

Collection Framework

- **Wrapper classes**
- **Collection Framework**
 - List interface and its implementations
 - Iterable interface
- **Collections utility class**
- **Set Interface**
 - Implementation classes
 - Overriding equals and hashCode methods
- **Map interface**
 - **Implementations**
- **Concurrent Collections**
 - **ConcurrentHashMap**
- **Fail-fast and Fail-safe Iterators**

Wrapper Classes

- + Wrapper classes are used to represent primitive values when an Object is required.
- + Wrapper classes are used extensively to wrap primitive classes for use in Collections since a **Collection** can only store objects. It is also used with a number of reflection API classes.
- + The 8 primitive wrapper classes are: Byte (byte), Short (short), Character (char), Integer (int), Long (long), Float (float), Double (double), and Boolean (boolean).

Auto-boxing & Auto-unboxing

- + Java 5 SE introduced auto-boxing. This is the automatic conversion of a primitive to an Object wrapper if required.
- + Auto-unboxing is the automatic conversion of a primitive wrapper class to a primitive.
- + An **Exception** can occur when auto-unboxing if the primitive wrapper is **null**.

Auto-boxing & Auto-unboxing

Introduced in java 5 release

Auto boxing:

Implicit Conversion of primitive type to object type:

```
int a = 100;
```

```
Integer iObj = a; same as Integer iObj = new Integer(a);
```

Auto unboxing :

Implicit Conversion of object type to primitive type :

```
int x = iobj; same as int x = iobj.intValue();
```

```
double d = 3.14;  
Double dObj = new Double(d);  
double r = Double.parseDouble(dObj);  
same as  
double r = dObj.doubleValue();
```

```
double d = 3.14;  
Double dObj = d;  
double r = dObj;
```

Auto-Boxing

Auto-Unboxing

Converting String objects to primitive types

Converting String objects to primitive types:

Use static method, **parsexxx()** of the wrapper classes, where xxx indicates wrapper class name.

Ex. To convert String object to double type,

```
String string="3.14";
```

```
double x = Double.parseDouble(string);
```

Collection Framework

A collection is a container that represents a group of objects .

Java 1.2 provided **Collection Framework** , a unified architecture for representing and manipulating collections in a standard way.

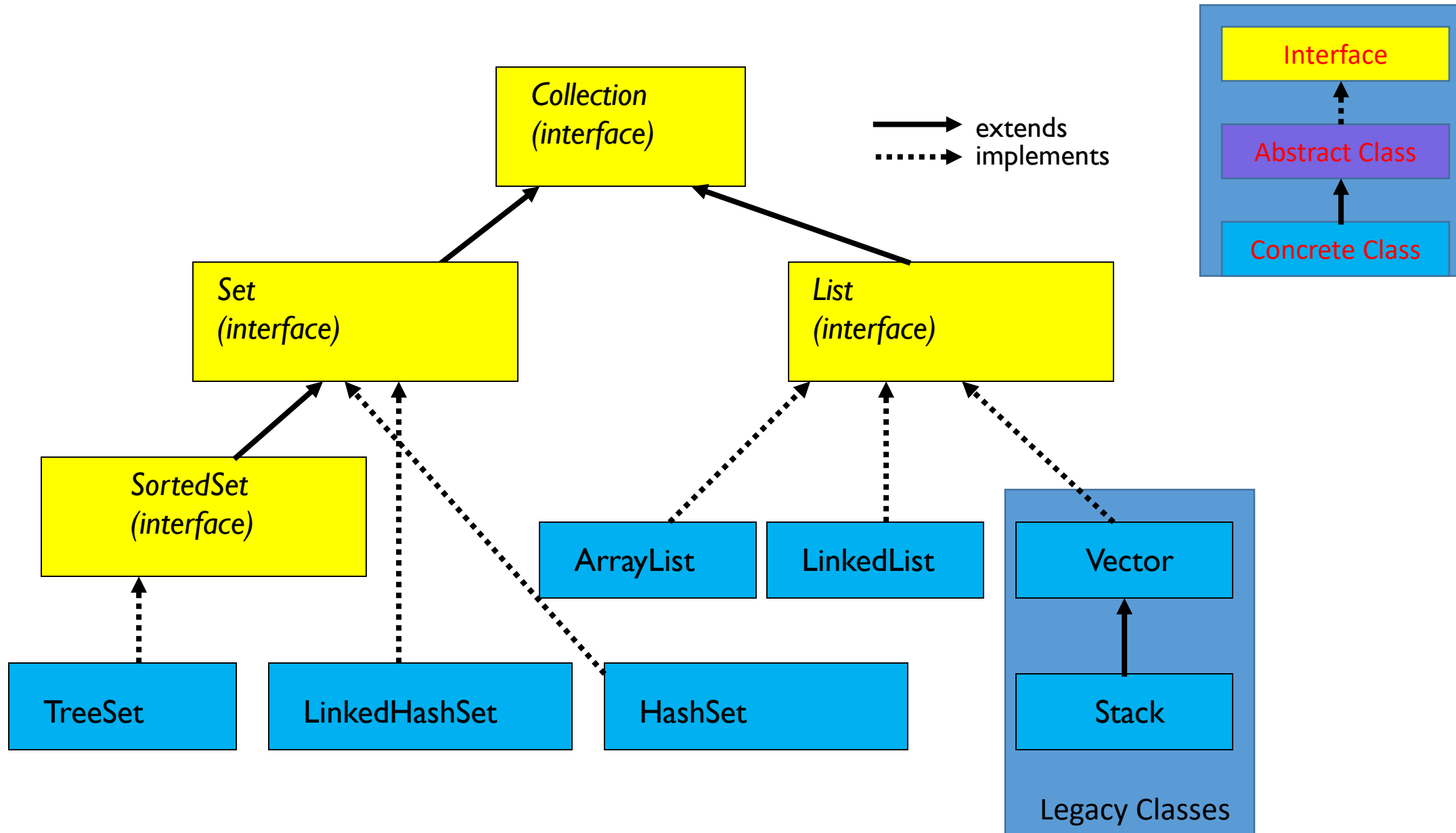
The initial version of Java 1.1 came up with few Collection classes such as **Vector, Stack, Hashtable & Properties**.

The primary advantages of a collections framework are that it:

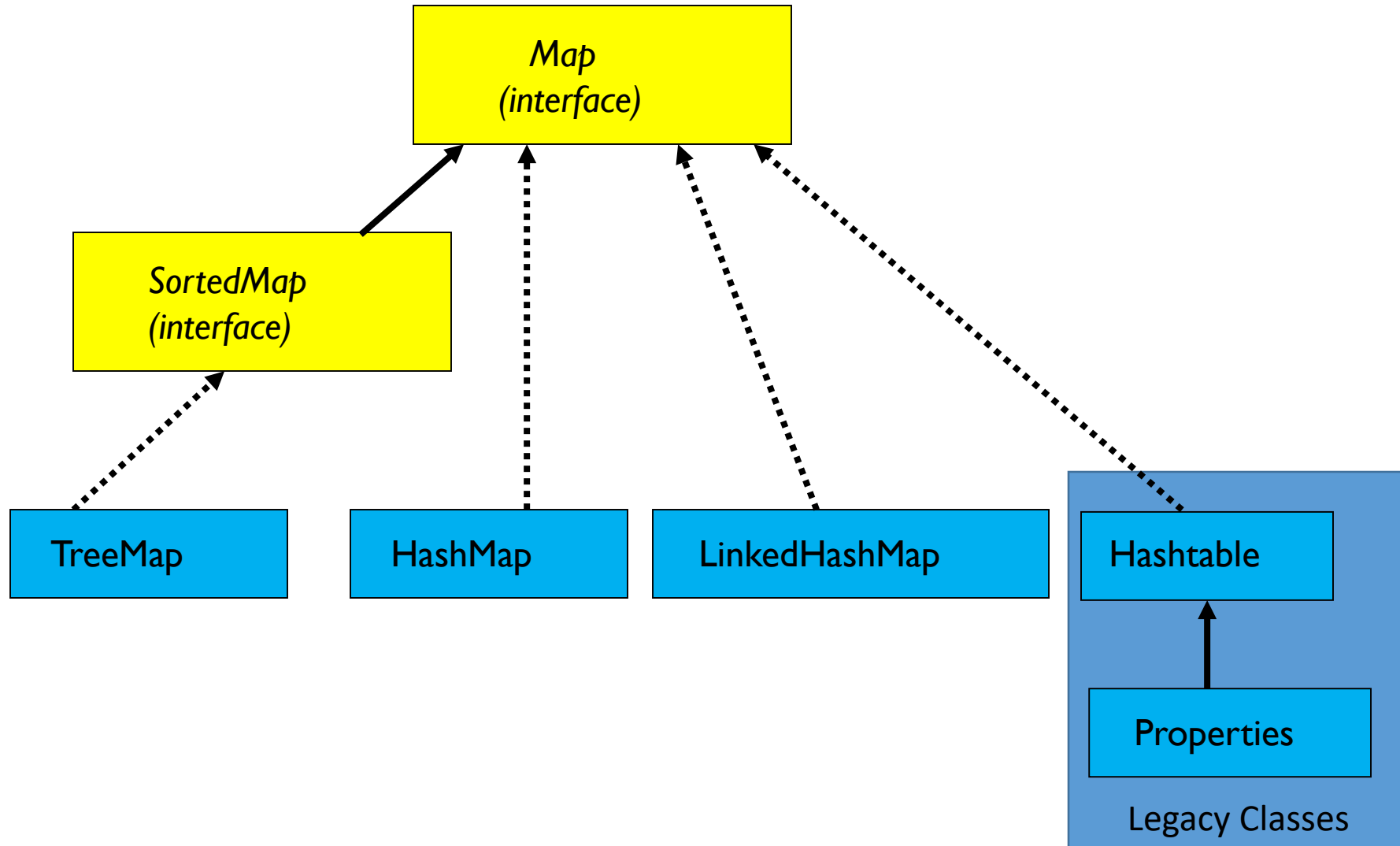
- **Reduces programming effort** by providing data structures and algorithms so you don't have to write them yourself.
- **Increases performance** by providing high-performance implementations of data structures and algorithms.

Note: Collection Framework was introduced in Java 1.2. The legacy classes, **Vector , Stack, Hashtable, Properties** which were prior to Java 1.2 release are retro-fitted into the Collection framework

Collection Framework API



Collection Framework



Collection Framework

The Java Collection Framework contains *two high level interfaces*: **Collection<T>** and **Map<T>**

I. **java.util.Collection<T>** interface

- A collection represents a group of objects, known as its elements.
- **List<T>** interface extends **Collection<T>** interface
 - The main implementation classes are: **ArrayList** and **LinkedList**
 - In case of an **ArrayList**:
 - An ordered collection also known as a sequence *where the element present at a particular index* is known.
 - Values are stored in the order in which they are added unless manipulated.
 - **List allows duplicate objects.**
- Differences between **ArrayList** and **LinkedList**
- **Set<T>** interface extends **Collection<T>** interface
 - **SortedSet<T>** interface extends **Set<T>** interface
- The main implementation classes are: **HashSet**, **LinkedHashSet** and **TreeSet**
- A collection **does not contain duplicate entries**, and at the most one null value allowed.
- Objects being added to a set should implement ***equals and hashCode methods*** otherwise it is possible to add multiple objects to Set which may be intrinsically equivalent.

Collection Framework

The **Deque<T>** interface extends **Collection<T>** interface

- This interface is pronounced as "*deck*", represents a double-ended queue.
- The Deque interface implementations are grouped into general-purpose and concurrent implementations.

General-Purpose Deque Implementations

- The Deque interface supports insertion, removal and retrieval of elements at **both ends**.
- The [ArrayDeque](#) class is the resizable array implementation of the Deque interface, efficient than LinkedList.

II. **java.util.Map<T>** interface

- **SortedMap<T>** interface extends **Map<T>** interface
- **The main implementation classes are HashMap, LinkedHashMap and TreeMap**
 - Map is a top-level interface that provides a **key-value** capability
 - A map stores values by key.
 - The keys must be unique, but the same value can be assigned to multiple keys.
 - Warning: do not use mutable objects for keys!.

Collection Interface

```
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);  
}
```

Collection Interface

java.util.Collection<E> (an interface)

public int size();

- Return number of elements in collection

public boolean isEmpty();

- Return true if collection holds no elements

public boolean add(E x);

- Make sure the collection includes x; returns true if collection has changed (some collections allow duplicates, some don't)

public boolean contains(Object x);

- Returns true if collection contains x (uses equals() method)

public boolean remove(Object x);

- Removes a single instance of x from the collection; returns true if collection has changed

public Iterator<E> iterator();

- Returns an Iterator that steps through elements of collection

java.util.Iterator<E> interface

public boolean hasNext();

- Returns true if the iteration has more elements

public E next();

- Returns the next element in the iteration
- Throws *NoSuchElementException* if no next element

public void remove();

- The element most recently returned by next() is removed from the underlying collection
- Throws *IllegalStateException* if next() not yet called or if remove() already called since last next()
- Throws *UnsupportedOperationException* if remove() not supported

Generics

Code that uses generics has many benefits over non-generic code:

- **Elimination of casts.**

The following code snippet without generics requires casting:

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```



Object java.util.List.get(int index)

- *When re-written to use generics, the code does not require casting:*

```
//List<String> list = new ArrayList<String>(); JDK 6  
List<String> list = new ArrayList<>(); // JDK 7  
list.add("hello");  
String s = list.get(0); // no casting is required
```

Generics

- **Stronger type checks at compile time.**

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- The following code **compiles fine** but throws **ClassCastException** at runtime because we are trying to cast Object in the list to String whereas one of the element is of type *Integer*

```
List list = new ArrayList(); // non-generic declaration of a list
list.add("abc");
list.add(new Integer(5)); //OK

for(Object obj : list){
    String str=(String) obj; //ClassCastException at runtime
}
```

Generics

Now let us use **generics** while declaring a list

```
//List<String> list = new ArrayList<String>(); // Java 6
```

```
List<String> list1 = new ArrayList<>(); // Java 7
```

```
list.add("abc");
```

```
//list.add(new Integer(5)); //compiler error
```

```
for(String str : list){
```

```
    //no type casting needed, avoids ClassCastException runtime exception
```

```
}
```

Notice that at the time of list creation, we have specified that the type of elements in the list will be String. So if we try to add any other type of object in the list, the program will throw compile time error.

Also notice that in for loop, we don't need to type cast the element, hence eliminating *ClassCastException* at runtime.

Additional Methods of Collection<E>

public Object[] toArray()

- Returns a new array containing all the elements of this collection

public <T> T[] toArray(T[] dest)

- Returns an array containing all the elements of this collection, *dest*

Bulk Operations:

- public boolean containsAll(Collection<?> c);
- public boolean addAll(Collection<? extends E> c)
- public boolean removeAll(Collection<?> c);
- public boolean retainAll(Collection<?> c);
- public void clear();

```
ArrayList<String> arrayList = new ArrayList<>();  
arrayList.add("element_1");  
arrayList.add("element_2");  
arrayList.add("element_3");
```

```
Iterator<String> iterator = arrayList.iterator();  
while(iterator.hasNext()){  
    System.out.println(iterator.next());  
}
```

Traversing through Collection Object

- **Enumeration** and **Iterator** interfaces provides capability to iterate through a collection
- New implementations should preferably use **Iterator** or **ListIterator**(traverse either way)

- **Iterator** interface methods

- **boolean hasNext()**
- **Object next()**
- **void remove()**

- **ListIterator** interface methods

- **boolean hasNext();**
- **boolean hasPrevious() ;**
- **Object next();**
- **Object previous();**
- **void remove() ;**

Enumeration interface methods

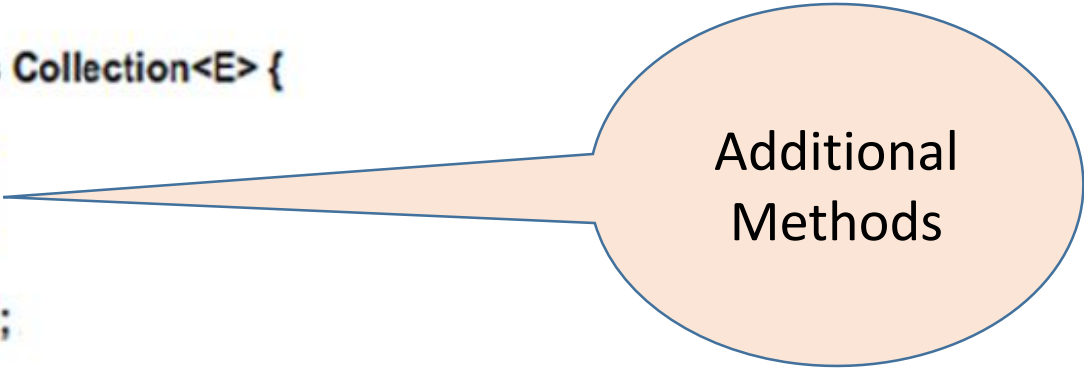
- ***hasMoreElements ()*** to check for the existence of elements in the collection
- ***nextElement ()*** to retrieve the next element

```
Enumeration<String> enumeration = Collections.enumeration(arrayList);  
while(enumeration.hasMoreElements()){  
    System.out.println(enumeration.nextElement());  
}
```

List interface

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).

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



Additional
Methods

Implementations of List interface

- ArrayList
 - > Offers constant-time positional access
 - > Fast
 - > Think of ArrayList as Vector without the synchronization overhead
 - > Most commonly used implementation
- LinkedList
 - > Use it you frequently add elements to the beginning of the List or iterate over the List to delete elements from its interior
- ArrayList default capacity is 10 and default capacity increment is 50 %
- LinkedList by nature does not have "capacity", since it does not allocate memory to the items before the items are added to the list. Each item in a LinkedList holds a pointer to the next in the list.

Collections utility class

This class consists exclusively of static methods that operate on or return collections.

The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are null.

Method Signature	Description
<code>Collections.sort(List myList)</code>	Sort the myList (implementation of any List interface) provided in argument in natural ordering.
<code>Collections.sort(List, comparator c)</code>	Sort the myList(implementation of any List interface) as per comparator c ordering (c class should implement comparator interface)
<code>Collections.shuffle(List myList)</code>	Puts the elements of myList ((implementation of any List interface)in random order
<code>Collections.reverse(List myList)</code>	Reverses the elements of myList ((implementation of any List interface)
<code>Collections.binarySearch(List mlist, T key)</code>	Searches the mlist (implementation of any List interface) for the specified object using the binary search algorithm.
<code>Collections.copy(List dest, List src)</code>	Copy the source List into dest List.
<code>Collections.frequency(Collection c, Object o)</code>	Returns the number of elements in the specified collection class c (which implements Collection interface can be List, Set or Queue) equal to the specified object
<code>Collections.synchronizedCollection(Collection c)</code>	Returns a synchronized (thread-safe) collection backed by the specified collection.

Set<E> interface

- A collection that cannot contain duplicate elements
- Models the mathematical set abstraction and is used to represent sets
 - > cards comprising a poker hand
 - > courses making up a student's schedule
 - > the processes running on a machine
- The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited

Implementation classes of Set<K> interface

HashSet

- HashSet is much faster than TreeSet (constant-time versus log-time for most operations) but offers no ordering guarantees
- Mostly commonly used implementation

TreeSet

- When you need to use the operations in the `SortedSet` interface, or if `value-ordered iteration` is required

LinkedHashSet

- *LinkedHashSet* maintains a linked list of the entries in the set, in the order in which they were inserted.
- This allows insertion-order iteration over the set.

HashSet. Constructs a new, empty set; the backing HashMap instance has **default initial capacity** (16) , **load factor** (0.75) and **capacity increment** is 100%.

For example product of capacity and load factor as $16 * 0.75 = 12$. This represents that after filling up 12th slot into the HashSet , its capacity becomes 32.

HashSet class

- This class implements the Set interface, backed by a hash table (actually a HashMap instance).
- It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.
- This class permits the null element.
- This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets.

Implementing Hashing Technique

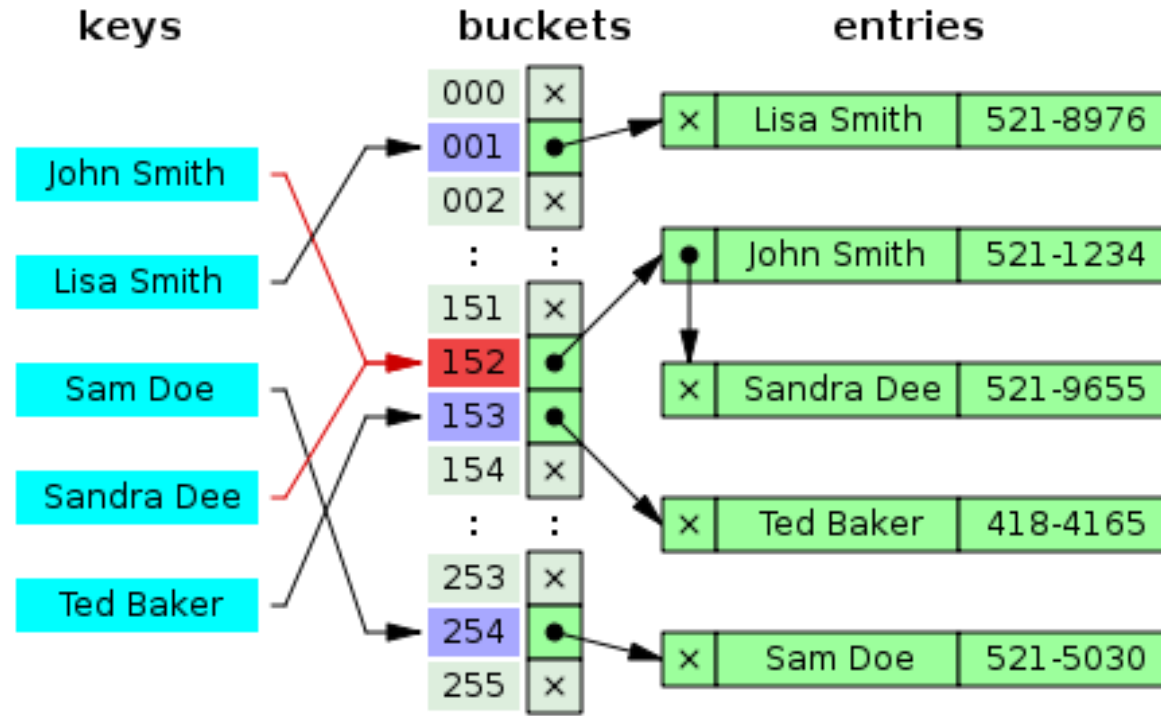
The hash function is applied on the **object's hashCode** and the result of that is taken as a slot or bucket. This will be the index (the *hash*) of an array element(HashMap instance) where the corresponding value is to be sought.

Ideally, the hash function should map each possible key to a unique slot index, but is rarely achievable in practice.

Instead, most hash table designs assume that *hash collisions*—different keys that map to the same hash value—will occur and must be accommodated in some way.

Implementing Hashing Technique

A small phone book as a hash table, where Hash collisions are resolved by separate chaining



- For example, We want to retrieve the value using the key, **Sandra Dee** by calling the **get()** method, when we retrieve a value it will compute the hash code by implicitly calling the **hashCode()** and direct us to a bucket which has two elements.
- Those two elements are scanned sequentially by comparing the keys using their **equals()** method.
- When the key matches we get the respective value. ***This is the reason why the class PhoneBook must override hashCode() and equals() methods.***

HashSet Example

```
public class HashSetDemo {  
  
    public static void main(String[] args) {  
        Set<String> courseSet=new HashSet<>();  
  
        courseSet.add("Core Java-I");  
        courseSet.add("Core Java-II");  
        courseSet.add("Java EE");  
        courseSet.add("Core Java-I");  
  
        System.out.println(courseSet.size());  
  
        System.out.println(courseSet);  
    }  
}
```

```
3  
[Java EE, Core Java-I, Core Java-II]
```

HashSet Example

Effect of **not** overriding *hashCode()* method

Consider the following class, *Student* that doesn't override *hashCode()* method

```
public class Student {  
    private Integer admissionCode;  
    private String studentName;  
    private Integer marks;  
  
    public Student(){  
    }  
    public Student(Integer admissionCode, String studentName, Integer marks) {  
        super();  
        this.admissionCode = admissionCode;  
        this.studentName = studentName;  
        this.marks = marks;  
    }  
    //getter and setter methods  
    // override toString method  
}
```

HashSet Example contd..

```
public class StudentSet {  
  
    public static void main(String[] args) {  
        try{  
            Student student1=new Student(10110,"Ravi Kumar",96);  
            Student student2=new Student(10111,"Jones",78);  
            Student student3=new Student(10110,"Ravi Kumar",96);  
  
            Set<Student> studentSet=new HashSet<>();  
  
            studentSet.add(student1);  
            studentSet.add(student2);  
            boolean flag=studentSet.add(student3);  
            System.out.println(studentSet.size());  
            System.out.println(flag);  
  
        }  
        catch(Exception exception){  
        }  
    }  
}
```

Output:

3

true

For each new object, **default** hashCode() method generates a unique hash code.

All the three objects are placed in the HashSet even though data is same for student1 and student2 objects

HashSet Example contd..

Now **override equals() and hashCode() methods** of Student class and run again the tester class

Output this time will be:

2

false

The overridden hashCode method generate same integer value for two equal objects.

Since hashcode generated for student3 is same as student1, student3 object is not placed in the HashSet

TreeSet Example

```
try{
    Student student1=new Student(10115,"Ravi Kumar",96);
    Student student2=new Student(10113,"Jones",78);
    Student student3=new Student(10110,"Ravi Kumar",96);

    Set<Student> studentSet=new TreeSet<>();

    studentSet.add(student1);
    studentSet.add(student2);
    boolean flag=studentSet.add(student3);
    System.out.println(studentSet.size());
    System.out.println(flag);
    System.out.println(studentSet);

}
catch(Exception exception){
}
```

Exception in thread "main"
java.lang.ClassCastException:
com.Int.businessstier cannot be cast to
java.lang.Comparable

Note: If Student objects are stored in *TreeSet* , Student class must ***implement Comparable interface***.

- Not necessary in case of *HashSet* and *LinkedHashSet*.

TreeSet Example

Modify Student class by implementing Comparable interface and re-run the tester class.

```
public class Student implements Comparable<Student>{
.....
@Override
public int compareTo(Student student2) {
    Student student1=this;
    if(student1.getAdmissionCode()<student2.getAdmissionCode()){
        return -1;
    }else if(student1.getAdmissionCode()>student2.getAdmissionCode()){
        return 1;
    }else{
        return 0;
    }
}
}
```


Map<K,V> Interface

Map

- A collection that provides a **key-value** capability

java.util.Map<K,V>

Set	List
Key	Value
A	Apple
B	Banana
C	Cat
D	Dog

- Maps keys to values.
- A **Map** cannot contain duplicate **keys**; each **key can map** to at most one **value**
- **Map implementations except TreeMap** allow **only one null key** (Since Map does not allow duplicate keys) but a **Map** can have **more than one null values**

Default initial capacity of the HashMap takes is 16 and load factor is 0.75 (i.e 75% of current map size) and capacity increment is 100%. The load factor represents at what level the HashMap capacity should be doubled.

For example product of capacity and load factor as $16 * 0.75 = 12$. This represents that after storing the 12th key – value pair into the HashMap , its capacity becomes 32.

HashMap

- The Map interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings
 - a set of keys -> Use ***keySet()*** to retrieves only keys
 - collection of values -> Use ***values()*** to retrieves only values
 - set of key-value mappings -> Use ***entrySet()*** to retrieve entire Map
- The key is usually a non-mutable object.
- The value object however can be a mutable object.

Map<K,V> and SortedMap<K,V> Interfaces

Map<K,V> Interface

```
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);  
    }  
}
```

SortedMap<K,V> extends Map<K,V> Interface

- 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

- The **Map.Entry** interface enables you to work with a map entry.
- The **entrySet()** method declared by the **Map** interface returns a **Set** containing the map entries.
- Each of these set elements is a **Map.Entry** object.

Implementations of Map interface

HashMap

- makes absolutely *no guarantees* about the *iteration order*. It will even change completely when new elements are added.
- Use it when you want maximum speed and don't care about iteration order
- This is the most commonly used implementation

TreeMap

- Iterates according to the *natural ordering* of the *keys* according to their *compareTo()* method (or an externally supplied Comparator).
- Additionally, it implements the **SortedMap** interface, which contains methods that depend on this sort order.

LinkedHashMap

- iterates in *the order in which the entries were put* into the map.
- Use this implementation if u want near-HashMap performance and insertion-order iteration.

Iterating through a Map<K,V>

```
Map<String,String> hashMap = new HashMap<>();
```

```
for (Map.Entry<String,String> entry : hashMap.entrySet())  
    {  
        System.out.println("Key = "+ entry.getKey() +",Value = "+ entry.getValue());  
    }
```

or

```
Set s = hashMap.entrySet();
```

```
Iterator i = s.iterator();
```

```
while(i.hasNext() ) {
```

```
    Map.Entry entry = (Map.Entry)i.next();
```

```
    System.out.println(entry.getKey() + ":" + entry.getValue());
```

```
}
```



**Iterator i =
hashMap.entrySet().iterator();**

Hashtable – Legacy class

Hashtable stores data in form of **key/value pairs**.

Using the key, the value can be retrieved from the data structure.

Properties of Hashtable

Following are the properties of Hashtable.

- Hashtable is a **legacy class** introduced with JDK 1.0.
- Stores data in **form of key/value** pairs.
- It **does not accept** duplicate keys. If the same key is added twice, the earlier one is replaced (overwritten), that is old value is lost.
- Value can be same for different keys (like value red can be there for keys tomato and apple or two students with roll number 40 and 50 may have the same birthday).
- Key and value **cannot be null** and if added, JVM throws [NullPointerException](#).
- Hashtable methods are synchronized (like Vector).
- Hashtable can be designed for **generics** to accept same type of data.
- Its equivalent in Collection framework is HashMap.
- Unlike the new collection implementations, **Hashtable is synchronized**.

Synchronized Collections

By default Collection Framework implementations are not thread-safe.

By using static methods of utility class, **Collections**, we can convert unsynchronized implementations to synchronized implementations.

```
Collections.synchronizeList();
```

```
Collections.synchronizeMap();
```

```
Collections.synchronizeSet();
```

Though both Synchronized and Concurrent Collection classes provide thread-safety, the differences between them comes in **performance**, **scalability** and how they achieve thread-safety.

Synchronized collections like synchronized HashMap, Hashtable, HashSet, Vector, and synchronized ArrayList are much slower than their concurrent counterparts e.g. ConcurrentHashMap, CopyOnWriteArrayList, and CopyOnWriteHashSet.

Main reason for this slowness is **locking**; synchronized collections locks the whole collection while concurrent collections do not lock the whole Map or List.

Concurrent Collections

The `java.util.concurrent` package includes a number of additions to the Java Collections Framework.

These are categorized by the collection interfaces provided:

- [BlockingQueue](#) defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
- [ConcurrentMap](#) is a sub interface of [java.util.Map](#) that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization.
- The standard general-purpose implementation of `ConcurrentMap` is [ConcurrentHashMap](#), which is a concurrent analog of [HashMap](#).

ConcurrentHashMap

- In Map implementations, If multiple threads access it, for ex., when one thread is reading the HashMap instance while other thread is writing to it, throws *ConcurrentModificationException*.
- To resolve this, use ConcurrentHashMap
- ConcurrentHashMap allows multiple threads to work on its implementation.
- In normal HashMap, there will be only one thread per one implementation.
- For ConcurrentHashMap, there will be one lock per bucket. Default number of buckets are 16.
- So a ConcurrentHashMap implementation has 16 monitors(locks). Hence 16 threads at a time can access it.

Ex.

```
private static Map<Long,Address> addressMap=new ConcurrentHashMap<>();
```

Fail-safe and fail-fast iterators

Java Collections supports two types of Iterators, **fail-safe** and **fail-fast**.

The main distinction between a fail-fast and fail-safe Iterator is whether or not the underlying collection can be modified while its being iterated.

If Iterator detects any structural change after iteration has begun for ex. adding or removing a new element then it throws **ConcurrentModificationException**, this is known as **fail-fast behavior** and these iterators are called *fail-fast iterator* because they fail as soon as they detect any modification.

Fail-safe and fail-fast iterators

Most of the Collection classes from Java 1.4 e.g. Vector, ArrayList, HashMap, HashSet has fail-fast iterators.

Fail-safe iterators were introduced in Java 1.5 when concurrent collection classes e.g. **ConcurrentHashMap**, **CopyOnWriteArrayList** and **CopyOnWriteArraySet** was introduced.

Fail-safe iterators use a **view of original collection** for iteration and that is why they do not throw *ConcurrentModificationException* even **when original collection is modified** after iteration has begun.

Fail-Fast Example

```
public class HashMapFailFast {  
  
    public static void main(String[] args) {  
  
        Map<String,String> myMap = new HashMap<>();  
        myMap.put("1", "1");myMap.put("2", "1");  
        myMap.put("3", "1");myMap.put("4", "1");  
        myMap.put("5", "1");myMap.put("6", "1");  
        System.out.println("HashMap before iterator: "+myMap);  
        Iterator<String> iterator = myMap.keySet().iterator();  
        while(iterator.hasNext()){  
            String key = iterator.next();  
            if(key.equals("3")) {  
                myMap.put(key+"new", "new3");  
            }  
        }  
        System.out.println("HashMap after iterator: "+myMap);  
    }  
}
```

HashMap before iterator: {1=1, 2=1, 3=1, 4=1, 5=1, 6=1}

Exception in thread "main"

[java.util.ConcurrentModificationException](#)

at java.util.HashMap\$HashIterator.nextNode(Unknown Source)

at java.util.HashMap\$KeyIterator.next(Unknown Source)

at

com.Int.presentationtier.HashMapFailFast.main([HashMapFailFast.java:18](#))

Fail-Safe Example

```
public class ConcurrentHashMapFailSafe {  
  
    public static void main(String[] args) {  
        Map<String,String> myMap = new ConcurrentHashMap<String,String>();  
        myMap.put("1", "1"); myMap.put("2", "1");  
        myMap.put("3","3");myMap.put("4", "1");  
        myMap.put("5", "1");  
        System.out.println("ConcurrentHashMap before iterator: "+myMap);  
        Iterator<String> iterator = myMap.keySet().iterator();  
  
        while(iterator.hasNext()){  
            String key = iterator.next();  
            if(key.equals("3")) {  
                myMap.put(key+"new", "new3");  
            }  
        }  
        System.out.println("ConcurrentHashMap after iterator: "+myMap);  
    }  
}
```

Exception handling and fail- fast/ fail-safe

Exception handling in java introduces us to design our programs to be **fail safe**. Use exception handling to fail gracefully and quickly.

Exception should not be carried around in the sequence flow for too long. Golden rule with exception handling is, **throw early and catch late**.

When we hit an exception, it should be thrown immediately. We should not catch an exception unless we are sure what to do with it and the action we take by catching it should negate the failure.



Thank You!