

Basic data structures

1. Array
2. Graph
3. Hash Refer MS team recording
4. Linked List
5. Pointers
6. Recursion
7. Stack
8. Queues
9. Structure
10. Tree

1. Array

2. Graph

Refer MS team recording

3. Hash/Hashing/Hash function

Refer MS team recording

4. Linked List

Refer MS team recording

5. Pointers

Refer MS team recording

6. Recursion

Refer MS team recording

7. Stack

Refer MS team recording

8. Queue

Refer MS team recording



Refer MS team recording

9. Structure

Structs are value types while classes are reference types.

Structs can be instantiated without using a new operator. A struct cannot inherit from another struct or class, and it cannot be the base of a class.

A structure doesn't support inheritance, and polymorphism, but a class supports both.

A Structure is not secure and cannot hide its implementation details from the end-user while a class is secure and can hide its programming and designing details

Class	Structure
Members of a class are private by default.	Members of a structure are public by default.
Memory allocation happens on the heap.	Memory allocation happens on a stack.
It is a reference type data type.	It is a value type data type.
It is declared using the class keyword.	It is declared using the struct keyword.

10. Tree

Refer MS team recording

The Java Collections Framework is like the C++ Standard Template Library: "a unified architecture for representing and manipulating collections (objects that group multiple elements into a single unit)."

All of the core collections featured in the Java Collections Framework are already present in core Python.

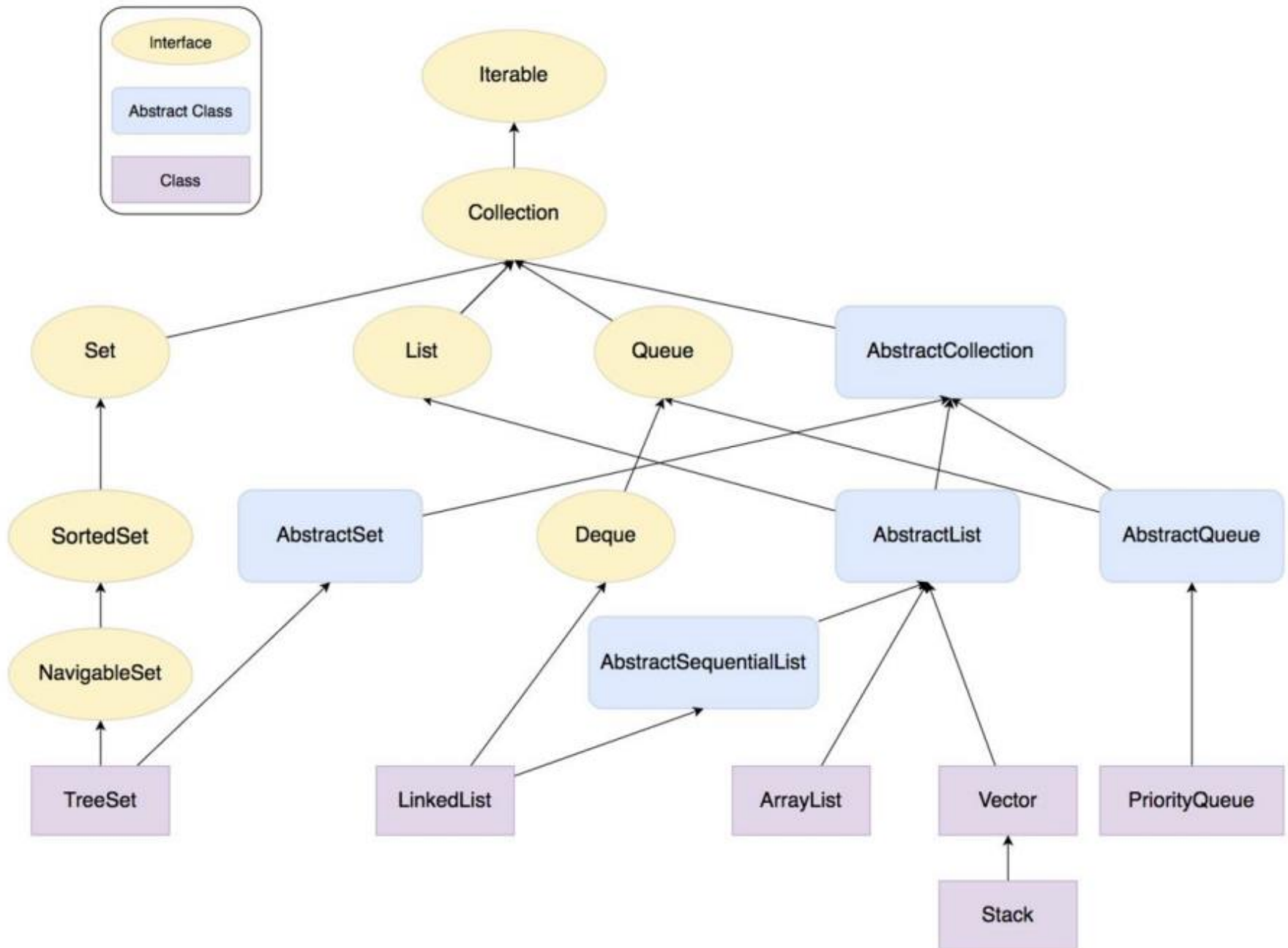
As it turns out, the equivalent to the Java Collections Framework in Python is... Python. All of the core collections featured in the Java Collections Framework are already present in core Python.

Give it a try! Sequences provide lists, queues, stacks, etc. Dictionaries are your hash-tables and maps. Sets are present, etc.

One might consider Python a "higher" language than Java, because it natively provides all of these higher order abstract data types intrinsically. (It also supports Object Oriented, procedural, and functional programming methodologies.)

NOTE If you are familiar with C++, then you will find it helpful to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls a collection. However, there are significant differences between the Collections Framework and the STL. It is important to not jump to conclusions.

The Collections Framework hierarchy is as follows:



CSE2005 (Object Oriented Programming Systems)

Module 4
**The Collections Framework and Generic
Programming**

Dr. Arundhati Das



Java's most powerful subsystems: the *Collections Framework*

- “The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.” -Java complete reference book
- Why Collection Framework? What is the advantage?
 - Java provides many data structures implemented in pre-defined interfaces and classes, so that you don't have to write them from scratch. This will help in solving and focusing on a bigger problem.
 - Already implemented in an optimized way
 - Less computing time

NOTE If you are familiar with C++, then you will find it helpful to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls a collection. However, there are significant differences between the Collections Framework and the STL. It is important to not jump to conclusions.

All of the core collections featured in the Java Collections Framework are already present in core Python.

Collections framework contd.

- The data structures are:
 - It defines a collections framework as “a unified architecture for representing and manipulating collections, allowing them to be manipulated independent of the details of their representation.”
 - It defines a “collection” as “an object that represents a group of objects”.
- **The java.util package contains all the classes and interfaces for the Collection framework, Java’s most powerful subsystems.**

Collections framework contd.

- In JDK 1.2 onwards, Java developers introduced the Collections Framework, which is an important framework to help you achieve all of your data operations.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection framework provides framework provides many interfaces (Set, List, Queue, Deque)
- classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

[Try to understand the collection framework and collection are two things, collection is an interface under the framework of collection. Collection framework contain other interfaces such as map and iterator]

Collections framework contd.

Collection Interfaces - Represent different types of collections, such as sets, lists, queue and maps. These interfaces form the basis of the framework.

General-purpose Implementations - Primary implementations of the collection interfaces.

Legacy Implementations - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.

Wrapper Implementations - Add functionality, such as synchronization, to other implementations.

Convenience Implementations - High-performance "mini-implementations" of the collection interfaces.

Abstract Implementations - Partial implementations of the collection interfaces to facilitate custom implementations.

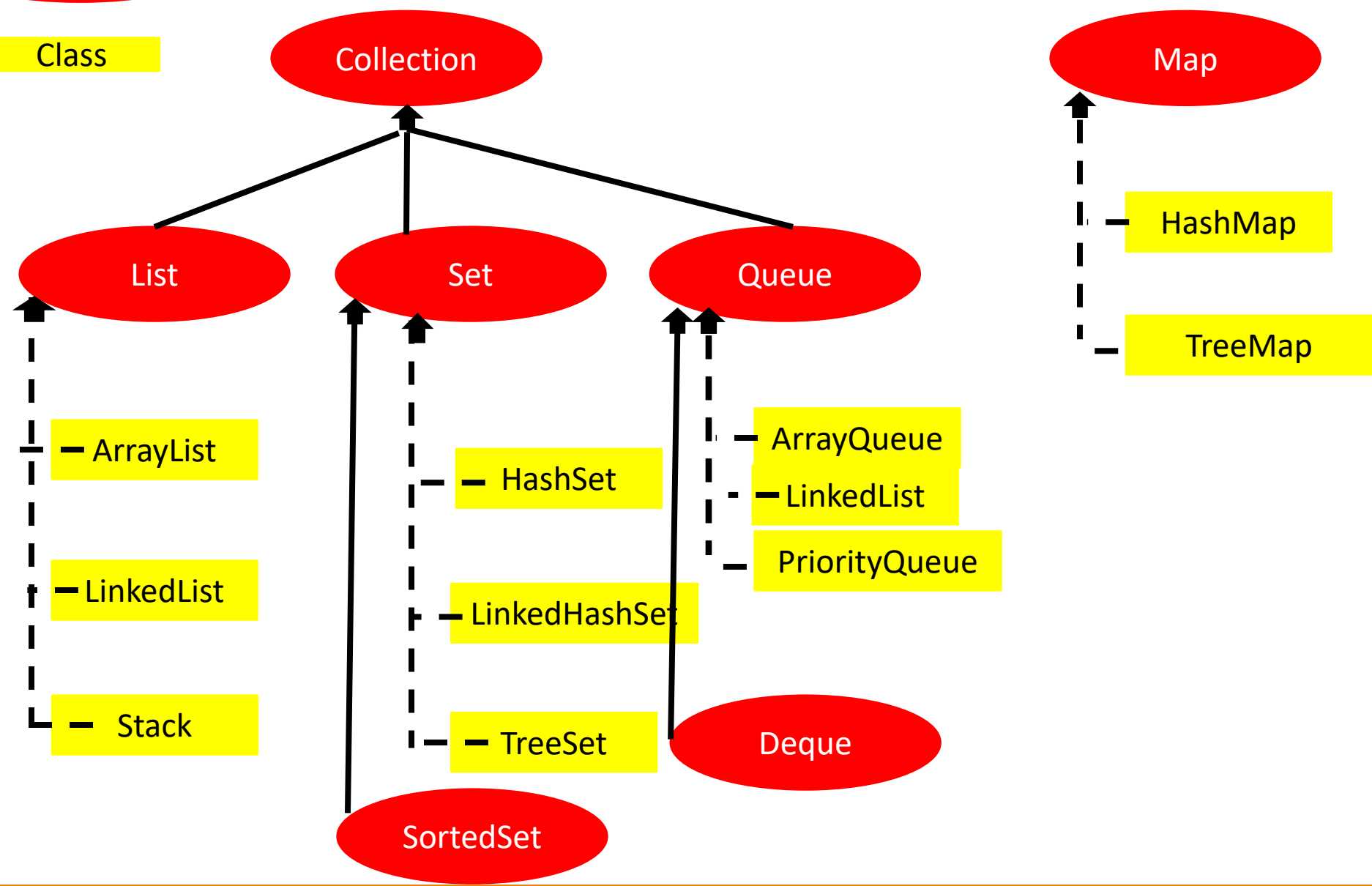
Algorithms - Static methods that perform useful functions on collections, such as sorting a list.

Infrastructure - Interfaces that provide essential support for the collection interfaces.

Array Utilities - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality is being added to the Java platform at the same time and relies on some of the same infrastructure.

Interface

Class

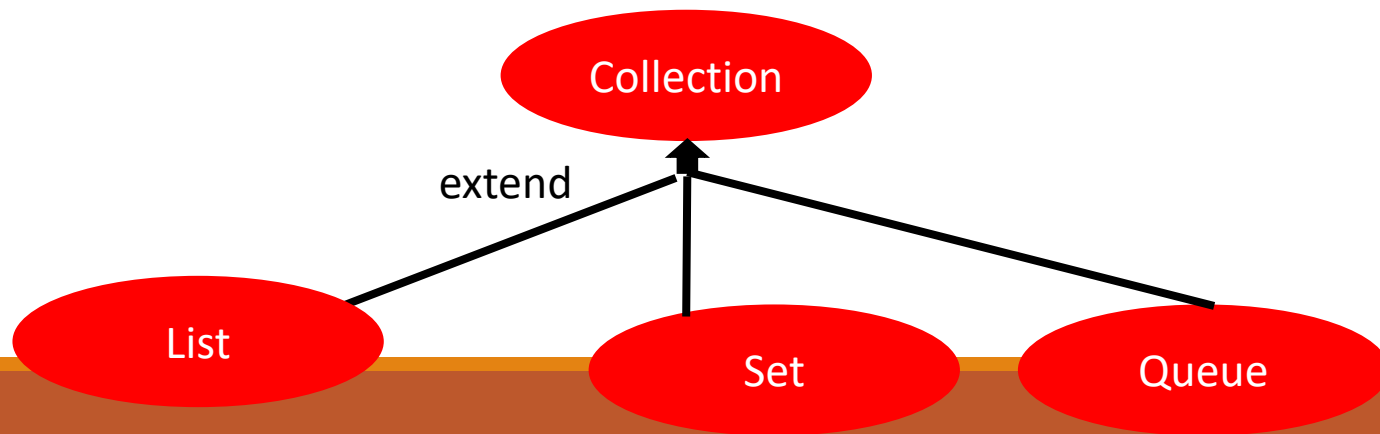


Collection interfaces

The core collection interfaces encapsulate different types of collections. They are interfaces so they do not provide an implementation!

The Collection interface is the interface which is implemented by all the classes in the collection framework.

Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered.



Methods of Collection interface

Method	Description
<code>boolean add(E obj)</code>	Adds <i>obj</i> to the invoking collection. Returns true if <i>obj</i> was added to the collection. Returns false if <i>obj</i> is already a member of the collection and the collection does not allow duplicates.
<code>boolean addAll(Collection<? extends E> c)</code>	Adds all the elements of <i>c</i> to the invoking collection. Returns true if the collection changed (i.e., the elements were added). Otherwise, returns false .
<code>void clear()</code>	Removes all elements from the invoking collection.
<code>boolean contains(Object obj)</code>	Returns true if <i>obj</i> is an element of the invoking collection. Otherwise, returns false .
<code>boolean containsAll(Collection<?> c)</code>	Returns true if the invoking collection contains all elements of <i>c</i> . Otherwise, returns false .
<code>boolean equals(Object obj)</code>	Returns true if the invoking collection and <i>obj</i> are equal. Otherwise, returns false .
<code>int hashCode()</code>	Returns the hash code for the invoking collection.
<code>boolean isEmpty()</code>	Returns true if the invoking collection is empty. Otherwise, returns false .
<code>Iterator<E> iterator()</code>	Returns an iterator for the invoking collection.
<code>default Stream<E> parallelStream()</code>	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations. (Added by JDK 8.)
<code>boolean remove(Object obj)</code>	Removes one instance of <i>obj</i> from the invoking collection. Returns true if the element was removed. Otherwise, returns false .
<code>boolean removeAll(Collection<?> c)</code>	Removes all elements of <i>c</i> from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false .
<code>default boolean removeIf(Predicate<? super E> predicate)</code>	Removes from the invoking collection those elements that satisfy the condition specified by <i>predicate</i> . (Added by JDK 8.)

Table 18-1 The Methods Declared by **Collection**

public interface Collection<E> extends Iterable<E>

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface. It contains only one abstract method `iterator()`.

Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

public interface Collection<E> extends Iterable<E>

```
public interface Collection<E> extends Iterable<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);           //optional  
    boolean remove(Object element);  //optional  
    Iterator<E> iterator();  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    //optional  
    boolean removeAll(Collection<?> c);  
    //optional  
    boolean retainAll(Collection<?> c);  
    //optional  
    void clear();  
    //optional  
  
    // Array operations  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```


A note on iterators

- for loop

- For each loop

Iterator interface provides the facility of iterating the elements in a forward direction only.

- An [Iterator](#) is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator()` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove(); //optional  
}
```

public interface **List<E>** extends Collection<E>

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);           //optional
    boolean add(E element);               //optional
    void add(int index, E element);        //optional
    E remove(int index);                   //optional
    boolean addAll(int index,
        Collection<? extends E> c);        //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

public interface **List**<E> extends Collection<E>

List

- an ordered collection (sometimes called a *sequence*).
- Lists can contain duplicate elements.
- The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).
- Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)

A note on ListIterators

The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasNext and the previous operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.

The nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous

The set method overwrites the last element returned by next or previous with the specified element.

The add method inserts a new element into the list immediately before the current cursor position.

```
public interface ListIterator<E> extends Iterator<E> {  
    boolean hasNext();  
    E next();  
    boolean hasPrevious();  
    E previous();  
    int nextIndex();  
    int previousIndex();  
    void remove(); //optional  
    void set(E e); //optional  
    void add(E e); //optional  
}
```



public interface **Set**<E> extends
Collection<E>

Set

- a collection that cannot contain duplicate elements.
- This interface models the mathematical set abstraction and is used to represent sets, the courses making up a student's schedule, or the processes running on a machine.
- Example implementations of Set interface are HashSet (Hashing based) and TreeSet (balanced BST based).

public interface **Set<E>** extends Collection<E>

```
public interface Set<E> extends Collection<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);           //optional
    boolean remove(Object element);   //optional
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optional
    boolean removeAll(Collection<?> c);       //optional
    boolean retainAll(Collection<?> c);       //optional
    void clear();                             //optional

    // Array Operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

public interface SortedSet<E> extends Set<E>

SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

```
public interface SortedSet<E> extends Set<E> {  
    // Range-view  
    SortedSet<E> subSet(E fromElement, E toElement);  
    SortedSet<E> headSet(E toElement);  
    SortedSet<E> tailSet(E fromElement);  
  
    // Endpoints  
    E first();  
    E last();  
  
    // Comparator access  
    Comparator<? super E> comparator();  
}
```

public interface Queue<E> extends Collection<E>

Queue

- Queue typically order elements in FIFO order except exceptions like PriorityQueue and Deque.
- In PriorityQueue, elements are deleted based on the priority.
- In Deque, Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.
- a collection of Queue is used to hold multiple elements prior to processing.
- Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

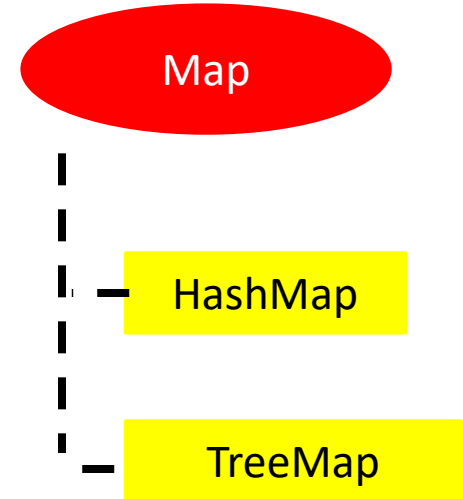
```
public interface Queue<E> extends Collection<E> {  
    E element(); //throws exception if queue is empty  
    E peek(); //null  
    boolean offer(E e); //add - bool  
    E remove(); //throws  
    E poll(); //null  
}
```


public interface Map<K,V>

Map

- an object that maps keys to values.
- A Map cannot contain duplicate keys; each key can map to at most one value.
- If you've used Hashtable, you're already familiar with the basics of Map.
- Example implementation are HashMap and TreeMap. TreeMap implements SortedMap.

The difference between Set and Map interface is that in Set we have only keys, whereas in Map, we have key, value pairs.



public interface Map<K,V>

```
public interface Map<K,V> {  
  
    // Basic operations  
    V put(K key, V value);  
    V get(Object key);  
    V remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
  
    // Bulk operations  
    void putAll(Map<? extends K, ? extends V> m);  
    void clear();  
  
    // Collection Views  
    public Set<K> keySet();  
    public Collection<V> values();  
    public Set<Map.Entry<K,V>> entrySet();  
  
    // Interface for entrySet elements  
    public interface Entry {  
        K getKey();  
        V getValue();  
        V setValue(V value);  
    }  
}
```

public interface SortedMap<K,V> extends Map<K,V>

SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

```
public interface SortedMap<K, V> extends Map<K, V>{  
    SortedMap<K, V> subMap(K fromKey, K toKey);  
    SortedMap<K, V> headMap(K toKey);  
    SortedMap<K, V> tailMap(K fromKey);  
    K firstKey();  
    K lastKey();  
  
    Comparator<? super K> comparator();  
}
```

Note on Comparator interface

Comparator is another interface (in addition to Comparable) provided by the Java API which can be used to order objects.

You can use this interface to define an order that is different from the Comparable (natural) order.

Comparable and Comparator interfaces both can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

Comparable and Comparator

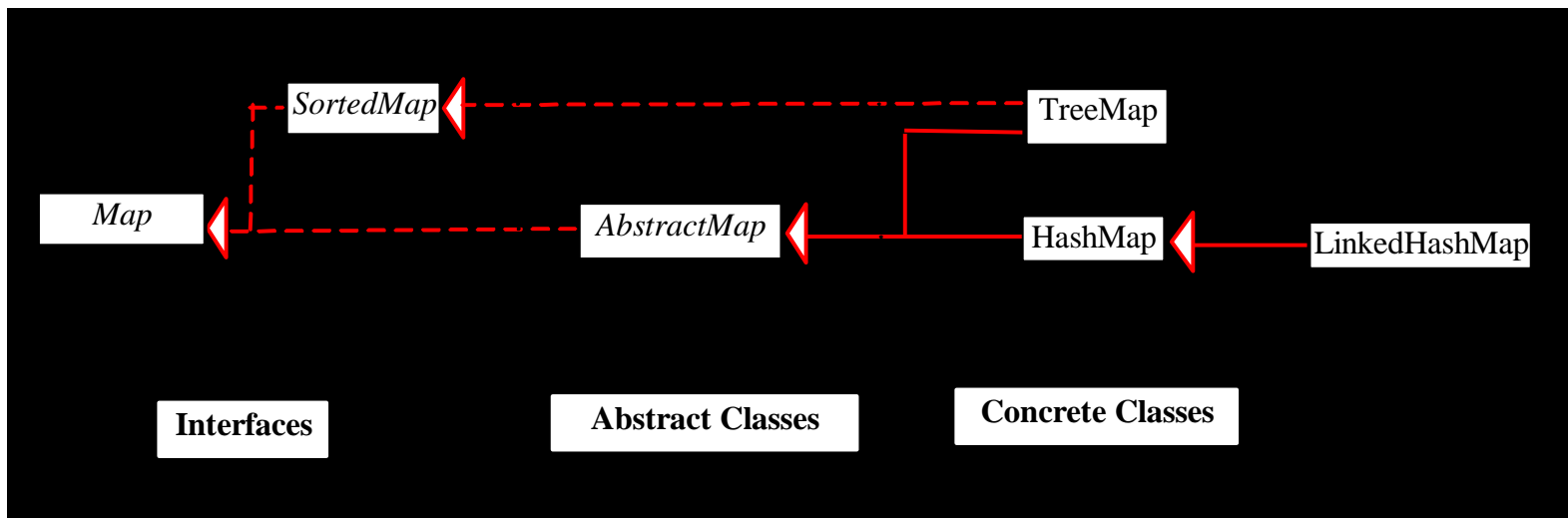
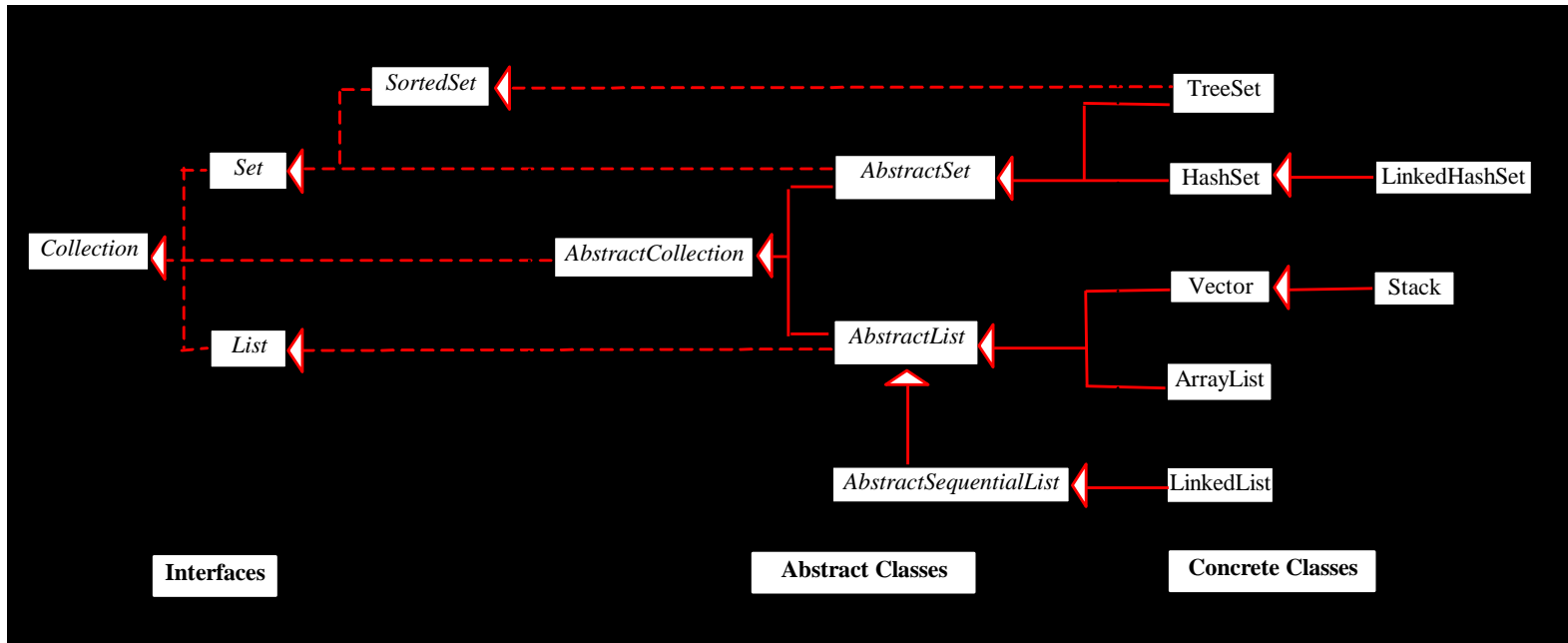
Comparable	Comparator
1) Comparable provides a single sorting sequence . In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides multiple sorting sequences . In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable affects the original class , i.e., the actual class is modified.	Comparator doesn't affect the original class , i.e., the actual class is not modified.
3) Comparable provides compareTo() method to sort elements.	Comparator provides compare() method to sort elements.
4) Comparable is present in java.lang package.	A Comparator is present in the java.util package.
5) We can sort the list elements of Comparable type by Collections.sort(List) method.	We can sort the list elements of Comparator type by Collections.sort(List, Comparator) method.

General-purpose Implementations

Interfaces	Implementations				
	Hash table	Resizable array	Tree (<u>sorted</u>)	Linked list	Hash table + Linked list
Set	HashSet		TreeSet (<u>sorted</u>)		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap (<u>sorted</u>)		LinkedHashMap

Note the naming convention

LinkedList also implements queue and there is a PriorityQueue implementation (implemented with heap)



Making your own implementations

Most of the time you can use the implementations provided for you in the Java API.

In case the existing implementations do not satisfy your needs, you can write your own by extending the abstract classes provided in the collections framework.

implementations

Each of the implementations offers the strengths and weaknesses of the underlying data structure.

What does that mean for:

- Hashtable
- Resizable array
- Tree
- LinkedList
- Hashtable plus LinkedList

Think about these tradeoffs when selecting the implementation!

Lists

- **List** in Java provides the facility to maintain the *ordered collection*.
- It contains the index-based methods to insert, update, delete and search the elements.
- It can have the duplicate elements also. We can also store the null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface.
- It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions.
- The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector.
- The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

```
//Creating a List of type String using ArrayList  
List<String> list=new ArrayList<String>();  
//Creating a List of type Integer using ArrayList  
List<Integer> list=new ArrayList<Integer>();  
//Creating a List of type Book using ArrayList  
List<Book> list=new ArrayList<Book>();  
//Creating a List of type String using LinkedList  
List<String> list=new LinkedList<String>();
```

List Methods

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
boolean equals(Object o)	It is used to compare the specified object with the elements of a list.
int hashCode()	It is used to return the hash code value for a list.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean contains(Object o)	It returns true if the list contains the specified element
boolean containsAll(Collection<?> c)	It returns true if the list contains all the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Splitter<E> splitter()	It is used to create splitter over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.

Collection: LinkedList

LinkedList implements the Collection interface.

It uses a doubly linked list internally to store the elements.

It can store the duplicate elements.

It maintains the insertion order

Ravi
Vijay
Ravi
Ajay

```
import java.util.*;
public class TestJavaCollection2{
    public static void main(String args[]){
        LinkedList<String> al=new LinkedList<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");
        Iterator<String> itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

Difference between ArrayList and LinkedList

- ArrayList and LinkedList both implements List interface and maintains insertion order.

ArrayList	LinkedList
1) ArrayList internally uses a dynamic array to store the elements.	LinkedList internally uses a doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

Wrapper class

A Wrapper class is a **class whose object wraps or contains primitive data types**

When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.

In other words, we can wrap a primitive value into a wrapper class object.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Vector in Java

ArrayList is not synchronized. Its equivalent synchronized class in Java is Vector.

- The **Vector** class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections. It is found in the **java.util package** and implements the [List](#) interface, so we can use all the methods of List interface here.
- Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
- They are very similar to [ArrayList](#) but Vector is synchronized and has some legacy method that the collection framework does not contain.
- It also maintains an insertion order like an ArrayList but it is rarely used in a non-thread environment as it is **synchronized** and due to which it gives a poor performance in adding, searching, delete and update of its elements.
- The Iterators returned by the Vector class are fail-fast. In the case of concurrent modification, it fails and throws the **ConcurrentModificationException**.

```
import java.io.*;
import java.util.*;

class VectorExample {

    public static void main(String[] args)
    {
        // Size of the
        // Vector
        int n = 5;

        // Declaring the Vector with
        // initial size n
        Vector<Integer> v = new Vector<Integer>(n);

        // Appending new elements at
        // the end of the vector
        for (int i = 1; i <= n; i++)
            v.add(i);

        // Printing elements
        System.out.println(v);

        // Remove element at index 3
        v.remove(3);

        // Displaying the vector
        // after deletion
        System.out.println(v);

        // Printing elements one by one
        for (int i = 0; i < v.size(); i++)
            System.out.print(v.get(i) + " ");

    }
}
```


Remove first character from a string in JavaScript

1. Using substring() method

The `substring()` method returns the part of the string between the specified indexes or to the end of the string.

```
1  let str = 'Hello';
2
3  str = str.substring(1);
4  console.log(str);
5
6  /*
7   Output: ello
8  */
```

[List]

Q. Enter 10 names into a list. Write a program to delete the first two characters of the names and arrange the resulting names in:

- a) alphabetical ascending order and print them out.
- b) alphabetical descending order and print them out.

```
List<String> l1=new ArrayList<>();
l1.add("Hritik");
l1.add("Praise");
l1.add("Aniket");
l1.add(2,"Manasa");
System.out.println(l1);
List<String> l2=new ArrayList<>();
for(String s:l1)
{
s=s.substring(2);
System.out.println(s);
l2.add(s);
}
Collections.sort(l2);
System.out.println("ascending"+l2);
Collections.sort(l2, Collections.reverseOrder());
System.out.println("descending"+l2);
```

[Set]

Q. A set of 10 names are given. Write a program to delete the first three characters of the names and arrange the resulting names in alphabetical order and print them out.

a) alphabetical ascending order and print them out.

b) alphabetical descending order and print them out.

```
Set<String> s1=new HashSet<>();
s1.add("Hritik");
s1.add("Praise");
s1.add("Manasa");
s1.add("Aniket");
System.out.println(s1);
Set<String> s2=new TreeSet<>();
for(String s:s1)
{
    s=s.substring(3);
    System.out.println(s);
    s2.add(s);
}
System.out.println("ascending "+s2);

Set<String> s2=new
TreeSet<>(Comparator.reverseOrder());
for(String s:s1)
{
    s=s.substring(3);
    System.out.println(s);
    s2.add(s);
}
System.out.println("descending "+s2);
```

2. Using `slice()` method

The `slice()` method extracts the text from a string and returns a new string.

```
1  let str = 'Hello';
2
3  str = str.slice(1);
4  console.log(str);
5
6  /*
7     Output: ello
8  */
```

3. Using substr() method

The `substr()` method returns a portion of the string, starting at the specified index and extending for a given number of characters or until the string's end.

```
1  let str = 'Hello';
2
3  str = str.substr(1);
4  console.log(str);
5
6  /*
7     Output: ello
8  */
```

[Stack]

Q. Take 5 user input for name and age and push it into a stack. Now write a method to check if the age of the person on the top of the stack is less than equal to 18. If yes, pop it and add it into one list named “Children”. Else add it into another list “Adult”. Display both the lists. Create an user-defined class Citizen where you can define the parameterized constructor with name and age. (LIFO)

```
Stack<Citizen> ci=new Stack<Citizen>();
```

```
ci.push(new Citizen("Amar",20));
```

```
ci.push(new Citizen("Akbar",19));
```

```
ci.push(new Citizen("Antony",14));
```

```
ci.push(new Citizen("Joy", 13));
```

```
ci.push(new Citizen("Happy",21));
```

```
//System.out.println("stack pop ci "+ci.pop());
```

```
System.out.println("stack peek ci "+ci.peek().age);
```

```
System.out.println("stack ci "+ci.size());
```

```
ArrayList<Citizen> children=new ArrayList();
```

```
ArrayList<Citizen> adult=new ArrayList();
```

```
if(ci.peek().age>=18)
```

```
{
```

```
System.out.println("hello from adult");
```

```
adult.add(new Citizen(ci.peek().name,ci.peek().age));
```

```
ci.pop();
```

```
}
```

```
else
```

```
{
```

```
System.out.println("hello from clideren");
```

```
children.add(new Citizen(ci.peek().name,ci.peek().age));
```

```
ci.pop();
```

```
}
```

```
class Citizen
{
String name;
int age;
Citizen(String name, int age)
{
this.name=name;
this.age=age;

System.out.println("name "+name+" age "+age);
}

}
```


[Queue] offer(), poll()

Q. Take 5 user input for name and age and push it into a queue. Now write a method to check if the age of the person on the queue's other end is less than equal to 18. If yes, pop it and add it into one list named "Child". Else add it into another list "Adult". Display both the lists. Create an user-defined class Citizen where you can define the parameterized constructor. (FIFO)

[Exception Handling] WORDLE game

Q. Generate a random 3 digit number using random() method. Ask user to guess 1 digit present in the 3 digit random number. If the digit is present, show its position. If it is not present, throw an user-defined exception showing message “invalid input”. Repeat this step for all 3 digits. If all 3 digits are guessed correctly, display a message “You Win!!!”. If all are not guessed until 5 wrong attempts, display message “You lose”.

//how to generate random number and convert it into array

```
int rand=(int)Math.floor(Math.random()*(1000-100+1)+100);  
System.out.println(rand);  
int number = rand;  
String tempString = Integer.toString(number);  
int numbers[] = new int[tempString.length()];  
for (int i = 0; i < tempString.length(); i++) {  
    numbers[i] = tempString.charAt(i) - '0';  
}
```

```
System.out.println("digits are      "+numbers[0]+"  
"+numbers[1]+"  "+numbers[2]);
```