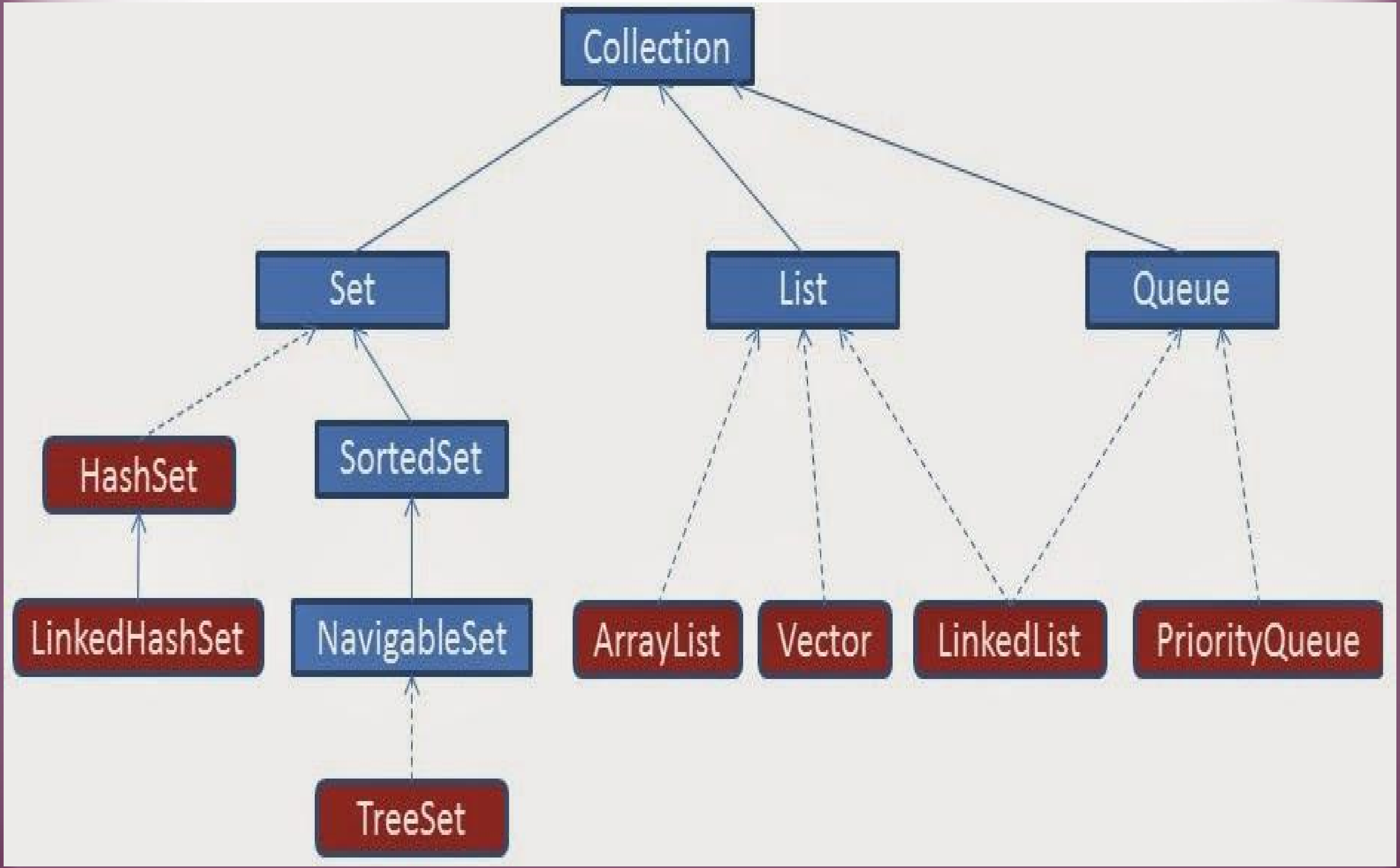
Java Collections Framework

Java Collection Framework

- Provides tools to maintain data container of object
- Alternative to the creation of custom data structures
- Consists of interfaces and classes within the **java.util package**
- **java.util.Collection** is the root **interface** for most of the Java collections
  - There is no *direct* implementation
  - Sub-interfaces are Set and List.
  - Used to pass collections around and manipulate them

Collection Hierarchy



Collection categories

- **Ordered Collection**
  - The objects within the collection are maintained in a particular order or not
  - When an object is added or removed from an ordered collection , the order is automatically maintained

- **Duplicate Elements**
  - Whether or not the collection allows duplicate objects
  - Collection will automatically reject an attempt to add an object

- **Mapping of Key & Value**
  - Whether or not the collection maps a key to a particular object .
  - This provides a means of quickly locating the object.
  - Such collections do not allow duplicate objects

Collections

- **Lists**
  - List of things
  - Classes that implement *List* Interface
- **Sets**
  - Unique things
  - Classes that implement *Set* Interface
- **Maps**
  - Things with unique id
  - Classes that implement M*ap* Interface
- **Queues**
  - Things arranged in order
  - Classes that implement *Queue* Interface

Methods in List Interface

- void **add**(object o)
- void **addAll**(Collection c)
- void **clear()**
- boolean **contains**(object)
- Object **get**(int index)
- int **indexOf**(object o)
- ListIterator **listIterator**()
- Object **set** (int index, Object element)

List Implementations

- **Array List**
  - a resizable-array implementation
    - unsynchronized
- **Linked List**
  - a doubly-linked list implementation
  - if elements frequently inserted/deleted within the List
    - May provide better performance than ArrayList
  - For queues and double-ended queues (deques)
- **Vector**
  - a synchronized resizable-array implementation of a List with additional "legacy" methods.

ArrayList

- Permits all elements, including null.
  - Dynamically add objects in the List.
  - By default, creates an array of size 10.
  - Adding or removing elements will automatically adjust the size

- **ArrayList**()
  - Constructs an empty list with an initial capacity of ten.

- **ArrayList**(int initialCapacity)
  - Constructs an empty list with the specified initial capacity.

Generics

- Introduced in Java 5.0
- Can put **any** Object in to Collections, because they hold Just Objects
  - Type-safe Collections are created ,ensuring type-safety at compile time rather than run-time
- Type erasure Applied For :
  - Stronger type checks at compile time.
  - Elimination of casts.
  - The bytecode, contains only ordinary classes, interfaces, and methods.
  - Type erasure ensures that no new classes are created for parameterized types;
    - Generics incur no runtime overhead.

Generics

- Creating an Instance of Classes of Generic Type

```
ArrayList<Employee> alist=new
                  ArrayList<Employee>();
Employee ramesh = new Employee(101,"Ramesh");

alist.add(ramesh);
```

- 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.

Two Schemes of Traversing Collections

- **for-each**
  - The for-each construct allows you to concisely traverse a collection or array using a for loop
- **for (Object o: collection)**
  - System.out.println(o);
- **Iterator**
  - An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired

ArrayList – Enhanced For Loop

```
public void printElements()
   {
      for(Employee employee :list)
      {
        System.out.println(employee .getEmpname());
            System.out.println(employee .getEmpno());
      }
   }
```

The Iterator Interface

- Provides for the one-way traversal of a Set or SortedSet collection (forward only)
- Has several required methods but the most frequently used are:

Method	**Usage**
`hasNext()`	**Returns `true` if the iteration has more elements**
`next()`	**Returns the next element in the iteration (as a generic `Object`)**

Array List –with Iterator

```
public void dispalyArrayList()
   {
        Iterator<Employee> itr = alist.iterator();
        while(itr.hasNext())
        {
              Employee empObj = itr.next();
              System.out.println(empObj.getEmpname());
              System.out.println(empObj.getEmpno());


        }
   }
```

Array List

- **When to use ArrayList**
  - To fetch data frequently
  - Adding data is not so frequent activity.
- **When not to use ArrayList**
  - When the list is updated frequently

The ListIterator Interface

- Extends the Iterator interface to provide for the two-way traversal of a List collection (forward and backward)
- The most frequently used are:

Method	Usage
haxNext()	Returns `true` if this list iterator has more elements when traversing the list in the forward direction
hasPrevious()	Returns `true` if this list iterator has more elements when traversing the list in the backward direction
next()	Returns the next element in the list (as a generic `Object`)
previous()	Returns the previous element in the list (as a generic `Object`)

The Four Methods

- Collections depends on :

1. equals,
2. compare
3. compareTo,
4. hashCode

- Collection with some sort of *membership test* uses equals
- Collection that depends on *sorting* requires larger/equal/smaller comparisons (compare or compareTo)
- Collection that depends on *hashing* requires both equality testing and hash codes
- While implementing hashCode, implement equals too.

Natural Ordering

- Comparable implementations provide a natural ordering for a class, which allows objects of that class to be sorted automatically
- If the List consists of String elements, it will be sorted into alphabetical order
- String class implements the Comparable interface.
- If the elements do not implement Comparable, **Collections.sort(list)** will throw a ClassCastException.

Comparable Interface

- Implementing Comparable interface requires to implement the method compareTo()

- Imposes a total ordering on the objects of each class that implements it.

- The class's compareTo method is referred to as its *natural comparison method*.

- public int **compareTo**(Object o)

  - Compares this object with the specified object for order.
  - Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Implementing Comparable

```
public class  Book implements Comparable<Book>{
private int bookno;
private String bookname;
private String author;

public int compareTo(Book otherbook) {
  if(this.bookno<otherbook.bookno) return -1;
  if(this.bookno>otherbook.bookno) return 1;
  return 0;
 }
}
```

- To Sort the ArrayList Using Comaprable the static method in the Collectins class is invoked
  - **Collections.sort(arrList);**

Using a separate Comparator

- Can put the comparison method in a separate class that implements Comparator instead of Comparable
- The more flexible Comparator can also be used
- Comparator is in java.util, not java.lang
- Comparable requires a definition of compareTo but Comparator requires a definition of compare

Implementing Comparator

```
import java.util.Comparator;

public class BookNumberComp implements Comparator<Book>
{

public int compare(Book obj1, Book obj2) {

  if(obj1.getBookno()<obj2.getBookno())return -1;
  if(obj1.getBookno()>obj2.getBookno())return 1;
  return 0;
}


}
```

Implementing Multiple Comparators

```
public class BookNumberComp implements Comparator<Book>
  {
   public int compare(Book obj1, Book obj2) {

   if(obj1.getBookno()<obj2.getBookno())return -1;
   if(obj1.getBookno()>obj2.getBookno())return 1;
         return 0;
   }
  }

  public class BookNameComp implements Comparator<Book>
  {
   public int compare(Book obj1, Book obj2) {
    String bk1 = obj1.getBookname();
    String bk2 = obj2.getBookname();
     return bk1.compareTo(bk2);
   }
  }
```

Collections.Sort() with Comparator

```
    public static void main(String[] args) {

            ArrayList<Book> bkList = new ArrayList<Book>();

    Book bk1 = new Book(23,"java","kathy sierra");
    Book bk2 = new Book(101,"j2ee","james gosling");
    Book bk3 = new Book(11,"struts","haffzel manning");

            bkList.add(bk1);
            bkList.add(bk2);
            bkList.add(bk3);

BookComparators bc = new BookComparators();

Collections.sort(bkList, bc);

            for(Book bk:bkList)
            {
                System.out.println(bk.getBookno());
            }
}
```

When to use each

- *The Comparable interface is simpler and less work*
  - The Object of the class implements Comparable
  - A public int compareTo(Object o) method is overriden
  - When no argument constructor is used in TreeSet or TreeMap
- *The Comparator interface is more flexible but slightly more work*
  - Many different classes that implement Comparator
  - Sorting the TreeSet or TreeMap differently with each comparator
  - TreeSet or TreeMap are Constructed using the comparator

Interface Set

- Extends Collection
- An unordered collection of objects
  - can grow as needed
  - easy to add/delete
  - API independent of implementation (HashSet, TreeSet)
  - Elements have no positions
  - Duplicate entries not allowed

TreeSet

- A balanced binary tree implementation
- Imposes an ordering on its elements
- The elements in the TreeSet are sorted in ascending element order, sorted according to the *natural order* of the elements
- It can be sorted by the comparator provided at set creation time
- TreeSet instance performs all key comparisons using its compareTo (or compare) method,

Hash Set

- Non Duplicate
- Un-Ordered
- Internally uses HashMap.
  - Adds a value in 'map' by calling put(E,o);
  - They Key E is the element passed in add
  - 'o' as the value which is a dummy Object created by doing Object o = new Object;
    - Its common for all key's entered in HashMap 'map'.
- Checks the Key is already present by calling the equals method of 'element'.

equals” operation of “Set” Interface

- Set also adds a stronger contract on the behavior of the equals and hashCode operations,
- Set instances to be compared meaningfully even if their implementation types differ
- Two Set instances are equal if they contain the same elements
- The equality is determined by equals and hashcode Implementation

Hash Set – Duplicates Not Allowed

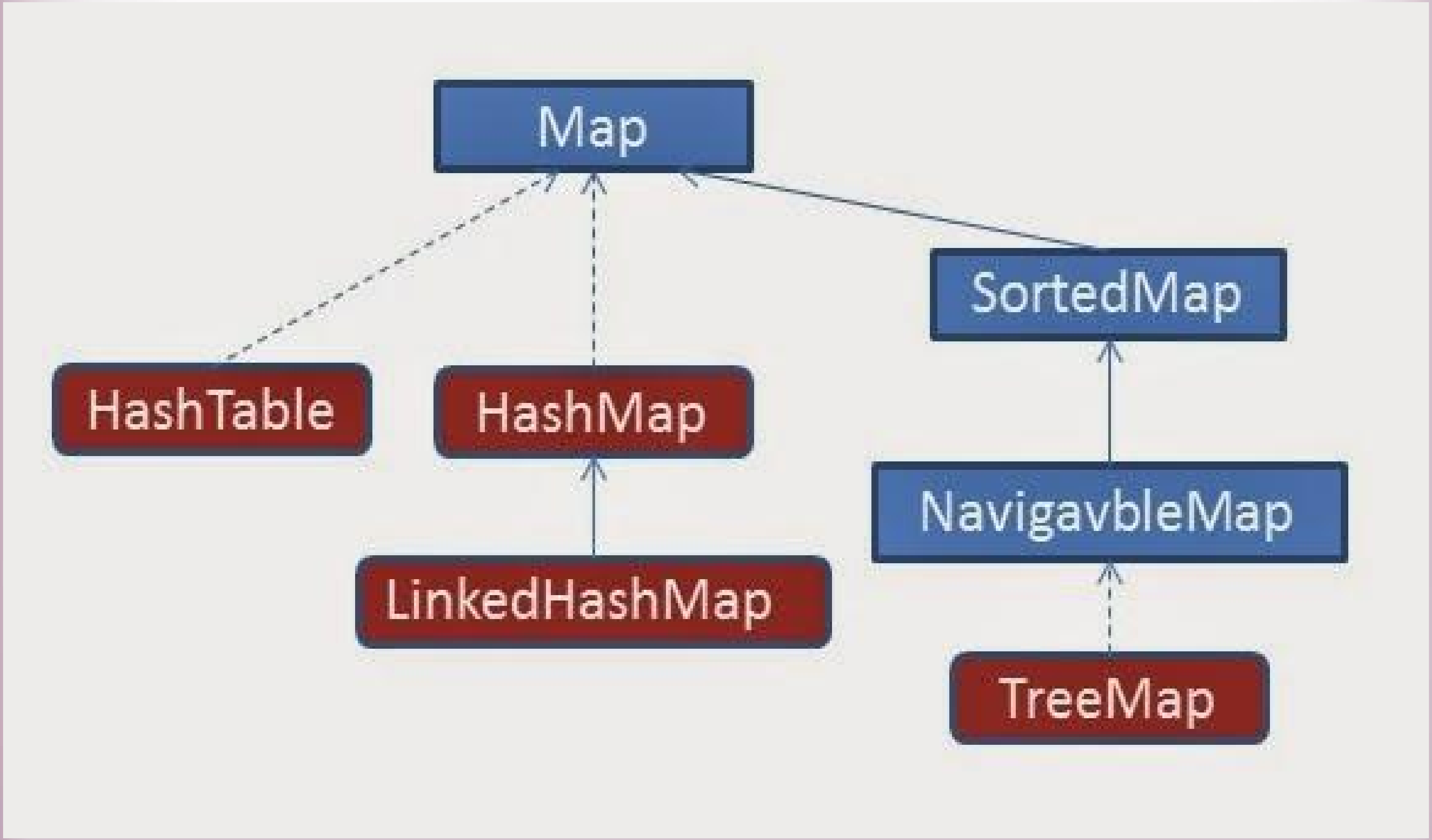
```
public static void main(String args[]) {

    HashSet<Book> hs = new HashSet<Book>();

Book b1= new Book(100,"java","herbert");
Book b2 = new Book(101,"j2ee","kathy siera");
Book b3 = new Book(101,"j2ee","kathy siera");

    hs.add(b1);
    hs.add(b2);

    for(Book bk:hs)
    {
          System.out.println(bk.getBookno());
          System.out.println(bk.getBookname());
    }
  }
```

Tree Set With Comparators

- Tree Set Class overloaded constructors one without argument and another one which takes comparator as arguments.

```
    BookComparators bc = new BookComparators();

 TreeSet<Book> tscomp = new TreeSet<Book>(bc);

    Book bk1 = new Book(23,"java","kathy sierra");
    Book bk2 = new Book(101,"j2ee","james gosling");
    Book bk3 = new Book(11,"struts","haffzel manning");

    tscomp.add(bk1);
    tscomp.add(bk2);
    tscomp.add(bk3);

   }
```

Collection Hierarchy- Map



Why Map is not Extending Collection

- Maps work with key/value pairs
  - They map keys to values.
- Its content can be viewed
  - As a set of keys,
  - A collection of values
  - A set of key-value mappings.
- Mappings are not collections and collections are not mappings.

- Maps can be viewed as Collections of keys, values, or pairs
- It has Collection view operations
  - keySet, entrySet, and values

Map

- interface Map (does not extend Collection)
- An object that maps keys to values
- Ordering may be provided by implementation class, but not guaranteed

- **Methods to Add and Get Objects**
  - Object put(Object key, Object value)
  - Object get(Object key)

- Iterators for keys, values, (key, value) pairs

Map Implementations

- **Hash Map**
  - A hash table implementation of Map
  - Supports null keys & values
  - Not Ordered
- **Tree Map**
  - A balanced binary tree implementation
  - Imposes an ordering on its elements based on comparable/compartor
- **Hash table**
  - Synchronized hash table implementation of Map interface.
  - Doesn't support null keys and values

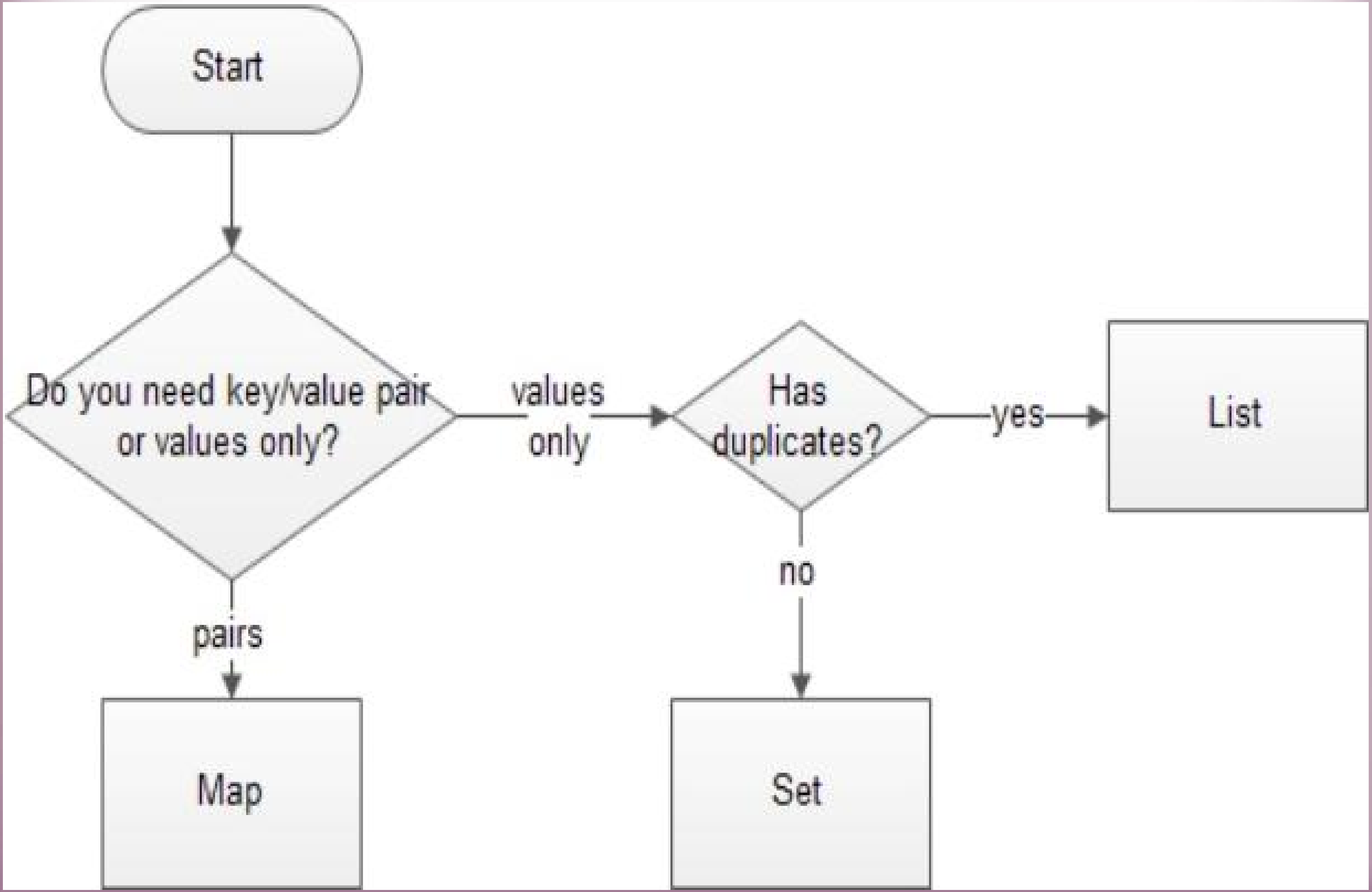
Hash Map

- Works on the principal of hashing.
- It stores values in the form of key-value pair
  - Access a value by providing the key.
- The 'key' element should implement equals() and hashcode() method.
- HashMap stores key-value pair in **Map.Entry** static nested class implementation.

Working of Hash Map

- **put()**
  - Invoked by passing key-value pair,
  - The Entry is stored in the LinkedList, so if there is an already existing entry, it uses equals() method to check if the passed key already exists, if yes it overwrites the value else it creates a new entry and stores this key-value Entry.
- **get()**
  - Invoked by passing Key,
  - It uses the hashCode() to find the index in the array and then use equals() method to find the correct Entry and return its value.

Deciding the Collection Interface


Start
Do you need key/value pair or values only?
values only
Has duplicates?
yes
List
no
pairs
Map
Set