Collections

Tanjina Helaly

Collection

- Collection (sometimes called a container) is an object that holds other objects that are accessed, placed, and maintained under some set of rules.
- Examples
  - Sets
  - List
  - Map

Collection vs. Array

- Array
  - fixed size.
  - Type safe.

```
Integer[] data = new Integer[2];
data[0] =5;
data[1] = “Hello”;// error
```

  - Can store primitive type.
  - Better performance but
    - If we need to increase or decrease the size of an array – it become inefficient.
      - Need to declare a new array
      - Copy the element from old array to the new one.
  - It is not always feasible to know how big an array will be needed for an application.

Collection vs. Array

- Collection
  - dynamic array. No size limitation
  - Need explicit casting while retrieving data.
  - Not type safe.

```
Collection data = new ArrayList();
data.add(5); // OK, will auto boxing
data.add("Hello");// OK
```

  - Have methods that perform useful computations, such as searching and sorting, on objects.

Collection Framework

- A collections framework is a unified architecture for representing and manipulating **collections** in your programs.
  - The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.

- Collections were not part of the original Java release, but were added by J2SE 1.2.
- Prior to the Collections Framework, Java provided ad hoc classes such as **Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects.**

Collection Framework-Components

- 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.
  - **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.

Collection Framework – Goal/Benefits

- The Collections Framework was designed to meet several goals.
  - First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
  - Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
    - Reduces programming effort.
    - Reduces effort to learn and to use new API
  - Third, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is built upon a set of standard interfaces.
    - Reduces effort to design new APIs
    - Several standard implementations (such as **LinkedList, HashSet, and TreeSet) of these interfaces** are provided that you may use as-is.

Collection Framework – Goal/Benefits

- *Algorithms are another important part of the collection mechanism.*
  - *Algorithms operate* on collections and are defined as static methods within the **Collections class.**
    - **Thus, they are** available for all collections.
    - Each collection class need not implement its own versions.
    - The algorithms provide a standard means of manipulating collections.

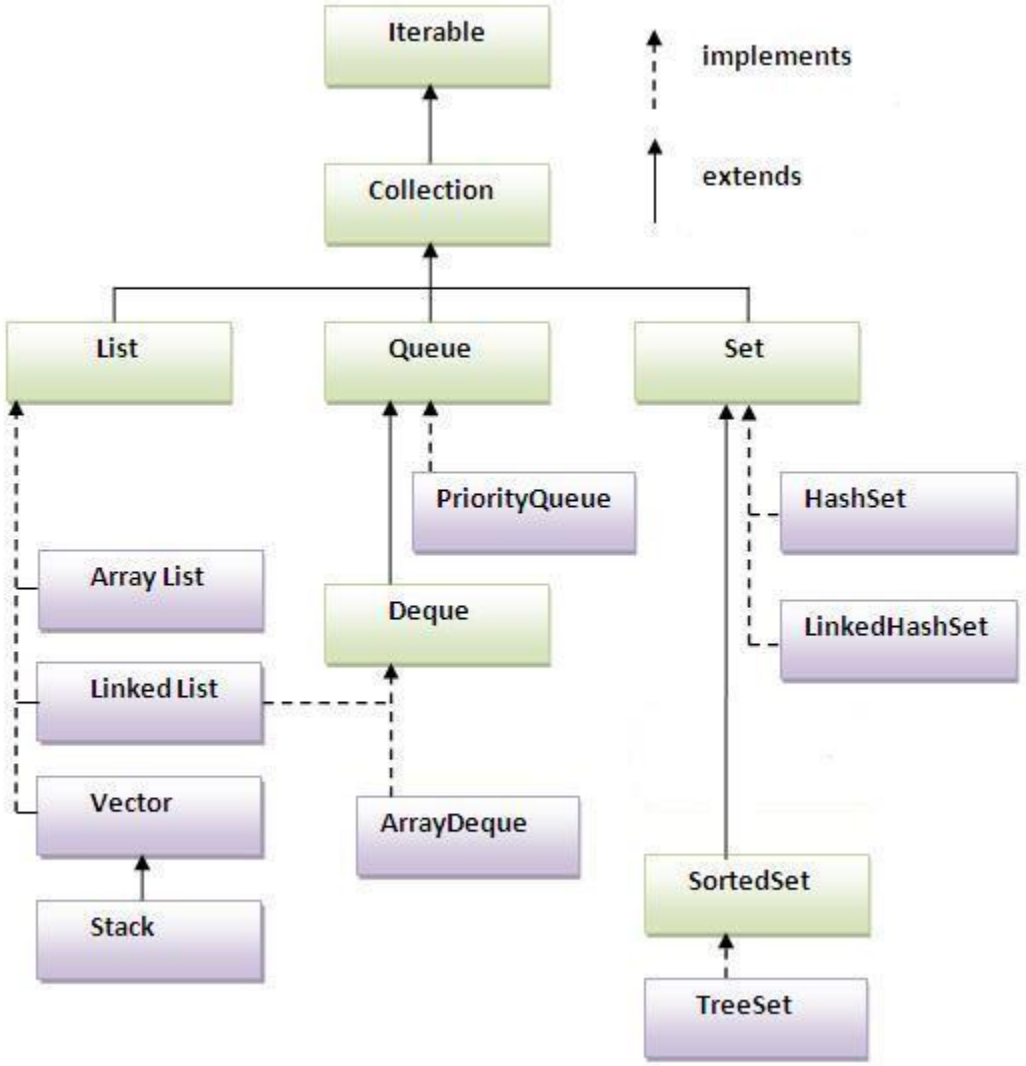
Iterator

- Is an item closely associated with the Collections Framework
- 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
  - It *offers a general-purpose, standardized way of accessing the elements within a* collection, one at a time.
  - Iterator provides the facility of iterating the elements in forward direction only.
- There are only three methods in the Iterator interface. They are:
  - **public boolean hasNext()** it returns true if iterator has more elements.
  - **public object next()** it returns the element and moves the cursor pointer to the next element.
  - **public void remove()** it removes the last elements returned by the iterator. It is rarely used.

Collection Interfaces

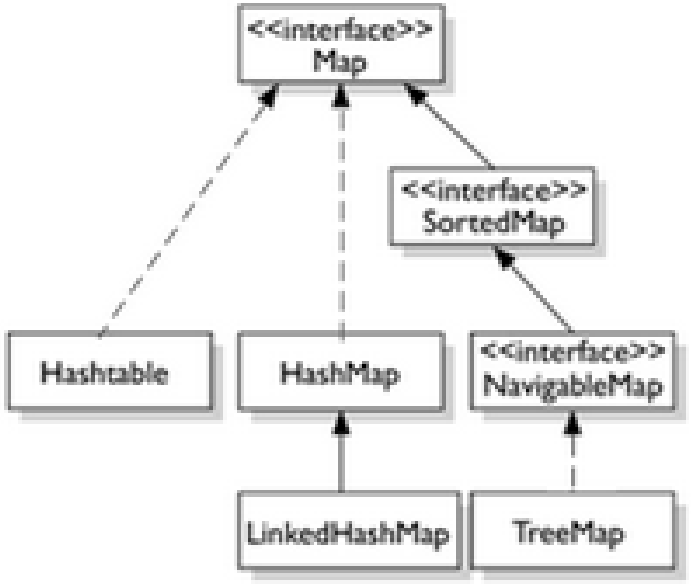
- The *collection interfaces* are divided into two groups. The most basic interface, java.util.Collection.
- The other collection interfaces are based on java.util.Map
  - Map are not true collections.
  - However, these interfaces contain *collection-view* operations, which enable them to be manipulated as collections.

Hierarchy of Collection Framework



Map – not true collection

- A Map
  - is an object that maps keys to values
  - cannot contain duplicate keys:
  - Each key can map to at most one value.



Methods of Collection interface

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element from the collection.
8	public boolean contains(Object element)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collection.
14	public int hashCode()	returns the hashcode number for collection.

ArrayList

- Java ArrayList class uses a dynamic array for storing the elements.It extends AbstractList class and implements List interface.
- can contain duplicate elements.
- maintains insertion order.
- non synchronized.
- allows random access because array works at the index basis.
- manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

ArrayList

- Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

- **Non-Generic:**
  - Can hold any type of object
  - Example:
    - ArrayList al=**new** ArrayList(); //creating non-generic arraylist

- **Generic:**
  - allows you to have only one type of object in collection
  - the type is specified in angular braces.
  - is forced to have only specified type of objects in it.
  - If you try to add another type of object, it gives *compile time error*.
  - Example:
    - ArrayList<String> al=**new** ArrayList<String>(); //creating generic arraylist

ArrayList - Example

```
import java.util.*;
class TestCollection1{
    public static void main(String args[]){
        ArrayList<String> al=new ArrayList<String>();//creating arraylist
        al.add("Ravi");//adding object in arraylist
        al.add("Vijay");
        al.add("Ravi");  // duplicate object
        al.add("Ajay");

       //getting Iterator from arraylist to traverse elements .
       // Can also use for or enhanced for loop to access the item
        Iterator itr=al.iterator();
         while(itr.hasNext()){
           System.out.println(itr.next());
        }
    }
}
```

- Output

```
Ravi
Vijay
Ravi
Ajay
```

ArrayList – Some Methods

Method	Description
`boolean add(E e)`	Appends the specified element to the end of this list.
`void add(int index, E element)`	Inserts the specified element at the specified position in this list.
`boolean addAll(Collection<? extends E> c)`	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
`void clear()`	Removes all of the elements from this list.
`boolean contains(Object o)`	Returns `true` if this list contains the specified element.
`E get(int index)`	Returns the element at the specified position in this list.
`int indexOf(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean isEmpty()`	Returns `true` if this list contains no elements.
`Iterator<E> iterator()`	Returns an iterator over the elements in this list in proper sequence.
`E remove(int index)`	Removes the element at the specified position in this list.
`boolean remove(Object o)`	Removes the first occurrence of the specified element from this list, if it is present.
`E set(int index, E element)`	Replaces the element at the specified position in this list with the specified element.
`int size()`	Returns the number of elements in this list.
`Object[] toArray()`	Returns an array containing all of the elements in this list in proper sequence (from first to last element).

HashSet

- It extends AbstractSet class and implements Set interface.
- contains unique elements only.
- non synchronized.
- HashSet allows null values however if you insert more than one nulls it would still return only one null value.
- HashSet doesn’t maintain any order, the elements would be returned in any random order.
  - Hence can’t access randomly.

HashSet - Example

```
import java.util.*;
class TestCollection1{
    public static void main(String args[]){
        HashSet<String> al=new HashSet<String>();  //creating hashset
        al.add("Ravi");   //adding object in hashset
        al.add("Vijay");
        al.add("Ravi");  // duplicate object
        al.add("Ajay");

       //getting Iterator from hashset to traverse elements .
       // Can also use enhanced for loop to access the item but not normal for loop
        Iterator itr=al.iterator();
         while(itr.hasNext()){
           System.out.println(itr.next());
        }
    }
}
```

- Output

```
Vijay
Ajay
Ravi
```

HashSet– Some Methods

Method	Description
boolean **add**(**E** e)	Adds the specified element to this set if it is not already present.
void **clear**()	Removes all of the elements from this set.
boolean **contains**(**Object** o)	Returns true if this set contains the specified element.
boolean **isEmpty**()	Returns true if this set contains no elements.
Iterator<**E**> **iterator**()	Returns an iterator over the elements in this set.
boolean **remove**(**Object** o)	Removes the specified element from this set if it is present.
int **size**()	Returns the number of elements in this set (its cardinality).

Difference between List and Set

- HashSet doesn’t maintain any order, the elements would be returned in any random order. List does.
- List can contain duplicate elements whereas Set contains unique elements only.

Hashtable

- Hashtable was part of the original java.util and is a concrete implementation of a **Dictionary**.
- However, Java 2 re-engineered Hashtable so that it also implements the **Map** interface.
- Thus, Hashtable is now integrated into the **collections** framework.
- Hashtable implements the Map interface and extends Dictionary class.
- It contains only unique elements.
- It may have not have any null key or value.
- It is synchronized.

Hashtable – some methods

Modifier and Type	Method and Description
boolean	contains(Object value) Tests if some key maps into the specified value in this hashtable.
boolean	containsKey(Object key) Tests if the specified object is a key in this hashtable.
boolean	containsValue(Object value) Returns true if this hashtable maps one or more keys to this value.
Enumeration<V>	elements() Returns an enumeration of the values in this hashtable.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
boolean	equals(Object o) Compares the specified Object with this Map for equality, as per the definition in the Map interface.
void	forEach(BiConsumer<? super K,? super V> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	hashCode() Returns the hash code value for this Map as per the definition in the Map interface.
boolean	isEmpty() Tests if this hashtable maps no keys to values.

Hashtable – some methods

Modifier and Type	Method and Description
Enumeration<K>	keys() Returns an enumeration of the keys in this hashtable.
Set<K>	keySet() Returns a Set view of the keys contained in this map.
V	put(K key, V value) Maps the specified key to the specified value in this hashtable.
void	putAll(Map<? extends K,? extends V> t) Copies all of the mappings from the specified map to this hashtable.
V	remove(Object key) Removes the key (and its corresponding value) from this hashtable.
boolean	remove(Object key, Object value) Removes the entry for the specified key only if it is currently mapped to the specified value.
V	replace(K key, V value) Replaces the entry for the specified key only if it is currently mapped to some value.
boolean	replace(K key, V oldValue, V newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
int	size() Returns the number of keys in this hashtable.
Collection<V>	values() Returns a Collection view of the values contained in this map.

Hashtable - Example

```
import java.util.Enumeration;
import java.util.Hashtable;

public class TestHashTable {
    public static void main(String[] args) {
        Hashtable<String, String> ht = new Hashtable<>();
        ht.put("011001", "Arifa");
        ht.put("011002", "Reza");
        ht.put("011003", "Basir");
        ht.put("011004", "Salma");

        // Retrieve the keys
        Enumeration<String> keys=ht.keys();
        while(keys.hasMoreElements()){
            String key = keys.nextElement();
            String val = ht.get(key); // Get the value
            System.out.printf("%s : %s", key, val);
        }

        // Retrieve the elements/values
        Enumeration<String> elements=ht.elements();
        while(elements.hasMoreElements()){
            System.out.println(elements.nextElement());
        }
    }
}
```

Output:

```
<terminated> TestHashTable
011004 : Salma
011003 : Basir
011002 : Reza
011001 : Arifa
Salma
Basir
Reza
Arifa
```

Collections Class

Collections Class

- This class consists exclusively of static methods that operate on or return collections.
- It contains polymorphic algorithms that operate on collections.

Some methods of Collections class

Modifier and Type	Method and Description
`static <T> Boolean addAll(Collection<? super T> c, T... elements)`	Adds all of the specified elements to the specified collection.
`static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`	Searches the specified list for the specified object using the binary search algorithm.
`static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`	Searches the specified list for the specified object using the binary search algorithm.
`static <T> void copy(List<? super T> dest, List<? extends T> src)`	Copies all of the elements from one list into another.
`static boolean disjoint(Collection<?> c1, Collection<?> c2)`	Returns `true` if the two specified collections have no elements in common.
`static <T> void fill(List<? super T> list, T obj)`	Replaces all of the elements of the specified list with the specified element.
`static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)`	Returns the maximum element of the given collection, according to the *natural ordering* of its elements.
`static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)`	Returns the maximum element of the given collection, according to the order induced by the specified comparator.

Some methods of Collections class

Modifier and Type	Method and Description
`static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)`	Returns the minimum element of the given collection, according to the *natural ordering* of its elements.
`static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)`	Returns the minimum element of the given collection, according to the order induced by the specified comparator.
`static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)`	Replaces all occurrences of one specified value in a list with another.
`static void reverse(List<?> list)`	Reverses the order of the elements in the specified list.
`static <T extends Comparable<? super T>> void sort(List<T> list)`	Sorts the specified list into ascending order, according to the **natural ordering** of its elements.
`static <T> void sort(List<T> list, Comparator<? super T> c)`	Sorts the specified list according to the order induced by the specified comparator.
`static void swap(List<?> list, int i, int j)`	Swaps the elements at the specified positions in the specified list.

- Notice all the methods that involve comparison of 2 objects, either require the Comparable interface or Comparator interface.

Comparable interface(java.lang)

- Class whose objects to be sorted, compared implement this interface(Comparable or Comparable<T>).
  - have to implement compareTo(Object) or compareTo(<T>)method.
  - The comparison logic has to be implemented inside the compreTo() method.
  - collection of that object can be sorted automatically using Collection.sort() or Arrays.sort().
  - Object will be sort on the basis of compareTo method in that class.

Comparator interface (java.util)

- Class whose objects to be sorted do not need to implement this interface.
- Some third class can implement this interface to sort
- Need to implement the sorting logic inside compare(<T> o1, <T< o2) method.

Example - Comparable interface

```
import java.util.*;

public class Employee {
	private int id;
	private String name;
	private Integer salary;

	public Employee(int id, String name, Integer sal){
		this.id = id;
		this.name = name;
		this.salary = sal;
	}

	public Integer getSalary(){
		return salary;
	}

	public String toString(){
		return id+"\t"+name+"\t"+salary;
	}
}
```

Example - Comparable interface

```
import java.util.*;

public class TestEmployee {
	public static void main(String a[]){
		List<Employee> emps = new ArrayList<Employee>();
		emps.add(new Employee(10, "Shakil", 25000));
		emps.add(new Employee(120, "Mamun", 45000));
		emps.add(new Employee(210, "Zaman", 14000));
		emps.add(new Employee(150, "Hasan", 24000));
		Collections.sort(emps); // The method sort(List<T>) in the type Collections is not applicable
 for the arguments (List<Empl>)
		System.out.println("Sorted List");
		for(Employee e: emps)
			System.out.println(e.toString());

		Employee maxSal = Collections.max(emps); // The method max(Collection<? extends T>)
 in the type Collections is not applicable for the arguments (List<Empl>)
		System.out.println("\nEmployee with max salary: "+maxSal);
	}
}
```

Example - Comparable interface

```
import java.util.*;
public class Employee implements Comparable<Employee>{
	private int id;
	private String name;
	private Integer salary;

	public Employee(int id, String name, Integer sal){
		this.id = id;
		this.name = name;
		this.salary = sal;
	}

	public Integer getSalary(){
		return salary;
	}

	public int compareTo(Employee emp) {
		return this.salary.compareTo(emp.salary);
	}

	public String toString(){
		return id+"\t"+name+"\t"+salary;
	}
}
```

Example - Comparable interface

```
import java.util.*;

public class TestEmployee {
	public static void main(String a[]){
		List<Employee> emps = new ArrayList<Employee>();
		emps.add(new Employee(10, "Shakil", 25000));
		emps.add(new Employee(120, "Mamun", 45000));
		emps.add(new Employee(210, "Zaman", 14000));
		emps.add(new Employee(150, "Hasan", 24000));
		Collections.sort(emps); // No error
		System.out.println("Sorted List");
		for(Employee e: emps)
			System.out.println(e.toString());

		Employee maxSal = Collections.max(emps); // No error
		System.out.println("\nEmployee with max salary: "+maxSal);
	}
}
```

Example - Comparable interface

- Output

```
Sorted List
210        Zaman   14000
150        Hasan   24000
10         Shakil  25000
120        Mamun   45000

Employee with max salary: 120          Mamun   45000
```

Example - Comparator interface

```
import java.util.*;
public class Employee{
	private int id;
	private String name;
	private Integer salary;
	public Employee(int id, String name, Integer sal){
		this.id = id;
		this.name = name;
		this.salary = sal; }

	public Integer getSalary(){
		return salary;	}

	public String toString(){
		return id+"\t"+name+"\t"+salary; }
}

 class CompareEmployee implements Comparator<Employee> {
	@Override
	public int compare(Employee o1, Employee o2) {
		return o1.getSalary().compareTo(o2.getSalary());
	}
 }
```

Example - Comparator interface

```
import java.util.*;

public class TestEmployee {
        public static void main(String a[]){
                List<Employee> emps = new ArrayList<Employee>();
                emps.add(new Employee(10, "Shakil", 25000));
                emps.add(new Employee(120, "Mamun", 45000));
                emps.add(new Employee(210, "Zaman", 14000));
                emps.add(new Employee(150, "Hasan", 24000));
                Collections.sort(emps, new CompareEmployee ()); // No error
                System.out.println("Sorted List");
                for(Employee e: emps)
                                System.out.println(e.toString());

                Employee maxSal = Collections.max(emps, new CompareEmployee()); // No error
                System.out.println("\nEmployee with max salary: "+maxSal);
      }
}
```

Note: If want to apply different comparison logic for sort and max can use Anonymous class.

Example - Comparator interface

```
public class TestEmployee {
    public static void main(String a[]){
        List<Employee> emps = new ArrayList<Employee>();
        emps.add(new Employee(10, "Shakil", 25000));
        emps.add(new Employee(120, "Mamun", 45000));
        emps.add(new Employee(210, "Zaman", 14000));
        emps.add(new Employee(150, "Hasan", 24000));
        // Sort by name
        Collections.sort(emps, new Comparator<Employee>(){
            public int compare(Employee o1, Employee o2) {
              return o1.getName().compareTo(o2.getName());
            }
        });
        System.out.println("Sorted List");
        for(Employee e: emps)
          System.out.println(e.toString());

        // compare using salary
        Employee maxSal = Collections.max(emps, new Comparator<Employee>(){
            public int compare(Employee o1, Employee o2) {
              return o1.getSalary().compareTo(o2.getSalary());
            }
        });
        System.out.println("\nEmployee with max salary: "+maxSal);
    }
}
```

Output:

```
Sorted List
150     Hasan   24000
120     Mamun   45000
10      Shakil  25000
210     Zaman   14000

Employee with max salary: 120   Mamun   45000
```

Collection Features – Java version

Java Version	Collection Features	Example
1.2	Introduced Collection of Object	ArrayList al=**new** ArrayList();
1.5	Introduced Generics	ArrayList<String> al=**new** ArrayList<String>();
1.7	No need to specify the Type during Object creation	ArrayList<String> al=**new** ArrayList<>();

Java Generics

Let’s start with an example

- We want to build a program that will work with List. The feature we need are
  - Display the items of the List
  - Find the sum and average of those numbers

Display the Items

Display the items

- Code to show the numbers

```
import java.util.ArrayList;

public class TestGenerics {

    public static void main(String[] args) {
        Integer[] iList = {1,2,3,4,5,6};
        Double[] dList = {1.0,2.0,3.0,4.0,5.0,6.0};
        Float[] fList = {1.0f,2.0f,3.0f,4.0f,5.0f,6.0f};

        showNumbers(iList);
        showNumbers(dList);
        showNumbers(fList);;

    }

}
```

Display the items

```
public static void showNumbers(Integer[] list){
    System.out.println("Integer list contains the following numbers.");
    for (int i: list)
        System.out.print(i + " ");

    System.out.println();
}

public static void showNumbers(Double [] list){
    System.out.println("Double list contains the following numbers.");
    for (double i: list)
        System.out.print(i + " ");

    System.out.println();
}

public static void showNumbers(Float [] list){
    System.out.println("Float list contains the following numbers.");
    for (float i: list)
        System.out.print(i + " ");

    System.out.println();
}
```

Java Generics

- **Java Genrics** is one of the most important feature introduced in Java 5.
- *Generics means **parameterized types**.*
- *Parameterized types* enable you to create **classes**, **interfaces**, and **methods** in which the type of data upon which they operate is specified as a parameter.
- Using generics, it is possible to **create a single class**, for example, that automatically **works with different types** of data.

Display the items – With Generics

```
import java.util.ArrayList;

public class TestWithGenerics {
    public static <T > void showNumbers(T[] list){
        System.out.println("List contains the following numbers.");
        for (T i: list)
            System.out.print(i + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        Integer[] iList = {1,2,3,4,5,6};
        Double[] dList = {1.0,2.0,3.0,4.0,5.0,6.0};
        Float[] fList = {1.0f,2.0f,3.0f,4.0f,5.0f,6.0f};

        showNumbers(iList);
        showNumbers(dList);
        showNumbers(fList);
    }
}
```

Display the items – With Generics

```
import java.util.ArrayList;

public class TestWithGenerics {
    public static <T > void showNumbers(T[] list){
        System.out.println("List contains the following numbers.");
        for (T i: list)
            System.out.print(i + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        Integer[] iList = {1,2,3,4,5,6};
        Double[] dList = {1.0,2.0,3.0,4.0,5.0,6.0};
        Float[] fList = {1.0f,2.0f,3.0f,4.0f,5.0f,6.0f};
        Character[] cList = {'a', 'b', 'c', 'd'};

        showNumbers(iList);
        showNumbers(dList);
        showNumbers(fList);
        showNumbers(cList);
    }
}
```

Generic Class and Interface

Generalized Class–without generics

- Java **always had** the ability to create generalized classes, interfaces, and methods by operating through references of type **Object.**
  - **Because Object is the superclass of all other classes, an Object reference can refer** to any type object.
  - Thus, in pre-generics code, generalized classes, interfaces, and methods used **Object references to operate on various types of objects.**
- **The problem was that they**
  - could not do so with type safety.
  - Do explicit casting

Generic Class – with Object

```
public class GenericsTypeOld {
        private Object t;
        public Object get() {
             return t;
         }

        public void set(Object t) {
             this.t = t;
         }

        public static void main(String args[]){
            GenericsTypeOld type = new GenericsTypeOld();
            type.set("Pankaj");
            String str = (String) type.get(); //type casting, error prone & can cause
 ClassCastException
            type.set(new Student(“abc”, 123));
            str = (String) type.get(); // will throw ClassCastException
         }
}
```

Generic Class

```
public class GenericsType<T> {
    private T t;
    public T get(){
        return this.t;
    }

    public void set(T t1){
        this.t=t1;
    }

    public static void main(String args[]){
        GenericsType<String> type = new GenericsType<>();
        type.set("Pankaj"); //valid
        type.set(new Student(“abc”, 123)); //invalid, will get compiler error

        GenericsType type1 = new GenericsType(); //raw type
        type1.set("Pankaj"); //valid
        type1.set(new Student(“abc”, 123)); //valid
        type1.set(10); //valid and autoboxing support
    }
}
```

Generic Class - Stack

```
import java.util.*;
public class GenericStack <T> {
    private ArrayList<T> stack = new ArrayList<T> ();
    private int top = 0;
    public int size () {
                return top;
     }
    public void push (T item) {
                stack.add (top++, item);
     }
    public T pop () {
                return stack.remove (--top);
    }
    public static void main (String[] args) {
        GenericStack<Integer> s = new GenericStack<Integer> ();  // this stack will hold only Integer. But
         you can use this class to create a stack to hold any type of object that you want.
         s.push (17);
         int i = s.pop ();
         System.out.format ("%4d%n", i);
     }
}
```

Generic Interface

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters.
  - This stands in sharp contrast to the variable naming conventions that you already know about
  - Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
- The most commonly used type parameter names are:
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

Generic Method – Another example

- Sometimes we don't want whole class to be parameterized, in that case we can create java generics method.
- Since constructor is a special kind of method, we can use generics type in constructors too.

```
public class GenericsMethods {
    //Java Generic Method
    public static <T> boolean isEqual(GenericsType<T> g1, GenericsType<T> g2){
        return g1.get().equals(g2.get());
    }

    public static void main(String args[]){
        GenericsType<String> g1 = new GenericsType<>();
        g1.set("Pankaj");

        GenericsType<String> g2 = new GenericsType<>();
        g2.set("Pankaj");

        boolean isEqual = GenericsMethods.<String>isEqual(g1, g2);
        //above statement can be written simply as
        isEqual = GenericsMethods.isEqual(g1, g2);
    }
}
```

Generic Method – Find Max

- Create a static generic method which will take a generic array as parameter and find the maximum element for the array and return.

```
public class TestGeneric {

    public static <T extends Comparable<T>> T maximum(T[] a){
        T max = a[0];
        for(int i =1; i< a.length; i++)
            if(max.compareTo(a[i]) < 0)
                max = a[i];
        return max;
    }

    public static void main(String[] args) {
        System.out.println("Max: " + TestGeneric.maximum(new Integer[]{1,2,5,3,9,12,8}));
        System.out.println("Max: " + TestGeneric.maximum(new Double[]{1.9,2.0,5.9,3.2,9.5}));
        System.out.println("Max: " + TestGeneric.maximum(new String[]{"abc", "xyz", "Xyz", "yY"}));
    }
}
```

```
<terminated> TestGeneric
Max: 12
Max: 9.5
Max: yY
```

Generic Method – Find Max

```
public class TestGeneric {
	public static <T extends Comparable<T>> T maximum(T[] a){
		T max = a[0];
		for(int i =1; i< a.length; i++)
			if(max.compareTo(a[i]) < 0)
					max = a[i];
		return max;
	}

	public static void main(String[] args) {
		System.out.println("Max: " + TestGeneric.maximum(new Integer[]{1,2,5,3,9,12,8}));
		BankAccount b1 = new BankAccount("abc", "123", 2000.0);
		BankAccount b2 = new BankAccount("ab", "23", 5000.0);
		System.out.println("Max: " + TestGeneric.maximum(new BankAccount[]{b1,b2})); //The method maximum(T[])
		in the type TestGeneric is not applicable for the arguments (BankAccount[])
	}
}

class BankAccount {
	String name, id;
	Double balance;
	public BankAccount(String name, String id, Double balance) {
		this.name = name;
		this.id = id;
		this.balance = balance;
	}
}
```

Generic Method – Find Max

```
public class TestGeneric {
	public static <T extends Comparable<T>> T maximum(T[] a){
		T max = a[0];
		for(int i =1; i< a.length; i++)
			if(max.compareTo(a[i]) < 0)
					max = a[i];
		return max;
	}

	public static void main(String[] args) {
		System.out.println("Max: " + TestGeneric.maximum(new Integer[]{1,2,5,3,9,12,8}));
		BankAccount b1 = new BankAccount("abc", "123", 2000.0);
		BankAccount b2 = new BankAccount("ab", "23", 5000.0);
		System.out.println("Max: " + TestGeneric.maximum(new BankAccount[]{b1,b2}));
	}
}

class BankAccount implements Comparable<BankAccount>{
	String name, id;
	Double balance;
	public BankAccount(String name, String id, Double balance) {
		this.name = name;
		this.id = id;
		this.balance = balance;
	}
	public int compareTo(BankAccount o) {
		return balance.compareTo(o.balance); //can also use Double.compare(balance, o.balance) for primitive
	}
}
```

Class & Method- Different Generics

- In the following example, the inspect method may take different type of parameter than the class itself.

```
public class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text");     }
}
```

Erasure

Erasure

- When your Java code is compiled, all generic type information is removed (erased)

- And replace the type parameters with their real type,
  - If no type is specified it will be replaced with **Object.**

- **then applying the** appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments.

Erasure – Compiled code with type

Original Class	Compiled class when specify String type e.g. GenericType<**String**>

public class GenericsType<T> {
private T t;
public T get(){
return this.t;
}
 	
public void set(T t1){
this.t=t1;
}
}
	public class GenericsType {
	private String t;
	public String get(){
	return this.t;
	}
	
	public void set(String t1){
	this.t=t1;
	}
	}

Erasure –Compiled code without type

Original Class	Compiled class when specify String type e.g. GenericType a = new GenericType();

public class GenericsType<T> {
private T t;
public T get(){
return this.t;
}
 	
public void set(T t1){
this.t=t1;
}
}
	public class GenericsType {
	private Object t;
	public Object get(){
	return this.t;
	}
	
	public void set(Object t1){
	this.t=t1;
	}
	}

Calculate the Average

Calculate the Average

```
public class TestWithBound {
    public static <T> double getAverage(T[] list){
        double sum = 0;

        for(T item: list){
            sum = sum + item; // The operator + is undefined for the argument
             type(s) double, T
            sum += item.doubleValue(); // The method doubleValue() is undefined
             for the type T
        }

        return sum/list.length;
    }

}
```

Bounded Type Parameters

- There may be times when you want to **restrict the types that can be used as type arguments** in a **parameterized** type.
- For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.
  - UpperBound

Upper Bound

- When we want to restrict the type parameter to allow a specified class and its descendant, we use upper bound.
- In another word; when specifying a type parameter, you can create an upper bound from which all type arguments must be derived.
- How to declare
  - "U extends T"
    - means **any class which extends T**.
    - Thus, we are referring to the *children of T*.
    - Hence, **T is the upper bound.**
    - **The upper-most class in the inheritance hierarchy**
  - Example – T extends Number
    - means any class extend the Number class e.g. Integer, Double, Float etc.

Upper Bound

- Note that, in this context, extends is used in a general sense to mean either
  - "**extends" (as in classes) or**
  - **"implements" (as in interfaces).**

- A bound can include both a class type and one or more interfaces.
  - In this case, the class type must be specified first.

- When a bound includes an interface type,
  - **only type arguments that implement that interface are legal**.

Upper Bound

- When specifying a bound that has **a class** and **an** interface, or **multiple** interfaces, use the **&** operator to connect them.

- For example,

  class Gen<T extends MyClass & MyInterface> {}

- Any type argument passed to T must be a subclass of *MyClass* and implement *MyInterface*.

Calculate the Average-UpperBound

```
public class TestWithBound {
    public static <T extends Number > double getAverage(T[] list){
        double sum = 0;

        for(T item: list){
            sum += item.doubleValue(); // valid
        }

        return sum/list.length;
    }

}
```

Calculate the Average-UpperBound

```
public class TestWithBound {
    public static < T extends Number > double getAverage(T[] list){
        double sum = 0;

        for(T item: list){
            sum += item.doubleValue(); // valid
        }
        return sum/list.length;
    }

    public static void main(String[] args) {
        Integer[] iList = {1,2,3,4,5,6};
        Double[] dList = {1.0,2.0,3.0,4.0,5.0,6.0};
        Float[] fList = {1.0f,2.0f,3.0f,4.0f,5.0f,6.0f};
        Character[] cList = {'a', 'b', 'c', 'd'};

        System.out.println("Average:" + getAverage(iList));
        System.out.println("Average:" + getAverage(dList));
        System.out.println("Average:" + getAverage(fList));
        System.out.println("Average:" + getAverage(cList)); // The method getAverage(T[]) in the
          type TestWithBound is not applicable for the arguments (Character[])
        }
}
```

Another Example

- In the following example, the inspect method will take object of Number Class and its descendant.

```
public class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
}
```

Another Example

- So, if you call inspect with other types, you will get error as shown below

```
public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    integerBox.set(new Integer(10));
    integerBox.inspect("some text"); // error: this is still String!
}
```

- Compiler will give you the following error

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                   ^
1 error
```