GENERICS:

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array or an array of any type that supports ordering.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods:

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:

* All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type ( < E > in the next example).
* Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
* The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
* A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example:

Following example illustrates how we can print array of different type using a single Generic method:

public class GenericMethodTest

{

// generic method printArray

public static < E > void printArray( E[] inputArray )

{

// Display array elements

for ( E element : inputArray ){

System.out.printf( "%s ", element );

}

System.out.println();

}

public static void main( String args[] )

{

// Create arrays of Integer, Double and Character

Integer[] intArray = { 1, 2, 3, 4, 5 };

Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };

Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

System.out.println( "Array integerArray contains:" );

printArray( intArray ); // pass an Integer array

System.out.println( "\nArray doubleArray contains:" );

printArray( doubleArray ); // pass a Double array

System.out.println( "\nArray characterArray contains:" );

printArray( charArray ); // pass a Character array

}

}

Bounded Type Parameters:

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example:

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:

public class MaximumTest

{

// determines the largest of three Comparable objects

public static <T extends Comparable<T>> T maximum(T x, T y, T z)

{

T max = x; // assume x is initially the largest

if ( y.compareTo( max ) > 0 ){

max = y; // y is the largest so far

}

if ( z.compareTo( max ) > 0 ){

max = z; // z is the largest now

}

return max; // returns the largest object

}

public static void main( String args[] )

{

System.out.printf( "Max of %d, %d and %d is %d\n\n",

3, 4, 5, maximum( 3, 4, 5 ) );

System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",

6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

System.out.printf( "Max of %s, %s and %s is %s\n","pear",

"apple", "orange", maximum( "pear", "apple", "orange" ) );

}

}

Generic Classes:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example:

Following example illustrates how we can define a generic class:

public class Box<T> {

private T t;

public void add(T t) {

this.t = t;

}

public T get() {

return t;

}

public static void main(String[] args) {

Box<Integer> integerBox = new Box<Integer>();

Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));

stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());

System.out.printf("String Value :%s\n", stringBox.get());

}

}

This would produce the following result:

Integer Value :10

String Value :Hello World

COLLECTIONS:

The collections framework was designed to meet several goals.

* The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient.
* The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
* Extending and/or adapting a collection had to be easy.

Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList, HashSet,** and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

* **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations, i.e., Classes:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

The Collection Interfaces:

The collections framework defines several interfaces. This section provides an overview of each interface:

|  |  |
| --- | --- |
| **SN** | **Interfaces with Description** |
| 1 | [The Collection Interface](http://www.tutorialspoint.com/java/java_collection_interface.htm) This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | [The List Interface](http://www.tutorialspoint.com/java/java_list_interface.htm) This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | [The Set](http://www.tutorialspoint.com/java/java_set_interface.htm) This extends Collection to handle sets, which must contain unique elements |
| 4 | [The SortedSet](http://www.tutorialspoint.com/java/java_sortedset_interface.htm) This extends Set to handle sorted sets |
| 5 | [The Map](http://www.tutorialspoint.com/java/java_map_interface.htm) This maps unique keys to values. |
| 6 | [The Map.Entry](http://www.tutorialspoint.com/java/java_mapentry_interface.htm) This describes an element (a key/value pair) in a map. This is an inner class of Map. |
| 7 | [The SortedMap](http://www.tutorialspoint.com/java/java_sortedmap_interface.htm) This extends Map so that the keys are maintained in ascending order. |
| 8 | [The Enumeration](http://www.tutorialspoint.com/java/java_enumeration_interface.htm) This is legacy interface and defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. |

The Collection Classes:

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table:

|  |  |
| --- | --- |
| **SN** | **Classes with Description** |
| 1 | **AbstractCollection** Implements most of the Collection interface. |
| 2 | **AbstractList** Extends AbstractCollection and implements most of the List interface. |
| 3 | **AbstractSequentialList** Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. |
| 4 | [LinkedList](http://www.tutorialspoint.com/java/java_linkedlist_class.htm) Implements a linked list by extending AbstractSequentialList. |
| 5 | [ArrayList](http://www.tutorialspoint.com/java/java_arraylist_class.htm) Implements a dynamic array by extending AbstractList. |
| 6 | **AbstractSet** Extends AbstractCollection and implements most of the Set interface. |
| 7 | [HashSet](http://www.tutorialspoint.com/java/java_hashset_class.htm) Extends AbstractSet for use with a hash table. |
| 8 | [LinkedHashSet](http://www.tutorialspoint.com/java/java_linkedhashset_class.htm) Extends HashSet to allow insertion-order iterations. |
| 9 | [TreeSet](http://www.tutorialspoint.com/java/java_treeset_class.htm) Implements a set stored in a tree. Extends AbstractSet. |
| 10 | **AbstractMap** Implements most of the Map interface. |
| 11 | [HashMap](http://www.tutorialspoint.com/java/java_hashmap_class.htm) Extends AbstractMap to use a hash table. |
| 12 | [TreeMap](http://www.tutorialspoint.com/java/java_treemap_class.htm) Extends AbstractMap to use a tree. |
| 13 | [WeakHashMap](http://www.tutorialspoint.com/java/java_weakhashmap_class.htm) Extends AbstractMap to use a hash table with weak keys. |
| 14 | [LinkedHashMap](http://www.tutorialspoint.com/java/java_linkedhashmap_class.htm) Extends HashMap to allow insertion-order iterations. |
| 15 | [IdentityHashMap](http://www.tutorialspoint.com/java/java_identityhashmap_class.htm) Extends AbstractMap and uses reference equality when comparing documents. |

The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

The following legacy classes defined by java.util have been discussed in previous tutorial:

|  |  |
| --- | --- |
| **SN** | **Classes with Description** |
| 1 | [Vector](http://www.tutorialspoint.com/java/java_vector_class.htm) This implements a dynamic array. It is similar to ArrayList, but with some differences. |
| 2 | [Stack](http://www.tutorialspoint.com/java/java_stack_class.htm) Stack is a subclass of Vector that implements a standard last-in, first-out stack. |
| 3 | [Dictionary](http://www.tutorialspoint.com/java/java_dictionary_class.htm) Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. |
| 4 | [Hashtable](http://www.tutorialspoint.com/java/java_hashtable_class.htm) Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. |
| 5 | [Properties](http://www.tutorialspoint.com/java/java_properties_class.htm) Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String. |
| 6 | [BitSet](http://www.tutorialspoint.com/java/java_bitset_class.htm) A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed. |

The Collection Algorithms:

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

Collections define three static variables: EMPTY\_SET, EMPTY\_LIST, and EMPTY\_MAP. All are immutable.

|  |  |
| --- | --- |
| **SN** | **Algorithms with Description** |
| 1 | [The Collection Algorithms](http://www.tutorialspoint.com/java/java_collection_algorithms.htm) Here is a list of all the algorithm implementation. |

How to use an Iterator ?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.

|  |  |
| --- | --- |
| **SN** | **Iterator Methods with Description** |
| 1 | [Using Java Iterator](http://www.tutorialspoint.com/java/java_using_iterator.htm) Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces. |

How to use a Comparator ?

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Also this interface can be used to sort any instances of any class (even classes we cannot modify).

|  |  |
| --- | --- |
| **SN** | **Iterator Methods with Description** |
| 1 | [Using Java Comparator](http://www.tutorialspoint.com/java/java_using_comparator.htm) Here is a list of all the methods with examples provided by Comparator Interface. |

Collection Interface:

The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. These methods are summarized in the following table.

Because all collections implement Collection, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an**UnsupportedOperationException**.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **boolean add(Object obj)** Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates. |
| 2 | **boolean addAll(Collection c)** Adds all the elements of c to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false. |
| 3 | **void clear( )** Removes all elements from the invoking collection. |
| 4 | **boolean contains(Object obj)** Returns true if obj is an element of the invoking collection. Otherwise, returns false. |
| 5 | **boolean containsAll(Collection c)** Returns true if the invoking collection contains all elements of c. Otherwise, returns false. |
| 6 | **boolean equals(Object obj)** Returns true if the invoking collection and obj are equal. Otherwise, returns false. |
| 7 | **int hashCode( )** Returns the hash code for the invoking collection. |
| 8 | **boolean isEmpty( )** Returns true if the invoking collection is empty. Otherwise, returns false. |
| 9 | **Iterator iterator( )** Returns an iterator for the invoking collection. |
| 10 | **boolean remove(Object obj)** Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false. |
| 11 | **boolean removeAll(Collection c)** Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false. |
| 12 | **boolean retainAll(Collection c)** Removes all elements from the invoking collection except those in c. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false |
| 13 | **int size( )** Returns the number of elements held in the invoking collection. |
| 14 | **Object[ ] toArray( )** Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements. |
| 15 | **Object[ ] toArray(Object array[ ])** Returns an array containing only those collection elements whose type matches that of array. |

Example:

Following is the example to explain few methods from various class implementations of the above collection methods:

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

List a1 = new ArrayList();

a1.add("Zara");

a1.add("Mahnaz");

a1.add("Ayan");

System.out.println(" ArrayList Elements");

System.out.print("\t" + a1);

List l1 = new LinkedList();

l1.add("Zara");

l1.add("Mahnaz");

l1.add("Ayan");

System.out.println();

System.out.println(" LinkedList Elements");

System.out.print("\t" + l1);

Set s1 = new HashSet();

s1.add("Zara");

s1.add("Mahnaz");

s1.add("Ayan");

System.out.println();

System.out.println(" Set Elements");

System.out.print("\t" + s1);

Map m1 = new HashMap();

m1.put("Zara", "8");

m1.put("Mahnaz", "31");

m1.put("Ayan", "12");

m1.put("Daisy", "14");

System.out.println();

System.out.println(" Map Elements");

System.out.print("\t" + m1);

}

}

This would produce the following result:

ArrayList Elements

[Zara, Mahnaz, Ayan]

LinkedList Elements

[Zara, Mahnaz, Ayan]

Set Elements

[Zara, Mahnaz, Ayan]

Map Elements

{Mahnaz=31, Ayan=12, Daisy=14, Zara=8}