Java Course
Lecture 7 - Generics and collections

IT Learning &
Outsourcing Center

www.pragmatic.bg

Summary

- Generics
- Iterable
- Iterator
- Collection
- List
- Set
- Map

Generics Basics

- At its core, the term *generics* means *parameterised types*
- Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, interface, or method that operates on a parameterised type is called *generic*, as in *generic class* or *generic method*

- *Why not just use class Object ?*

Generics Example


```
public class Generic<T> {

    private T field;

    public Generic(T field) {
        this.field = field;
    }

    public T getField() {
        return field;
    }

    public void setField(T field) {
        this.field = field;
    }

    public void printGenericType(){
        System.out.println("Generic Filed is of type  " + field.getClass().getName());
    }

    public static void main(String[] args) {
        Generic<String> str = new Generic<String>("This is my field");
        str.printGenericType();

        Generic<Number> num = new Generic<Number>(5D);
        num.printGenericType();

    }
```

Syntax

- Generic Class

accessor class *ClassName<GenericDeclaration>*

```
public class Generic<T>
```

- *Generic Method*

accessor *<GenericDeclaration>* *returnType name()*

```
public <P> P showType(P arg)
```

Bounded types

- Sometimes we wish to restrict the type of the generic. We want to tell the compiler some how that we want to pass in only a family of classes and not all possible types of classes
- To do so we use the keyword *extends*
- *Example*

```
SomeClass<T extends Number>
```

WildCard

- The wildcard argument is specified by the **?**, and it represents an unknown type

* Note : You use the wild card when using the generic not when defining it, meaning when creating a generic variable of some sort !

- Example:

```
SomeClass<? extends Number>

SomeClass<? super Number>
```

Erasure

- Java generics are erasure type
  - meaning that they exist only at compile time

after which they are "erased"

Erasure types were created to preserve backward compatibility of the java bytecode. Old non generic programs need to be able to run on new modern virtual machines

Generic Restrictions

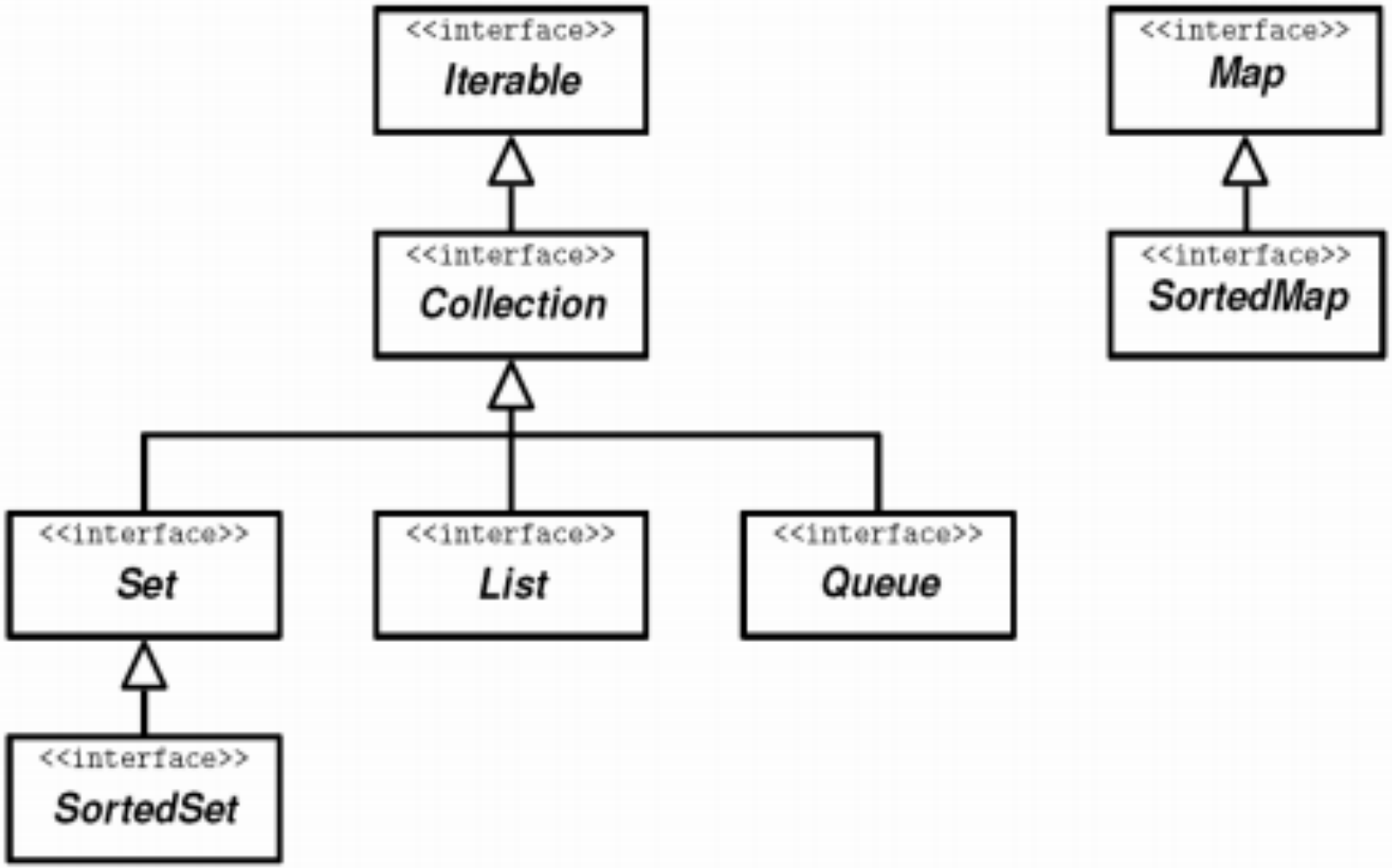
- Type Parameters Can't Be Instantiated

- No **static** member can use a type parameter declared by the enclosing class.

- You can still create generic static methods, but they must specify their own generic type

- You can't instantiate an array of elements from the generic type

Collections Hierarchy



Terminology

- Unmodifiable – collections that do not support any modification operations (such as add, remove and clear)
- Modifiable – collections that are not unmodifiable
- Immutable – unmodifiable collections that additionally guarantee that no change in the Collection object will ever be visible

Terminology

- Fixed-size – lists that guarantee that their size remains constant even though the elements may change
- Variable-size – lists that are not fixed size.
- Random access – lists that support fast (generally constant time) indexed element access

Iterators

- interface Iterator<E>:
  - Represents a position in a collection
  - Also allows the user to remove elements from the backing collection with well defined, useful semantics
- interface ListIterator<E> extends Iterator<E>:
  - Iterator for use with lists
  - Supports bi-directional iteration, element replacement, element insertion and index retrieval

Iterable Interface

- public interface Iterable<T>
  - Implementing this interface allows an object to be the target of the new **for** statement (also called for-each or for-in)
- Has only one method:
  - Iterator<T> iterator() – Returns an iterator over a set of elements of type T
- Usually implemented indirectly trough the Collection interface

Iterator

- *public* interface Iterator<E>

- An iterator over a collection
- Think of it like a cursor pointing to an element in a collection

Iterator Interface

- Has 3 methods:
  - boolean hasNext()
    - Returns true if the iteration has more elements
  - E next()
    - Returns the next element in the iteration.
  - void remove() (optional)
    - Removes from the underlying collection the current element
    - Can be called only once per call to next
- *The underlying collection should not be modified during iteration! Unless you use iterator ! ☺*

Collection Interface

- **public** *interface* Collection<E> *extends* Iterable<E>
- The root interface in the collection hierarchy
- A collection represents a group of objects, known as its elements
- Some collections allow duplicate elements and others do not
- Some are ordered and others unordered

Collection Interface

- The JDK does not provide any direct implementations of this interface
- Used to pass collections around and manipulate them where maximum generality is desired
- Bags or multi-sets (unordered collections that may contain duplicate elements) should implement this interface directly

Collection Interface

- All general-purpose Collection implementation classes should provide two ”standard” constructors:
  - a void (no arguments) constructor, which creates an empty collection
  - a constructor with a single argument of type Collection which creates a new collection with the same elements as its argument

Collection Interface

- All of the modification methods in the collection interfaces are labeled optional
- Some implementations may not perform one or more of these operations, throwing a runtime exception (UnsupportedOperationException)

Collection Interface

- Some of its methods:

  boolean add(E o) (optional)

  - Ensures that this collection contains the specified element

  boolean contains(Object o)

  - Returns true if this collection contains the specified element

  boolean remove(Object o) (optional)

  - Removes a single instance of the specified element from this collection, if it is present

Collection Interface

- Iterator<E> iterator()
  - Returns an iterator over the elements in this collection
- int size()
  - Returns the number of elements in this collection
- boolean isEmpty()
  - Returns true if this collection contains no elements

List Interface

- An ordered collection (also known as a sequence)
- Supports element access integer index
- Supports search for elements in the list
- List indices (like in Java arrays) are zero based
- Typically allows duplicate elements.
- Typically allows *null* elements.
- Provides a subList(from, to) view

List Interface

- Provides a special iterator – ListIterator allowing:
  - insertion
  - replacement
  - bidirectional access
- Iterating is preferable than indexing - it is fast on all implementations
- Searching methods should be used with caution – in many implementations they will perform costly linear searches

Set Interface

- A collection that contains no duplicate elements
- If an element is changed in a manner that affects equals comparisons, the behaviour of a set is not specified
- Great care must be exercised if mutable objects are used as set elements

Set Interface

- Some implementations prohibit null elements
- Some have restrictions on the types of their elements
- Trying to violate the restrictions may cause NullPointerException or ClassCastException

Map Interface

public interface Map<K,V>

- Does not implement the Collection interface
- Maps keys to values
- A map cannot contain duplicate keys
- Great care must be exercised if mutable objects are used as map keys

Map Interface

- Provides three collection views, which allow a map's contents to be viewed as:
  - a set of keys
  - a collection of values
  - a set of key-value mappings

Map.Member

- **static interface** Map.Entry<K,V>
- A map entry (key-value pair). Contains:
  - K getKey()
  - V getValue()
  - V setValue(V value) (optional)
  - int hashCode()
  - boolean equals(Object o)

Map operations

- Basic operations:
  - V get(Object key)
  - V put(K key,V value) (optional)
  - V remove(Object key) (optional)
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)

Map properties

- Collection views:
  - Set<Map.Entry<K,V>> entrySet()
- Returns a set view of the mappings contained in this map
  - Set<K> keySet()
- Returns a set view of the keys contained in this map
  - Collection<V> values()
- Returns a collection view of the values contained in this map

Queue Interface

- A collection designed for holding elements prior to processing
- Typically order elements in a FIFO (first-in-first-out) manner
- Has lots of implementations in the JDK
- Method names are not very usual

Collection Implementations

- These are the default collection implementations in the Java API:

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

Collection Classes

- The general-purpose implementations support all of the optional operations in the collection interfaces
- They have no restrictions on the elements they may contain
- They are unsynchronized
- Static factories from the Collections class can be used to make synchronization wrappers
- Have fail-fast iterators, which detect illegal concurrent modification

Search Implementations

- HashMap and HashSet
  - Implemented as hash tables
  - Provides fast search, insert and delete operations
  - Some operations like containsValue() are dependent on the amount of elements in the collection
  - When using make sure that equals() and hashCode() are implemented correctly

Ordered Implementations

- TreeSet and TreeMap
  - Usually implemented with red-black trees
  - Implement SortedSet, SortedMap interfaces
  - Store the elements sorted in ascending order
  - The elements are ordered by their natural order (Comparable) or by a Comparator given to the constructor

Resizable Array

- public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable ,Serializable
- ArrayList provides the operations provided by the java arrays
- Unlike the java arrays its not fixed-size

Resizable array

- The size, isEmpty, get, set, iterator, and listIterator operations run in constant time
- The add operation runs in amortized constant time – adding N elements requires O(N) time

Linked List

```
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>,
Queue<E>, Cloneable, Serializable
```

- Linked list implementation of the List interface
- Implements all optional list operations
- Provides uniformly named methods to get, remove and insert an element at the beginning and end of the list

Linked List (part 2)

- Can be used as a stack, queue, or double-ended queue (dequeue)
- Perform as could be expected for a doubly-linked list
- Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index
- Use ArrayList when insertion and deletion from the middle of the list is rare

java.util.Collections

- Utility class filled with "goodies".
- Collections.emptySet(),
  Collections.emptyList and
  Collections.emptyMap
  - Returning an the empty set, list and map immutable)

java.util.Collections

- Collections.singleton(T o),
  Collections.singletonList(T o),
  Collections.singletonMap(K key,V value)
- Returns an immutable singleton (set, list, or map, containing only the specified object or key-value mapping)
- Collections.nCopies(int n,T o) Returns an immutable list consisting of copies of specified object

Algorithms

- Java.util.Arrays
  - Contains useful methods that go well along with arrays
- java.util.Collections
  - contains set of polymorphic algorithms that operate on collections

Algorithms - Reordering

- sort(List)
  - Sorts list. Guaranteed fast performance and stability
- reverse(List)
  - Reverses the order of the elements in the list
- shuffle(List)
  - Randomly permutes the elements in list
- rotate(List list, int distance)
  - Rotates all of the elements in the list by the specified distance
- swap(List, int, int)
  - Swaps the elements at the specified positions in the specified list

Algorithm - Queries

- binarySearch(List, Object)
  - Searches for an element in an ordered list using the binary search algorithm
- min(Collection)
  - Returns the minimum element in collection
- max(Collection)
  - Returns the maximum element in collection
- frequency(Collection, Object)
  - Counts the number of times the specified element occurs in the specified collection

Algorithms - Modification

- fill(List, Object)
  - Overwrites every element in list with the specified value
- replaceAll(List list, Object oldVal, Object newVal)
  - Replaces all occurrences of one specified value with another

Algorithms – Collection transformations

- copy(List dest, List src)
  - Copies the source list into the destination list
- indexOfSubList(List source, List target)
  - Returns the index of the first sub list of source that is equal to target
- lastIndexOfSubList(List source, List target)
  - Returns the index of the last sub list of source that is equal to target

Algorithms – Collection transformations

- disjoint(Collection, Collection)
  - Determines whether two collections are disjoint, in other words, whether they contain no elements in common
- addAll(Collection<? super T>, T...)
  - Adds all of the elements in the specified array to the specified collection.

Q and A ?

Problems

- Does the Collection interface provides methods for adding an element?
- What is an unmodifiable collection?
- What is an immutable collection?
- What is a random access list?
- What is a fixed size list?
- Write a program for finding all different words in a given text and to display them in alphabetical order. What collection class will you use: Hashtable, HashMap, TreeMap, HashSet or TreeSet?