fast iteration and fast random access

a. HashMap b. ArrayList c. TreeSet

Q3. Which implementation of the List interface provides for the fastest insertion of a new element into the middle of the list?

a. Vector b. ArrayList c. LinkedList

Q4. ______ allows one null key and many null values

a. Hashtable b. HashMap

Generics

- Introduced in JDK 1.5
- Since Java 5, collections should be used only with generics
- Using Generics the Collection classes can be aware of the types they store
- A generic is a method that is recompiled with different types as the need arises
- Generics replaces runtime type checks with compile-time checks

Without Generics

- Elements of different types can be added to a collection

```
List list = new LinkedList();
list.add("foo");
list.add(7);
```

- Type casting is needed while retrieving the element, sometimes result in wrong casting

```
for (int i = 0; i < list.size(); i++) {
  String s = (String)list.get(i);
}
```

Run time error

Explicit type casting

With Generics

- No explicit type casting
- The compiler can now check the type correctness of the program at compile-time
- Improved readability & robustness

```
List<String> list = new LinkedList<String>();
list.add("foo");
list.add(7);
```

Compile time error

```
for (int i = 0; i < list.size(); i++) {
  String s = list.get(i);
}
```

No explicit type casting

Generics

- To iterate over generic collections, it’s a good idea to use a generic iterator

```
List<String> listOfStrings = new LinkedList<String>();
...
...
Iterator<String> i = listOfStrings.iterator();

while(i.hasNext()) {
    String s = i.next();
    System.out.println(s);
}
```

Writing your own generic types

-
  ```
  public class Box<T> {
      private List<T> contents;

      public Box() {
          contents = new ArrayList<T>();
      }

      public void add(T thing) { contents.add(thing); }

      public T grab() {
          if (contents.size() > 0) return contents.remove(0);
          else return null;
  }
  ```
- Box<String> myBox = new Box<String>();
- myBox.add(“Pen”);
- It is recommended to use single capital letters (such as T) for types

Enhanced for loop - *“foreach”*

- A new *foreach* loop is introduced to simplify the iteration process for the collections
- The for-each loop
  - allows to iterate over an array or a collection without using an index or an iterator
  - is less error-prone
  - combines nicely with generics

Enhanced for loop - *“foreach”*

- Consider the following example

```
Integer[] numArray = new Integer[…];
List numList = new LinkedList();
…
int sum = 0;
for (int i = 0; i < numArray.length; i++) {
   Integer n = numArray[i];
   sum += n.intValue();
}

for (Iterator i = numList.iterator(); i.hasNext(); ) {
   Integer n = (Integer)i.next();
   sum += n.intValue();
}
```

Enhanced for loop - “*foreach*”

- You can use following syntax

```
Integer[] numbers = new Integer[…];
List<Integer> numbers = new
LinkedList<Integer>();
…
int sum = 0;
for (Integer n : numbers)
        sum += n.intValue();
```

Enhanced for loop - *“foreach”*

- Loop Syntax

  for (*declaration* : *expression*)

  statements

- *expression* must be either an array or an object that implements the interface java.lang.Iterable (which guarantees the existence of an iterator)
- *declaration* declares the loop variable to which the elements of the array or the iterable object are assigned
- *statement* contains the code that is executed for each element of the array or the iterable object

Enhanced for loop - *“foreach”*

- The for-each loop cannot be used
  - when the position of an element is needed
  - for removing or replacing elements in a collection
  - for loops that must iterate over multiple collections in parallel

Summary

In this session, we have covered:

- Collection Framework
- Interfaces and classes in java.util package
- Utility classes
- StringTokenizer
- Generics
- Enhanced “foreach” loop

Thank you