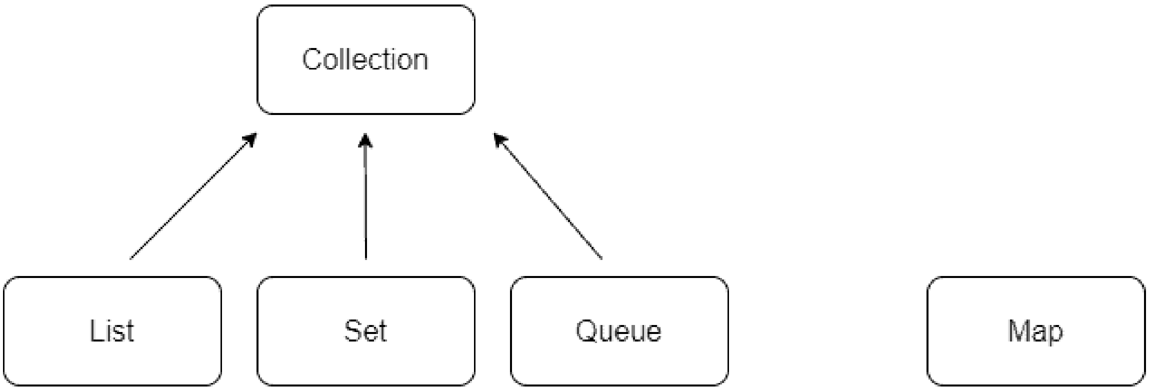
Collection Framework

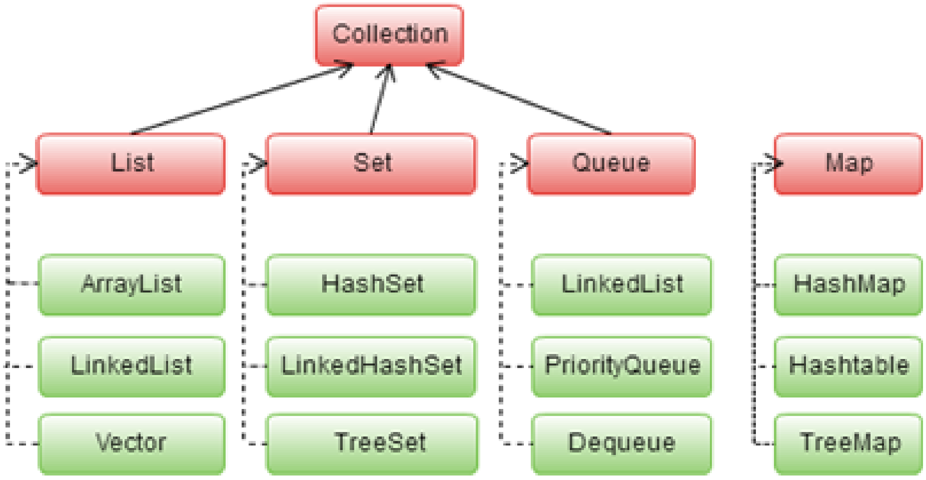
Any group of individual objects that are represented as a single unit is known as a Java Collection of Objects.

Fig: Real-time examples of Collection in Java

1. **Kiddy bank** - collection of coins. (Kiddy bank is collection, coins are objects)
2. **Bag** - collection of books
3. **Earth** - collection of human beings, animals etc.
4. **Classroom** - collection of teachers and students.

Important Collections:





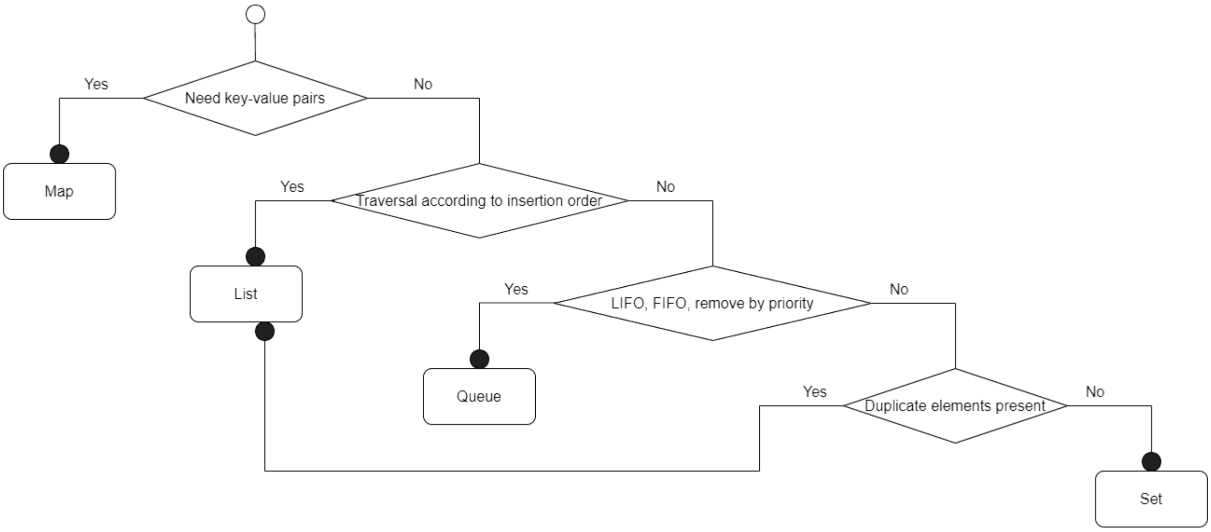
```
public class PiggyBank {
   public static void main(String[] args) {
       List<String> currencyList = new ArrayList<>();
       currencyList.add("Rupee");
       currencyList.add("Dollar");
       currencyList.add("Pound");
       currencyList.add("Euro");
       System.out.println("currencyList :: "+currencyList);

       Set<String> currencySet = new HashSet<>();
       currencySet.add("Rupee");
       currencySet.add("Dollar");
       currencySet.add("Pound");
       currencySet.add("Euro");
       System.out.println("currencySet :: "+currencySet);

       Queue<String> currencyQueue = new PriorityQueue<>();
       currencyQueue.add("Rupee");
       currencyQueue.add("Dollar");
       currencyQueue.add("Pound");
       currencyQueue.add("Euro");
       System.out.println("currencyQueue :: "+currencyQueue);
   }
}
```

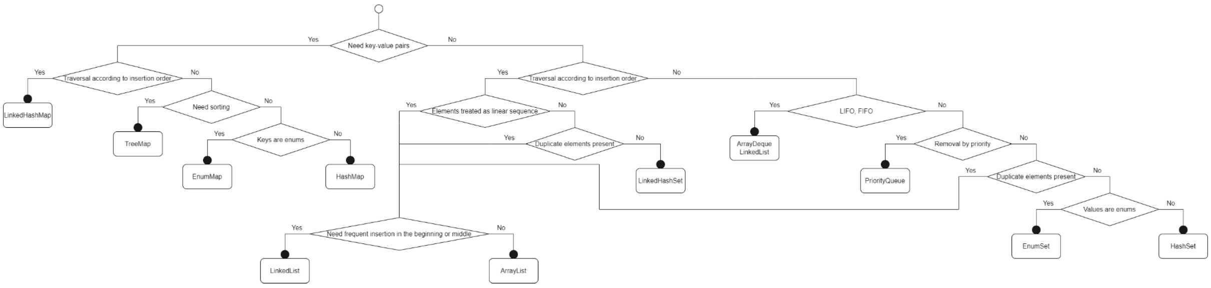
Output:

```
currencyList :: [Rupee, Dollar, Pound, Euro]
currencySet :: [Dollar, Rupee, Pound, Euro]
currencyQueue :: [Dollar, Euro, Pound, Rupee]
```

Java provides the complete tool kit. But, as a programmer, need to have the knowledge of when to use what and why.

It doesn’t make sense to use a queue, in place of a List.


Yes
Need key-value pairs
No
Map
Yes
Traversal according to insertion order
No
List
Yes
LIFO, FIFO, remove by priority
No
Queue
Yes
Duplicate elements present
No
Set



Java Collections Cheat Sheet

Java Concept Of The Day

Basics

What is Java Collection Framework?

Java Collection Framework is a framework which provides some predefined classes and interfaces to store and manipulate the group of objects. Using Java collection framework, you can store the objects as a List or as a Set or as a Queue or as a Map and perform basic operations like adding, removing, updating, sorting, searching etc... with ease.

Why Java Collection Framework?

Earlier, arrays are used to store the group of objects. But, arrays are of fixed size. You can't change the size of an array once it is defined. It causes lots of difficulties while handling the group of objects. To overcome this drawback of arrays, Java Collection Framework is introduced from JDK 1.2.

Java Collections Hierarchy :

All the classes and interfaces related to Java collections are kept in java.util package. List, Set, Queue and Map are four top level interfaces of Java collection framework. All these interfaces (except Map) inherit from java.util.Collection interface which is the root interface in the Java collection framework.

List

Intro :

- List is a sequential collection of objects.
- Elements are positioned using zero-based index.
- Elements can be inserted or removed or retrieved from any arbitrary position using an integer index.

Popular Implementations :

- ArrayList, Vector And LinkedList

Internal Structure :

- **ArrayList :** Internally uses re-sizable array which grows or shrinks as we add or delete elements.
- **Vector :** Same as ArrayList but it is synchronized.
- **LinkedList :** Elements are stored as Nodes where each node consists of three parts – Reference To Previous Element, Value Of The Element and Reference To Next Element.

Null Elements :

- **ArrayList :** Yes
- **Vector :** Yes
- **LinkedList :** Yes

Duplicate Elements :

- **ArrayList :** Yes
- **Vector :** Yes
- **LinkedList :** Yes

Order Of Elements :

- **ArrayList :** Insertion Order
- **Vector :** Insertion Order
- **LinkedList :** Insertion Order

Synchronization :

- **ArrayList :** Not synchronized
- **Vector :** Synchronized
- **LinkedList :** Not synchronized

Performance :

- **ArrayList :** Insertion -> O(1) (if insertion causes restructuring of internal array, it will be O(n)), Removal -> O(1) (if removal causes restructuring of internal array, it will be O(n)), Retrieval -> O(1)
- **Vector :** Similar to ArrayList but little slower because of synchronization.
- **LinkedList :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(n)

When to use?

- **ArrayList :** Use it when more search operations are needed then insertion and removal.
- **Vector :** Use it when you need synchronized list.
- **LinkedList :** Use it when insertion and removal are needed frequently.

Queue

Intro :

- Queue is a data structure where elements are added from one end called tail of the queue and elements are removed from another end called head of the queue.
- Queue is typically FIFO (First-In-First-Out) type of data structure.

Popular Implementations :

- PriorityQueue, ArrayDeque and LinkedList (implements List also)

Internal Structure :

- **PriorityQueue :** It internally uses re-sizable array to store the elements and a Comparator to place the elements in some specific order.
- **ArrayDeque :** It internally uses re-sizable array to store the elements.

Null Elements :

- **PriorityQueue :** Not allowed
- **ArrayDeque :** Not allowed

Duplicate Elements :

- **PriorityQueue :** Yes
- **ArrayDeque :** Yes

Order Of Elements :

- **PriorityQueue :** Elements are placed according to supplied Comparator or in natural order if no Comparator is supplied.
- **ArrayDeque :** Supports both LIFO and FIFO

Synchronization :

- **PriorityQueue :** Not synchronized
- **ArrayDeque :** Not synchronized

Performance :

- **PriorityQueue :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(1)
- **ArrayDeque :** Insertion -> O(1) , Removal -> O(n), Retrieval -> O(1)

When to use?

- **PriorityQueue :** Use it when you want a queue of elements placed in some specific order.
- **ArrayDeque :** You can use it as a queue OR as a stack.

Set

Intro :

- Set is a linear collection of objects with no duplicates.
- Set interface does not have its own methods. All its methods are inherited from Collection interface. It just applies restriction on methods so that duplicate elements are always avoided.

Popular Implementations :

- HashSet, LinkedHashSet and TreeSet

Internal Structure :

- **HashSet :** Internally uses HashMap to store the elements.
- **LinkedHashSet :** Internally uses LinkedHashMap to store the elements.
- **TreeSet :** Internally uses TreeMap to store the elements.

Null Elements :

- **HashSet :** Maximum one null element
- **LinkedHashSet :** Maximum one null element.
- **TreeSet :** Doesn't allow even a single null element

Duplicate Elements :

- **HashSet :** Not allowed
- **LinkedHashSet :** Not allowed
- **TreeSet :** Not allowed

Order Of Elements :

- **HashSet :** No order
- **LinkedHashSet :** Insertion order
- **TreeSet :** Elements are placed according to supplied Comparator or in natural order if no Comparator is supplied.

Synchronization :

- **HashSet :** Not synchronized
- **LinkedHashSet :** Not synchronized
- **TreeSet :** Not synchronized

Performance :

- **HashSet :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **LinkedHashSet :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **TreeSet :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(log(n))

When to use?

- **HashSet :** Use it when you want only unique elements without any order.
- **LinkedHashSet :** Use it when you want only unique elements in insertion order.
- **TreeSet :** Use it when you want only unique elements in some specific order.

Map

Intro :

- Map stores the data in the form of key-value pairs where each key is associated with a value.
- Map interface is part of Java collection framework but it doesn't inherit Collection interface.

Popular Implementations :

- HashMap, LinkedHashMap And TreeMap

Internal Structure :

- **HashMap :** It internally uses an array of buckets where each bucket internally uses linked list to hold the elements.
- **LinkedHashMap :** Same as HashMap but it additionally uses a doubly linked list to maintain insertion order of elements.
- **TreeMap :** It internally uses Red-Black tree.

Null Elements :

- HashMap : Only one null key and can have multiple null values
- LinkedHashMap : Only one null key and can have multiple null values.
- TreeMap : Doesn't allow even a single null key but can have multiple null values.

Duplicate Elements :

- **HashMap :** Doesn't allow duplicate keys but can have duplicate values.
- **LinkedHashMap :** Doesn't allow duplicate keys but can have duplicate values.
- **TreeMap :** Doesn't allow duplicate keys but can have duplicate values.

Order Of Elements :

- **HashMap :** No Order
- **LinkedHashMap :** Insertion Order
- **TreeMap :** Elements are placed according to supplied Comparator or in natural order of keys if no Comparator is supplied.

Synchronization :

- **HashMap :** Not synchronized
- **LinkedHashMap :** Not Synchronized
- **TreeMap :** Not Synchronized

Performance :

- **HashMap :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **LinkedHashMap :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **TreeMap :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(log(n))

When to use?

- **HashMap :** Use it if you want only key-value pairs without any order.
- **LinkedHashMap :** Use it if you want key-value pairs in insertion order.
- **TreeMap :** Use it when you want key-value pairs sorted in some specific order.

Code Refactor:

```
public class PiggyBankRefactored {
   public static void main(String[] args) {
       printCurrencyList();
       printCurrencySet();
       printCurrencyQueue();
   }

   private static void printCurrencyQueue() {
       Queue<String> currencyQueue = new PriorityQueue<>();
       currencyQueue.add("Rupee");
       currencyQueue.add("Dollar");
       currencyQueue.add("Pound");
       currencyQueue.add("Euro");
       System.out.println("currencyQueue :: "+currencyQueue);
   }

   private static void printCurrencySet() {
       Set<String> currencySet = new HashSet<>();
       currencySet.add("Rupee");
       currencySet.add("Dollar");
       currencySet.add("Pound");
       currencySet.add("Euro");
       System.out.println("currencySet :: "+currencySet);
   }

   private static void printCurrencyList() {
       List<String> currencyList = new ArrayList<>();
       currencyList.add("Rupee");
       currencyList.add("Dollar");
       currencyList.add("Pound");
       currencyList.add("Euro");
       System.out.println("currencyList :: "+currencyList);
   }
}
```

Output:

```
currencyList :: [Rupee, Dollar, Pound, Euro]
currencySet :: [Dollar, Rupee, Pound, Euro]
currencyQueue :: [Dollar, Euro, Pound, Rupee]
```

Using custom Java Objects:

```
public class Student {
   private int studentId;
   private String studentName;
   private double marks;
   private String grade;

   public Student(int studentId, String studentName, double marks, String grade) {
       this.studentId = studentId;
       this.studentName = studentName;
       this.marks = marks;
       this.grade = grade;
   }

   public int getStudentId() {
       return studentId;
   }

   public void setStudentId(int studentId) {
       this.studentId = studentId;
   }

   public String getStudentName() {
       return studentName;
   }

   public void setStudentName(String studentName) {
       this.studentName = studentName;
   }

   public double getMarks() {
       return marks;
   }

   public void setMarks(double marks) {
       this.marks = marks;
   }

   public String getGrade() {
       return grade;
   }

   public void setGrade(String grade) {
       this.grade = grade;
   }

   @Override
   public String toString() {
       return new StringJoiner(", ", Student.class.getSimpleName() + "[", "]")
               .add("studentId=" + studentId)
               .add("studentName='" + studentName + "'")
               .add("marks=" + marks)
               .add("grade='" + grade + "'")
               .toString();
   }
}
```

```java
public class StudentCollectionOperations {
  public static void main(String[] args) {
      List<Student> studentList = new ArrayList<>();
      studentList.add(new Student(1,"James",65,null));
      studentList.add(new Student(2,"Richard",90,null));
      studentList.add(new Student(3,"Gary",50,null));
      System.out.println(studentList);
      for(Student student : studentList){
        if(student.getMarks()>=70){
            student.setGrade("A");
        } else if(student.getMarks()>=60){
            student.setGrade("B");
        } else {
            student.setGrade("C");
        }
      }
      System.out.println(studentList);
      Collections.sort(studentList,
                      (Comparator.comparing(Student::getStudentName)));
      System.out.println(studentList);
  }
}
```

Output:

```
[Student[studentId=1, studentName='James', marks=65.0, grade='null'], Student[studentId=2, studentName='Richard',
marks=90.0, grade='null'], Student[studentId=3, studentName='Gary', marks=50.0, grade='null']]

[Student[studentId=1, studentName='James', marks=65.0, grade='B'], Student[studentId=2, studentName='Richard',
marks=90.0, grade='A'], Student[studentId=3, studentName='Gary', marks=50.0, grade='C']]

[Student[studentId=3, studentName='Gary', marks=50.0, grade='C'], Student[studentId=1, studentName='James', marks=65.0,
grade='B'], Student[studentId=2, studentName='Richard', marks=90.0, grade='A']]
```

After refactor:

```java
public class StudentCollectionOperationsRefactored {
  public static void main(String[] args) {
      List<Student> studentList = initStudentDetails();
      System.out.println(studentList);

      updateStudentGrades(studentList);
      System.out.println(studentList);

      sortStudentsBasedOnNames(studentList);
      System.out.println(studentList);
  }

  private static void sortStudentsBasedOnNames(List<Student> studentList) {
      Collections.sort(studentList,
                        (Comparator.comparing(Student::getStudentName)));
  }

  private static List<Student> initStudentDetails() {
      List<Student> studentList = new ArrayList<>();
      studentList.add(new Student(1,"James",65,null));
      studentList.add(new Student(2,"Richard",90,null));
      studentList.add(new Student(3,"Gary",50,null));
      return studentList;
  }
```

```
    private static void updateStudentGrades(List<Student> studentList) {
        for(Student student : studentList){
            if(student.getMarks()>=70){
                student.setGrade("A");
            } else if(student.getMarks()>=60){
                student.setGrade("B");
            } else {
                student.setGrade("C");
            }
        }
    }
}
```

Output:

```
[Student[studentId=1, studentName='James', marks=65.0, grade='null'], Student[studentId=2, studentName='Richard',
marks=90.0, grade='null'], Student[studentId=3, studentName='Gary', marks=50.0, grade='null']]

[Student[studentId=1, studentName='James', marks=65.0, grade='B'], Student[studentId=2, studentName='Richard',
marks=90.0, grade='A'], Student[studentId=3, studentName='Gary', marks=50.0, grade='C']]

[Student[studentId=3, studentName='Gary', marks=50.0, grade='C'], Student[studentId=1, studentName='James', marks=65.0,
grade='B'], Student[studentId=2, studentName='Richard', marks=90.0, grade='A']]
```