java.util

Shristi Technology Labs

Contents

- Collection Framework – an Overview
- List
- Set
- Map
- Properties, Random, UUID
- Calendar
- Locale
- Regular Expressions
- Example for functional Interfaces

Collection Framework

- **Java collections framework** (**JCF**) is a set of classes and interfaces that implement commonly reusable collection data structures
- JCF provides both interfaces that define various collections and classes that implement them.
- Allows collection of objects to be treated as single unit
- From java.util package

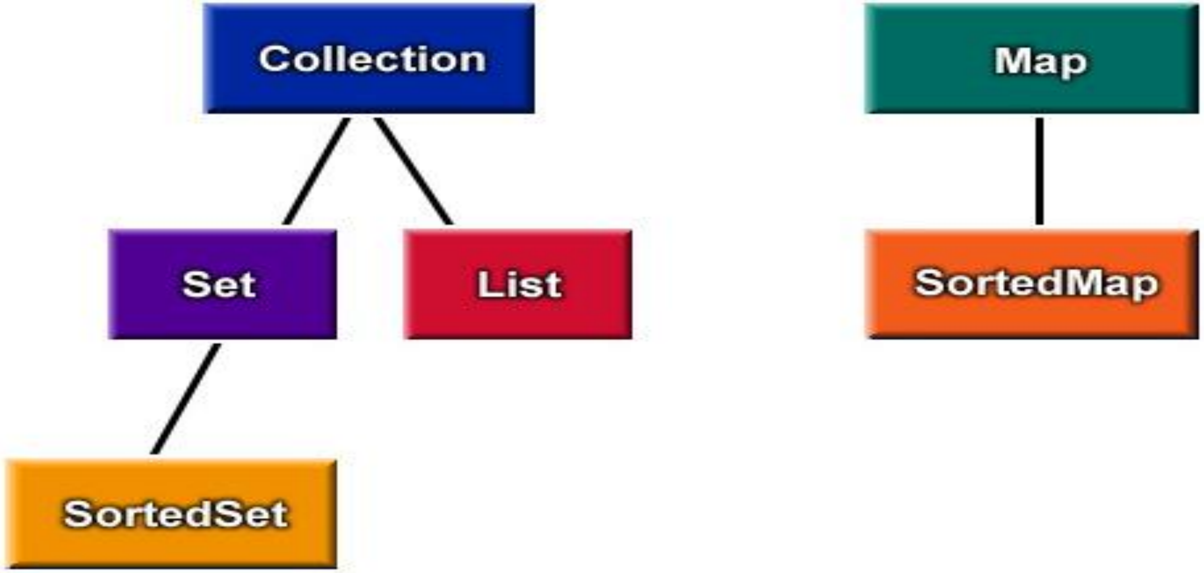
Collection Framework

- A collection is a group of objects called elements
- Collections may be ordered or unordered or sorted
- Some Collections accept duplicates while some do not

- Collection Framework has:
  - **Interfaces**: abstract data types representing collections.
  - **Implementations**: concrete implementations of the collection interfaces. They are reusable data structures.
  - **Algorithms**: methods to perform useful computations, like searching and sorting on objects that implement collection interfaces.

Interfaces

- The *core collection interfaces* that are used to manipulate collections, and to pass them from one method to another.

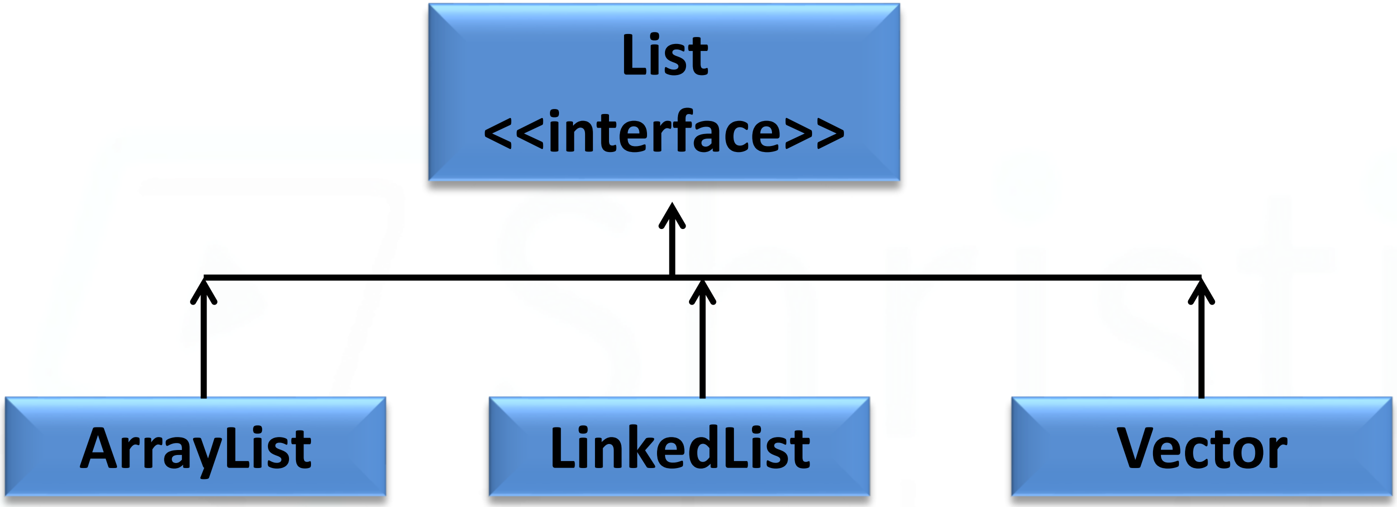


List

- Is an ordered collection
- Elements can be added or removed by index
- Behaves like a variable-size array
- Order of the items is well defined as they are added by index
- Allows positional access of elements

Classes that implement List

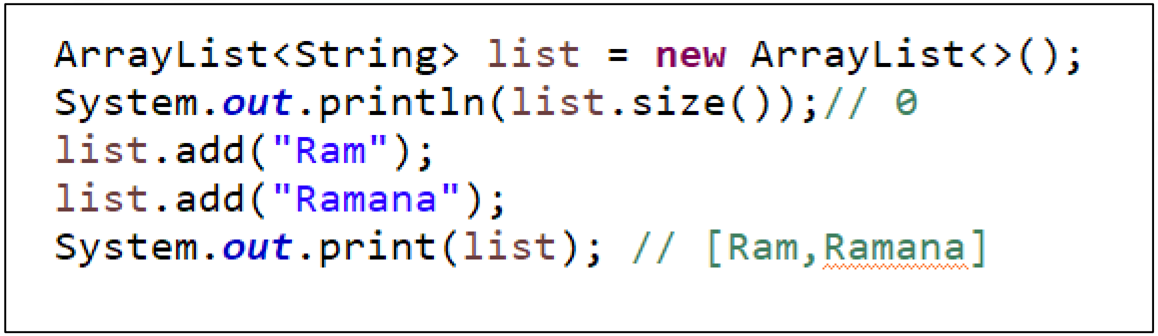
- ArrayList
- LinkedList
- Vector


List
<<interface>>
ArrayList
LinkedList
Vector

ArrayList

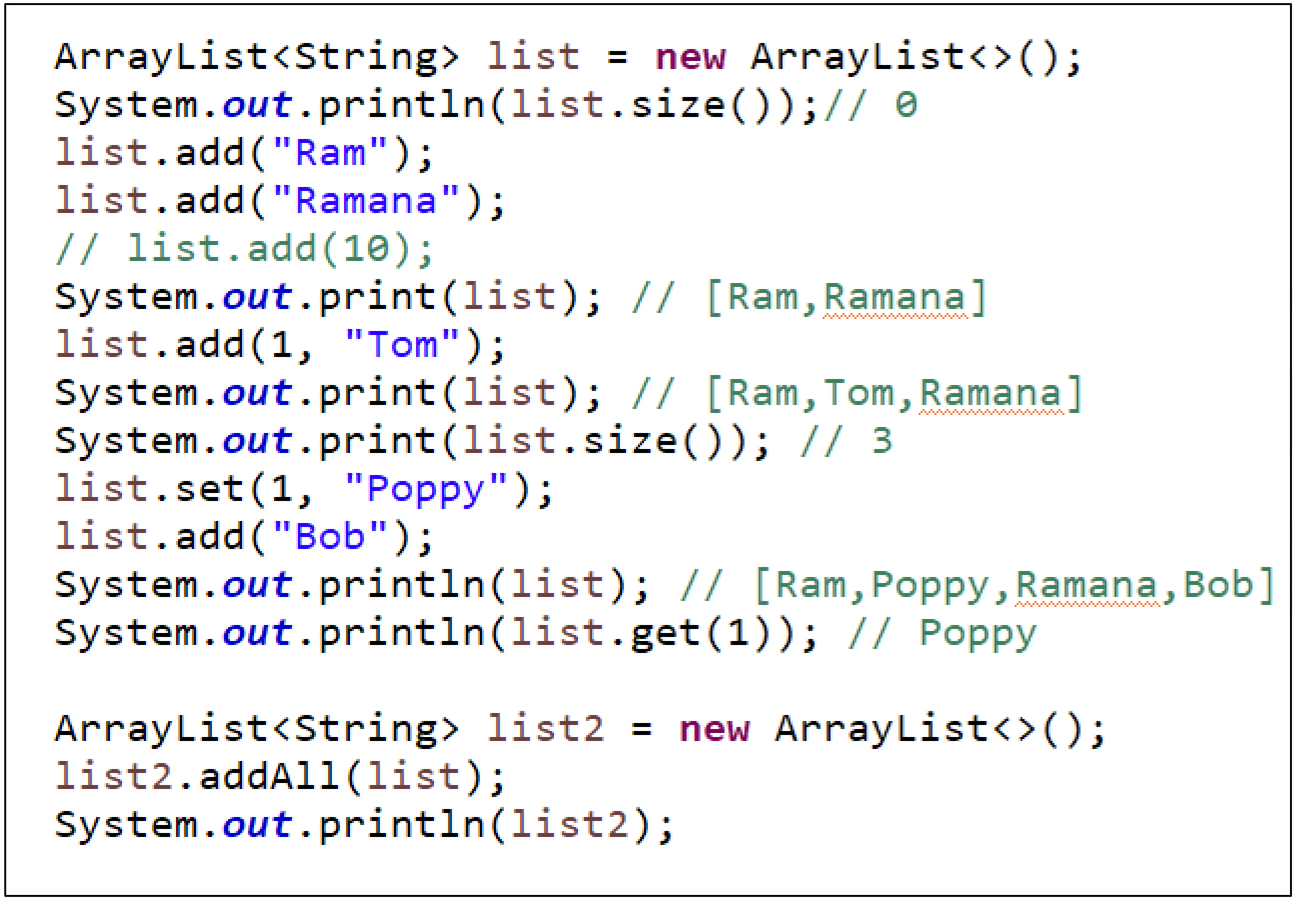
- **ArrayList** class provides resizable-array
- Is an ordered collection and implements List interface
- Allows duplicate elements including null.

eg.



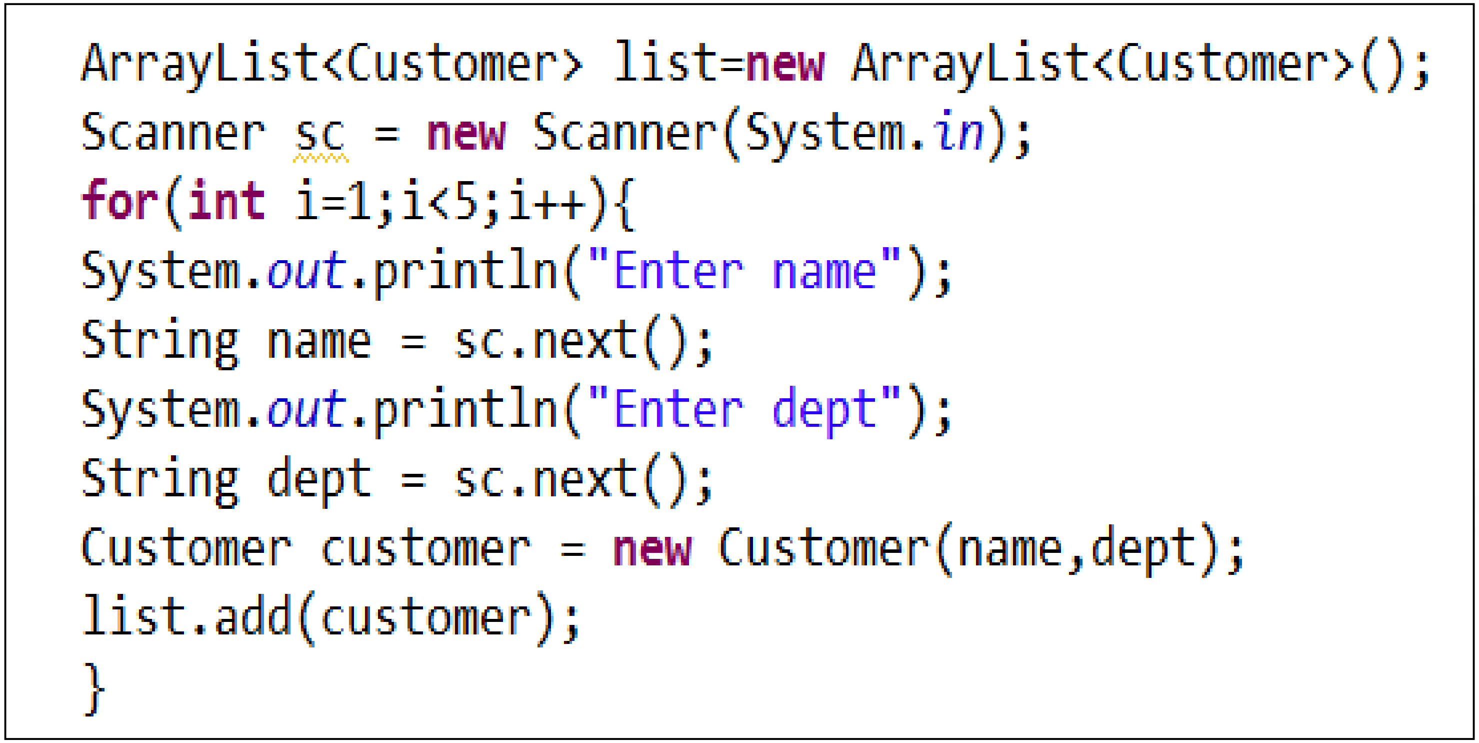
```java
ArrayList<String> list = new ArrayList<>();
System.out.println(list.size());// 0
list.add("Ram");
list.add("Ramana");
System.out.print(list); // [Ram,Ramana]
```

Example



```
ArrayList<String> list = new ArrayList<>();
System.out.println(list.size());// 0
list.add("Ram");
list.add("Ramana");
// list.add(10);
System.out.print(list); // [Ram,Ramana]
list.add(1, "Tom");
System.out.print(list); // [Ram,Tom,Ramana]
System.out.print(list.size()); // 3
list.set(1, "Poppy");
list.add("Bob");
System.out.println(list); // [Ram,Poppy,Ramana,Bob]
System.out.println(list.get(1)); // Poppy

ArrayList<String> list2 = new ArrayList<>();
list2.addAll(list);
System.out.println(list2);
```

Example adding Customer objects



```
ArrayList<Customer> list=new ArrayList<Customer>();
Scanner sc = new Scanner(System.in);
for(int i=1;i<5;i++){
System.out.println("Enter name");
String name = sc.next();
System.out.println("Enter dept");
String dept = sc.next();
Customer customer = new Customer(name,dept);
list.add(customer);
}
```

Iterator and ListIterator

- Is used to access the elements of collection object in sequential manner without knowing its underlying representation.

- Are interfaces based on Iterator design pattern

- Iterator is only for forward access

- ListIterator can iterate the elements in both forward and backward direction

Iterator

- **iterator()** method is used to return an iterator over the elements.

```
public Iterator<E> iterator()
```

Methods in Iterator

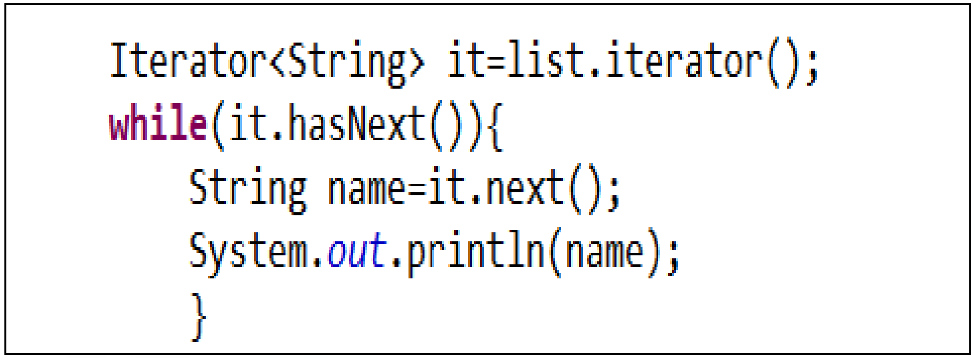
```
boolean hasNext()
  E next()
  void remove()
```

Few Methods in ListIterator

```
boolean hasPrevious()
E previous()
int nextIndex()
```

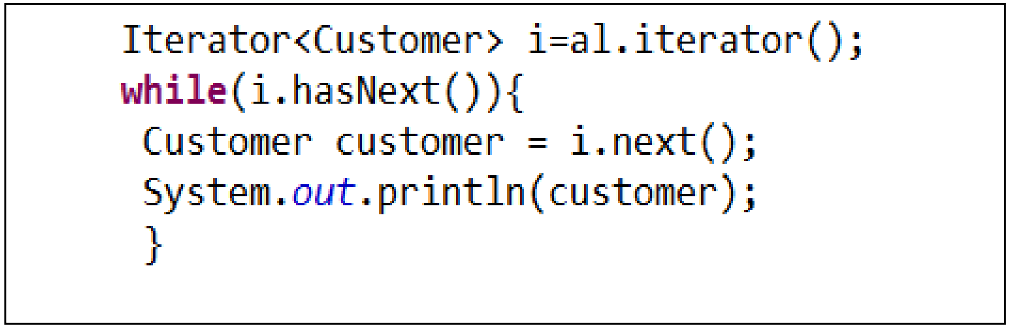
Example for Iterator

For String objects



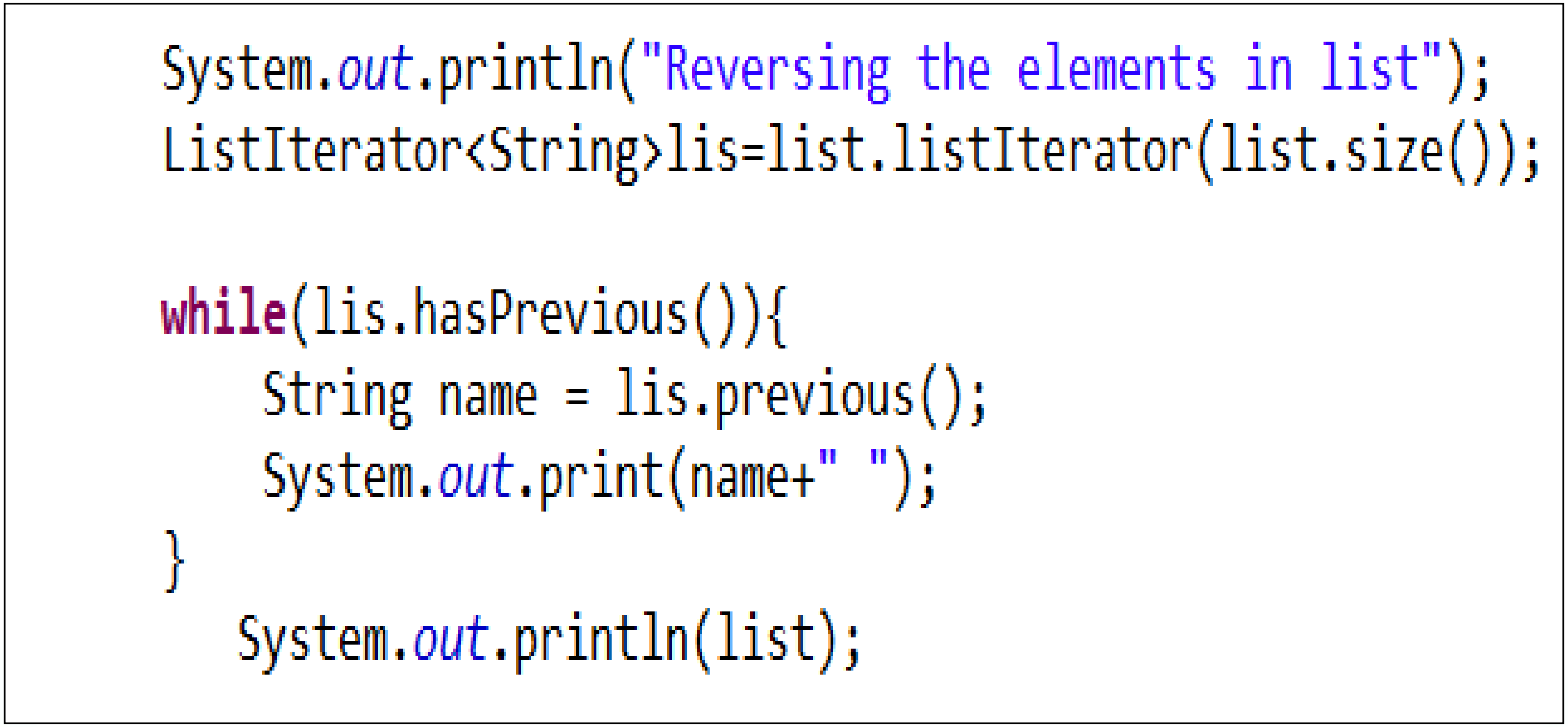
```
    Iterator<String> it=list.iterator();
    while(it.hasNext()){
        String name=it.next();
        System.out.println(name);
        }
```

For user defined(Customer) objects



```
    Iterator<Customer> i=al.iterator();
    while(i.hasNext()){
     Customer customer = i.next();
     System.out.println(customer);
     }
```

Example for ListIterator for reversing the list

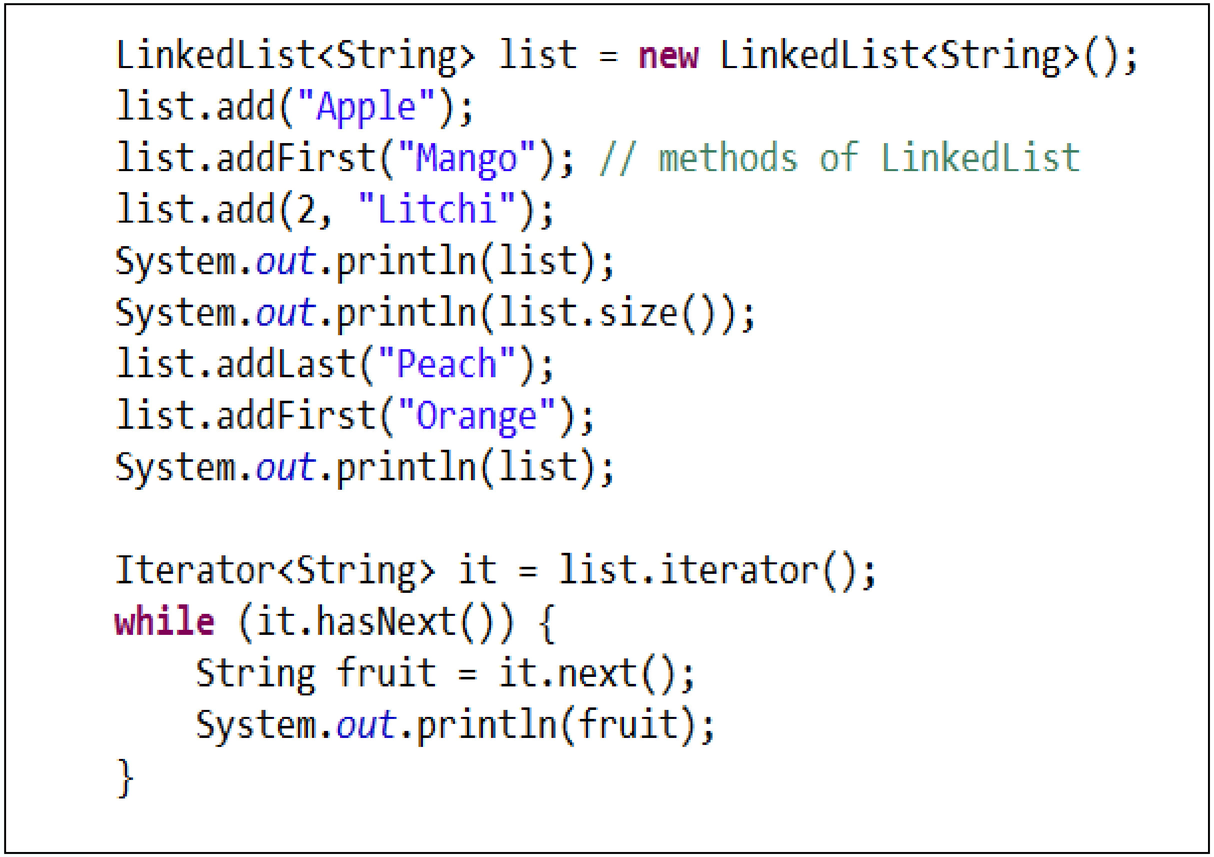


```java
System.out.println("Reversing the elements in list");
ListIterator<String>lis=list.listIterator(list.size());

while(lis.hasPrevious()){
    String name = lis.previous();
    System.out.print(name+" ");
}
   System.out.println(list);
```

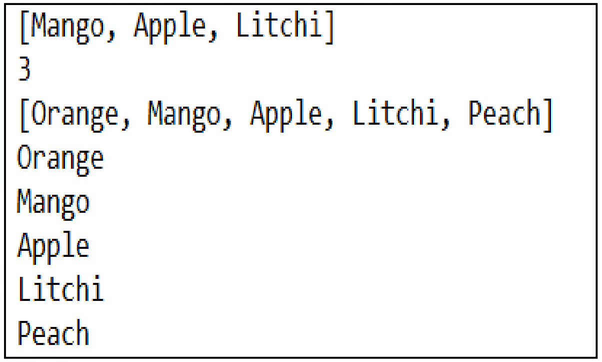
LinkedList

- **LinkedList** is an ordered set of data elements, each containing a link to its successor and sometimes its predecessor
- Each node points to next node by a pointer.
- Implements List and Queue interface
- Elements can be added in front or at the end of a collection
- Can be used for faster insertion

Example for LinkedList



```
LinkedList<String> list = new LinkedList<String>();
list.add("Apple");
list.addFirst("Mango"); // methods of LinkedList
list.add(2, "Litchi");
System.out.println(list);
System.out.println(list.size());
list.addLast("Peach");
list.addFirst("Orange");
System.out.println(list);

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



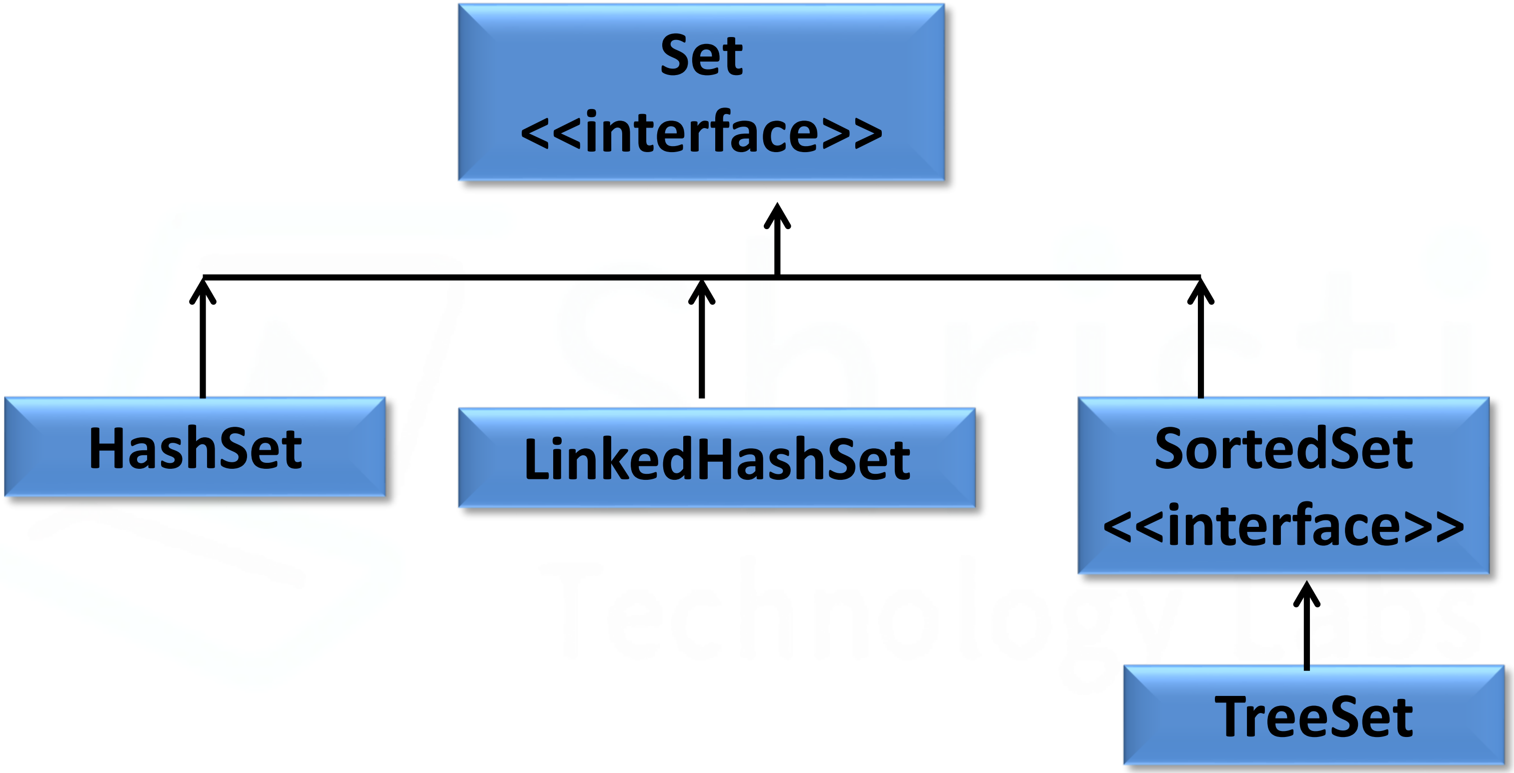
```
[Mango, Apple, Litchi]
3
[Orange, Mango, Apple, Litchi, Peach]
Orange
Mango
Apple
Litchi
Peach
```

Set

- Is a type of collection which does not accept duplicates
- Has both unordered and sorted collection
- Adds a stronger contract on the use of hashcode and equals method.
- This allows set instances to be compared meaningfully even if their implementation types differ.

Classes that implement Set

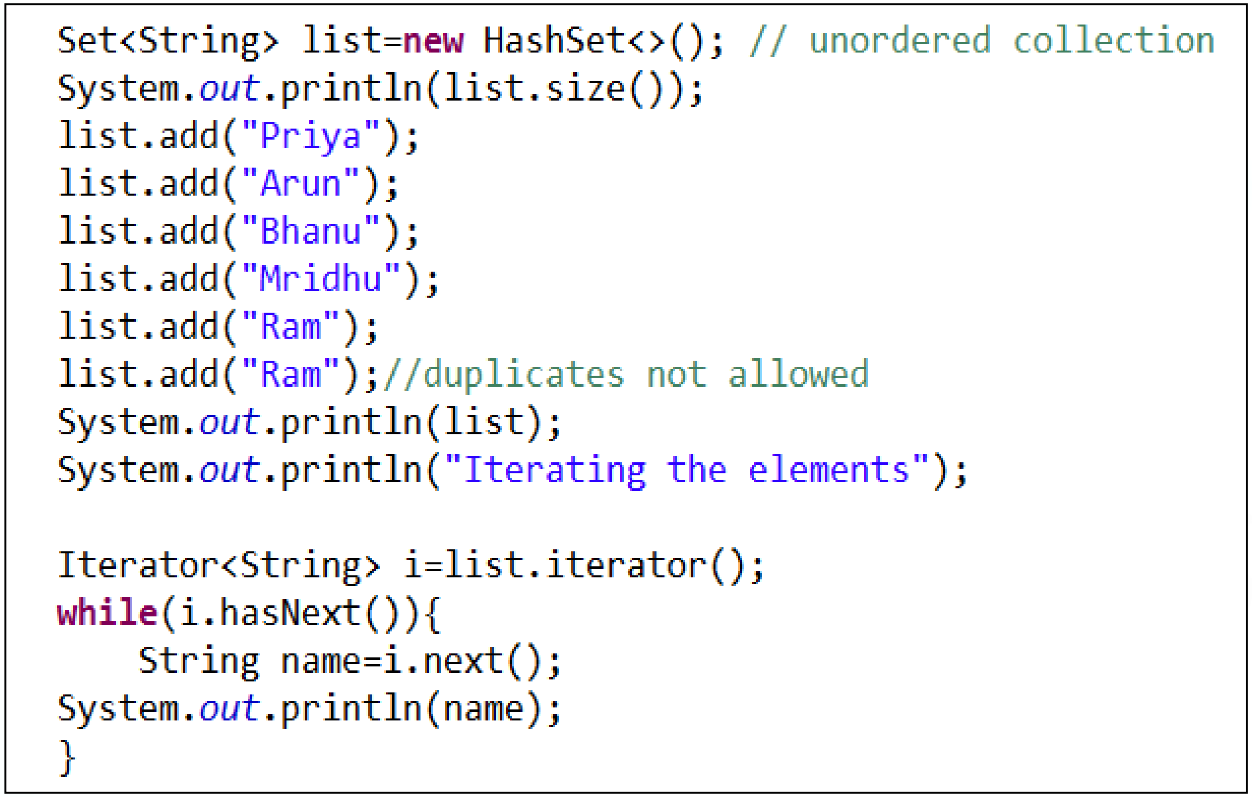
- HashSet
- LinkedHashSet
- TreeSet


Set
<<interface>>
HashSet
LinkedHashSet
SortedSet
<<interface>>
TreeSet

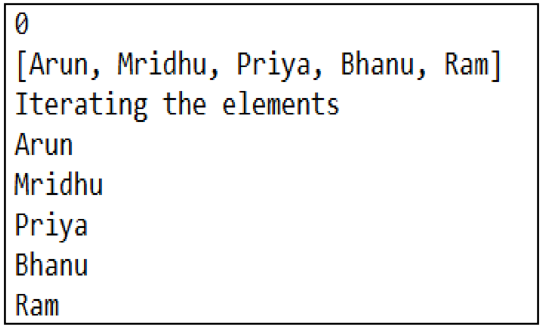
HashSet

- **HashSet** class implements the Set interface, backed by a hash table.
- Makes no guarantees in the iteration order of elements.
- Allows null element.
- HashSet methods are not synchronized
- ***Unordered Collection***

Example



```
Set<String> list=new HashSet<>(); // unordered collection
System.out.println(list.size());
list.add("Priya");
list.add("Arun");
list.add("Bhanu");
list.add("Mridhu");
list.add("Ram");
list.add("Ram");//duplicates not allowed
System.out.println(list);
System.out.println("Iterating the elements");

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

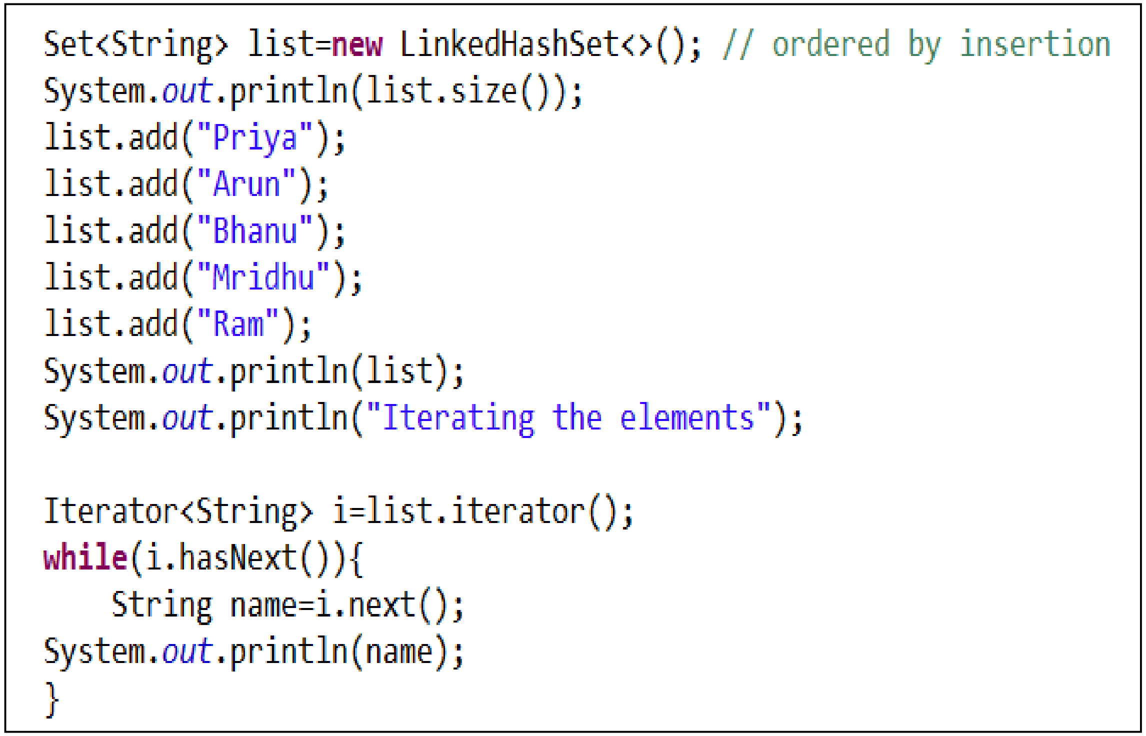


```
0
[Arun, Mridhu, Priya, Bhanu, Ram]
Iterating the elements
Arun
Mridhu
Priya
Bhanu
Ram
```

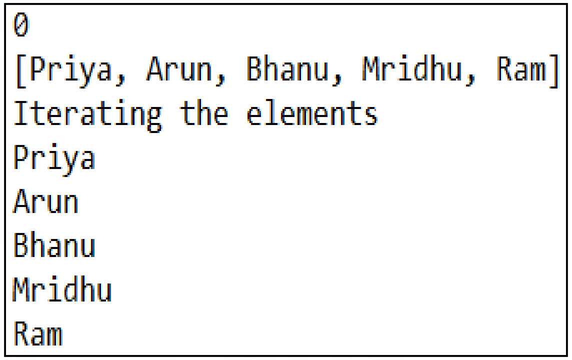
LinkedHashSet

- **LinkedHashSet** class implements the Set interface, backed by a hash table.
- Is implemented as a hash table with a linked list running through it
- Orders its elements based on the order in which they were inserted into the set (insertion-order).
- ***Ordered by insertion***

Example



```
Set<String> list=new LinkedHashSet<>(); // ordered by insertion
System.out.println(list.size());
list.add("Priya");
list.add("Arun");
list.add("Bhanu");
list.add("Mridhu");
list.add("Ram");
System.out.println(list);
System.out.println("Iterating the elements");

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

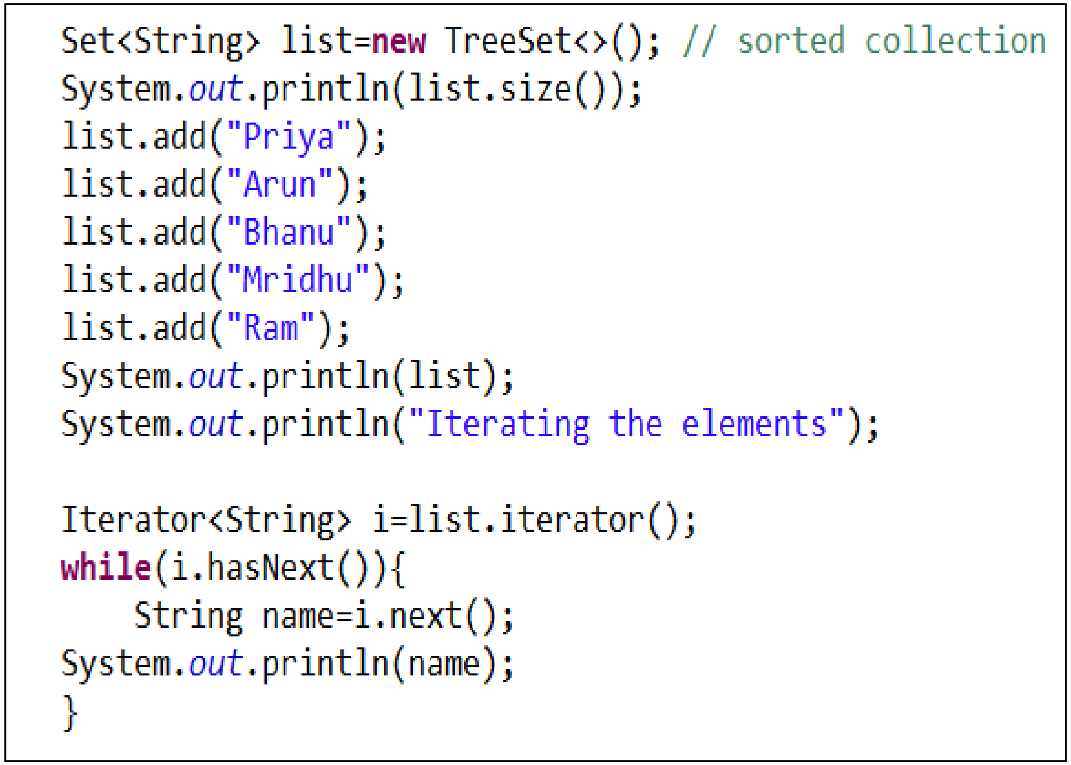


```
0
[Priya, Arun, Bhanu, Mridhu, Ram]
Iterating the elements
Priya
Arun
Bhanu
Mridhu
Ram
```

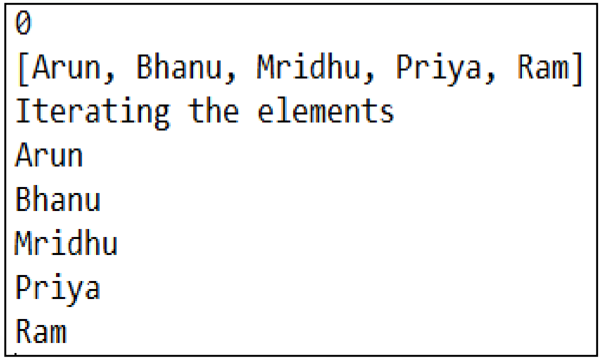
TreeSet

- **TreeSet** class implements the **SortedSet** interface
- Elements are ordered using their natural ordering or by a Comparator provided by Set during creation time
- Slower than HashSet
- Does not allow null elements(throws NullPointerException)
- Does not allow non-compatible elements(throws ClassCastException)
- ***Sorted Collection***

Example



```
Set<String> list=new TreeSet<>(); // sorted collection
System.out.println(list.size());
list.add("Priya");
list.add("Arun");
list.add("Bhanu");
list.add("Mridhu");
list.add("Ram");
System.out.println(list);
System.out.println("Iterating the elements");

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



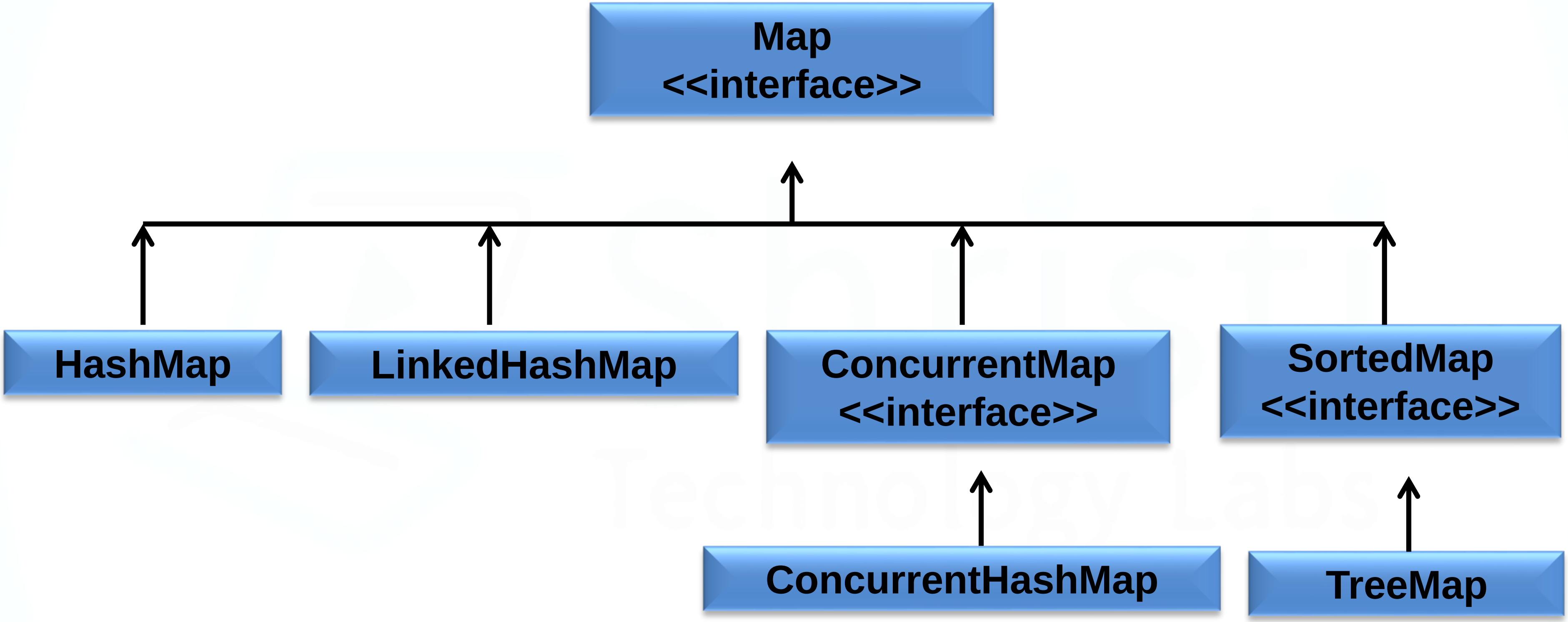
```
0
[Arun, Bhanu, Mridhu, Priya, Ram]
Iterating the elements
Arun
Bhanu
Mridhu
Priya
Ram
```

Map

- Stores an association or mapping between "keys“ & "values"
- A Map is used as an object that maps keys to values.
- Cannot contain duplicate keys
- Each key can map to at most one value.

Classes that implement Map

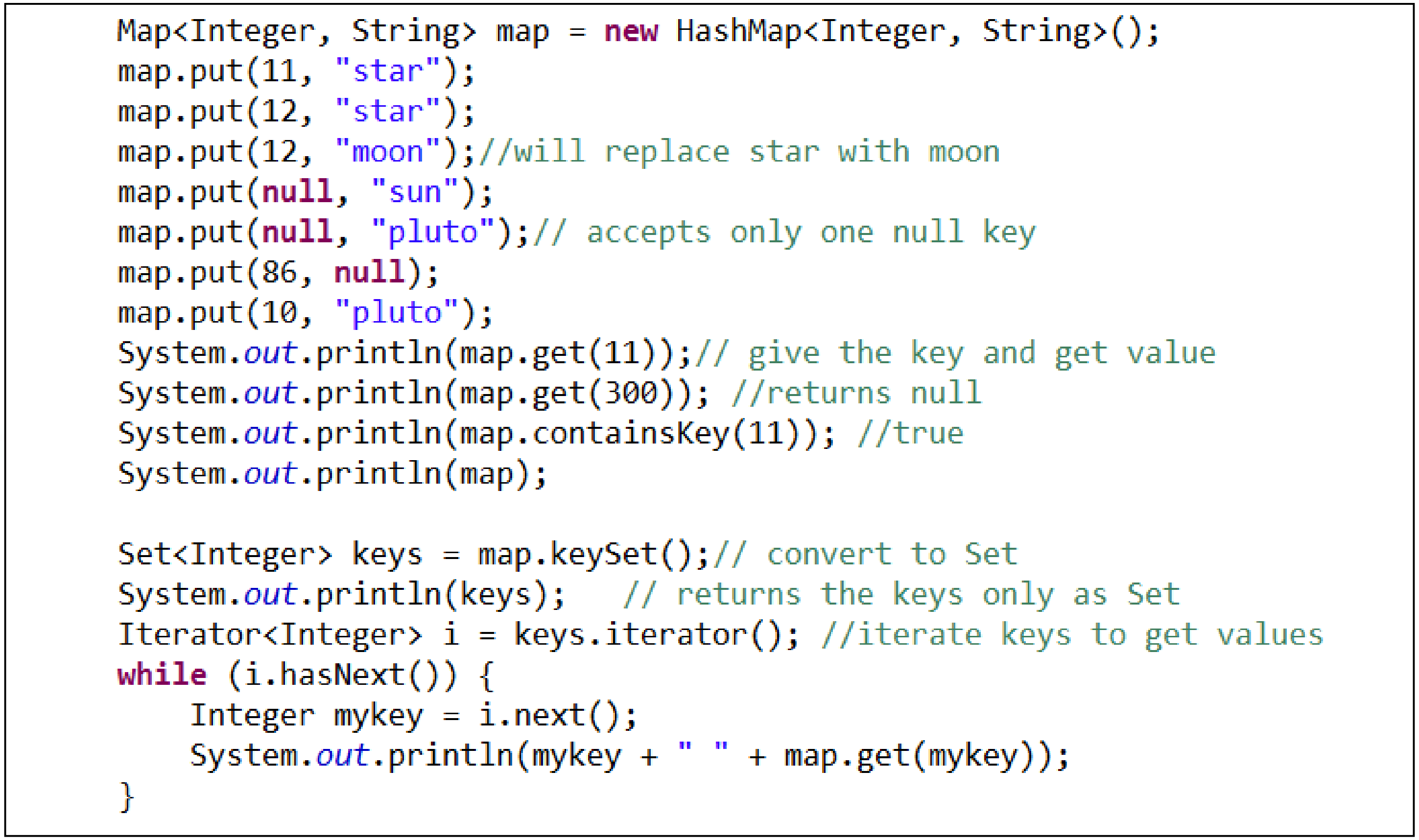
- HashMap
- LinkedHashMap
- TreeMap


Map
<<interface>>
HashMap
LinkedHashMap
ConcurrentMap
<<interface>>
SortedMap
<<interface>>
ConcurrentHashMap
TreeMap

HashMap

- Is a data structure that can map key to values
- Uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the correct value can be found.
- The elements in the map are added as per the hashcode values of the keys.
- Accepts one null key

Example - HashMap



```
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(11, "star");
map.put(12, "star");
map.put(12, "moon");//will replace star with moon
map.put(null, "sun");
map.put(null, "pluto");// accepts only one null key
map.put(86, null);
map.put(10, "pluto");
System.out.println(map.get(11));// give the key and get value
System.out.println(map.get(300)); //returns null
System.out.println(map.containsKey(11)); //true
System.out.println(map);

Set<Integer> keys = map.keySet();// convert to Set
System.out.println(keys);   // returns the keys only as Set
Iterator<Integer> i = keys.iterator(); //iterate keys to get values
while (i.hasNext()) {
    Integer mykey = i.next();
    System.out.println(mykey + " " + map.get(mykey));
}
```

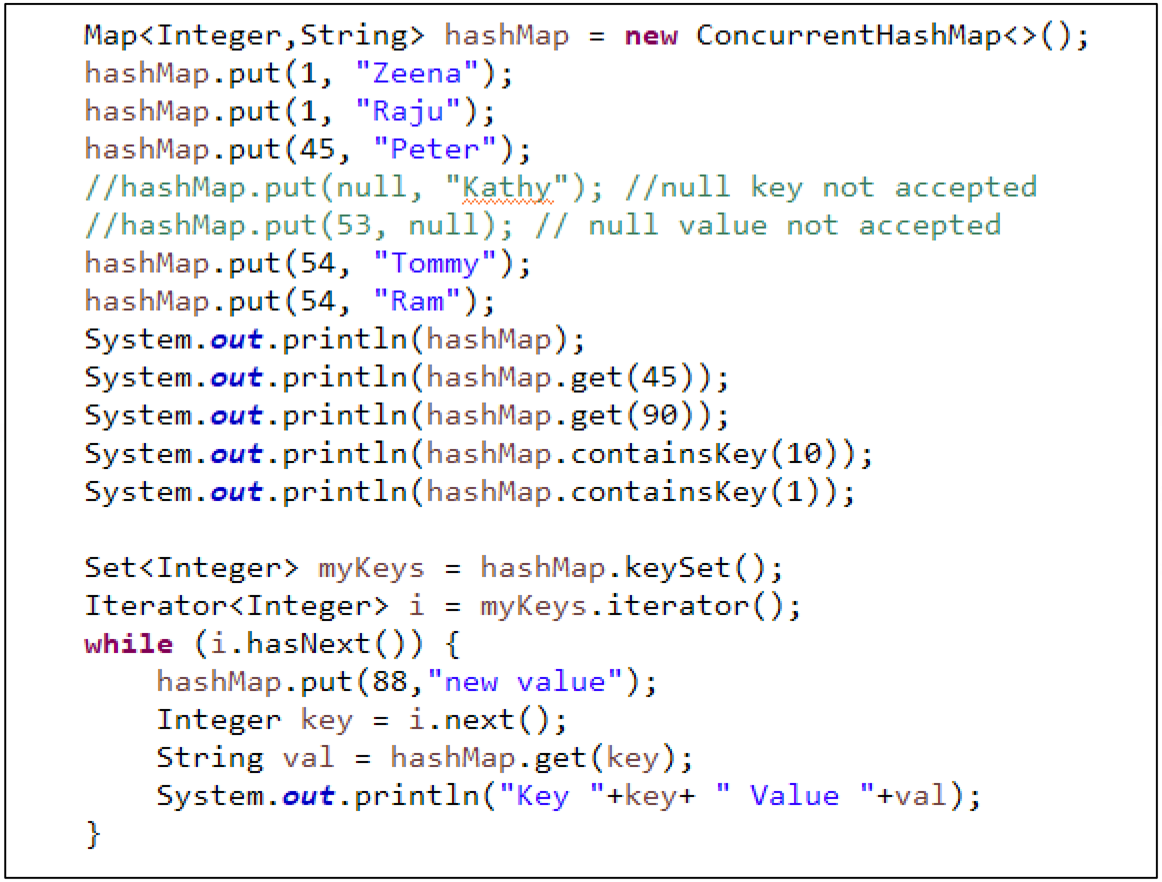
ConcurrentMap

- Can be used safely in concurrent and multithreading applications
- Is capable of handling concurrent access (puts and gets) to it.
- Does not lock the Map while reading from it, locks only a part of it during updates
- Does not lock the entire Map when writing to it, locks only the part of the it that is being written to, internally.

ConcurrentHashMap

- is the implementation of ConcurrentMap
- Does not accept null key and null value
- Allows to modify the map during iteration

Example - ConcurrentHashMap



```
Map<Integer,String> hashMap = new ConcurrentHashMap<>();
hashMap.put(1, "Zeena");
hashMap.put(1, "Raju");
hashMap.put(45, "Peter");
//hashMap.put(null, "Kathy"); //null key not accepted
//hashMap.put(53, null); // null value not accepted
hashMap.put(54, "Tommy");
hashMap.put(54, "Ram");
System.out.println(hashMap);
System.out.println(hashMap.get(45));
System.out.println(hashMap.get(90));
System.out.println(hashMap.containsKey(10));
System.out.println(hashMap.containsKey(1));

Set<Integer> myKeys = hashMap.keySet();
Iterator<Integer> i = myKeys.iterator();
while (i.hasNext()) {
    hashMap.put(88,"new value");
    Integer key = i.next();
    String val = hashMap.get(key);
    System.out.println("Key "+key+ " Value "+val);
}
```

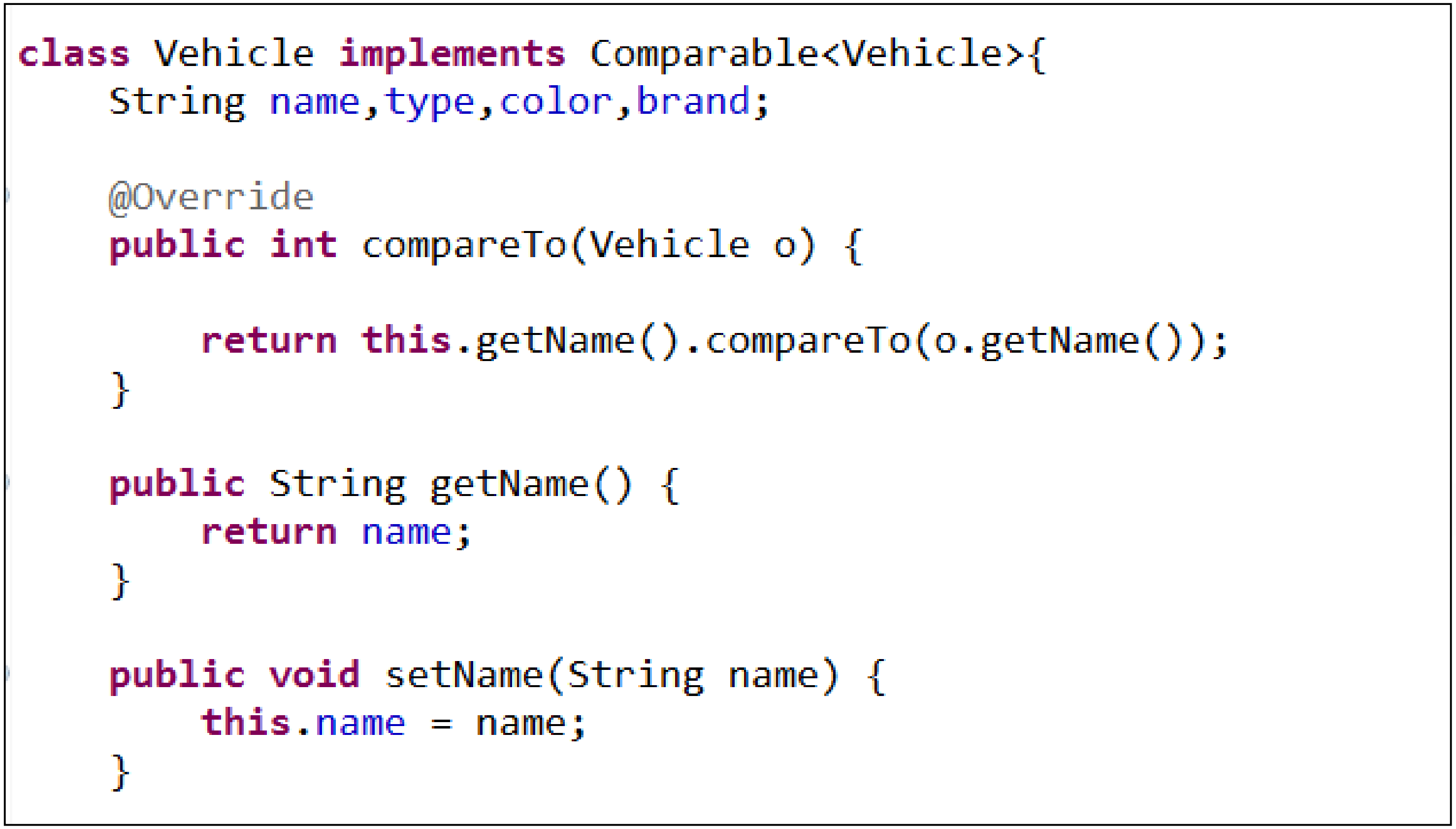
Comparable

- Is an interface from java.lang
- Is implemented by a class in order to be able to compare object of itself with other object of same class.
- Used only when you need one sorting sequence

Rules

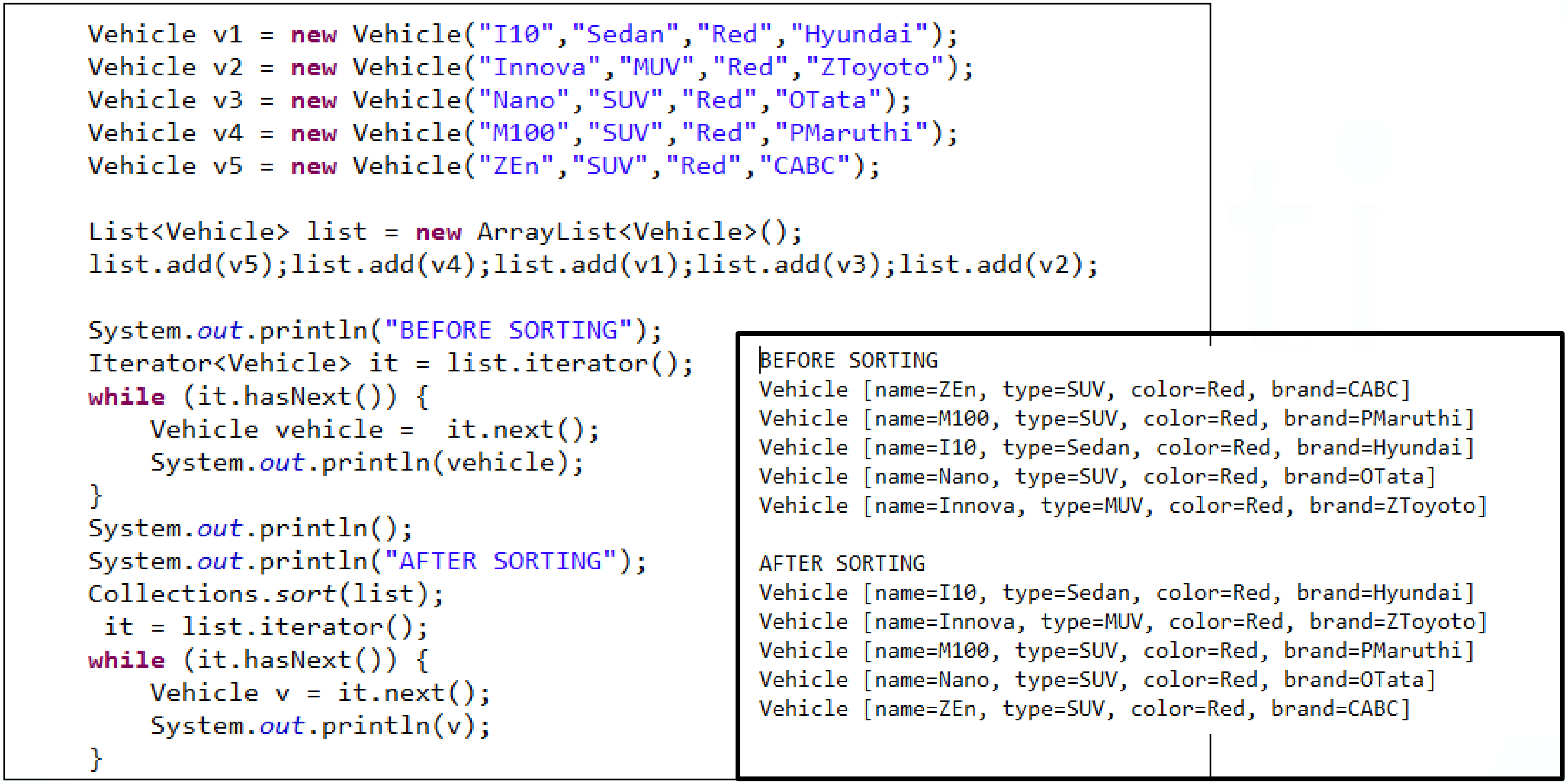
- The class itself must implement the interface
- Implement compareTo(Object o) method.
- Call **Collections.sort(List list)** method to sort the collection

Example – implementing Comparable



```
class Vehicle implements Comparable<Vehicle>{
    String name,type,color,brand;

    @Override
    public int compareTo(Vehicle o) {

        return this.getName().compareTo(o.getName());
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

Example – calling sort() method



```
Vehicle v1 = new Vehicle("I10","Sedan","Red","Hyundai");
Vehicle v2 = new Vehicle("Innova","MUV","Red","ZToyoto");
Vehicle v3 = new Vehicle("Nano","SUV","Red","OTata");
Vehicle v4 = new Vehicle("M100","SUV","Red","PMaruthi");
Vehicle v5 = new Vehicle("ZEn","SUV","Red","CABC");

List<Vehicle> list = new ArrayList<Vehicle>();
list.add(v5);list.add(v4);list.add(v1);list.add(v3);list.add(v2);

System.out.println("BEFORE SORTING");
Iterator<Vehicle> it = list.iterator();
while (it.hasNext()) {
    Vehicle vehicle =  it.next();
    System.out.println(vehicle);
}
System.out.println();
System.out.println("AFTER SORTING");
Collections.sort(list);
 it = list.iterator();
while (it.hasNext()) {
    Vehicle v = it.next();
    System.out.println(v);
}
```

```
BEFORE SORTING
Vehicle [name=ZEn, type=SUV, color=Red, brand=CABC]
Vehicle [name=M100, type=SUV, color=Red, brand=PMaruthi]
Vehicle [name=I10, type=Sedan, color=Red, brand=Hyundai]
Vehicle [name=Nano, type=SUV, color=Red, brand=OTata]
Vehicle [name=Innova, type=MUV, color=Red, brand=ZToyoto]

AFTER SORTING
Vehicle [name=I10, type=Sedan, color=Red, brand=Hyundai]
Vehicle [name=Innova, type=MUV, color=Red, brand=ZToyoto]
Vehicle [name=M100, type=SUV, color=Red, brand=PMaruthi]
Vehicle [name=Nano, type=SUV, color=Red, brand=OTata]
Vehicle [name=ZEn, type=SUV, color=Red, brand=CABC]
```

Comparator

- Is an interface from java.util
- Used to compare objects based on certain attributes/fields.
- Used when multiple sort sequences are needed.

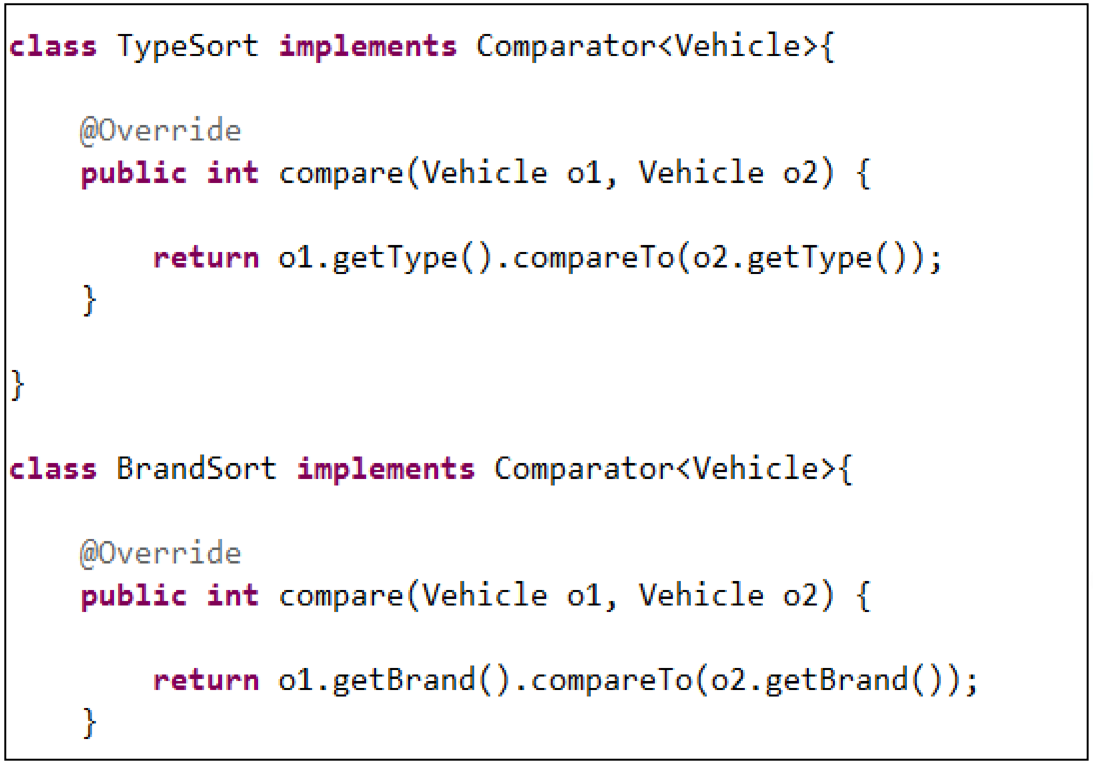
eg. One user might want to sort vehicles by price, one user by name, one by brand – not possible by Comparable as it allows only one sorting

- Create separate classes for different sort sequence.

Rules

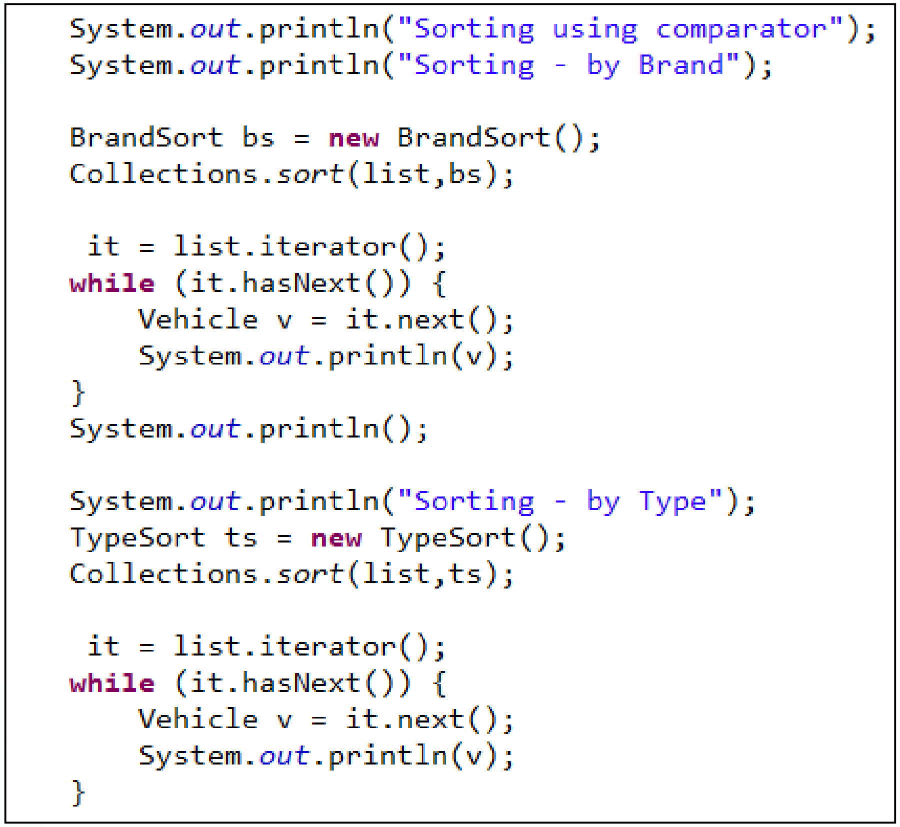
- Create a separate class that implements Comparator
- Implement *compare(Object o,Object o1)*.
- Call **Collections.sort(List obj, comparator obj)** method to sort the collection

Example - implementing Comparator

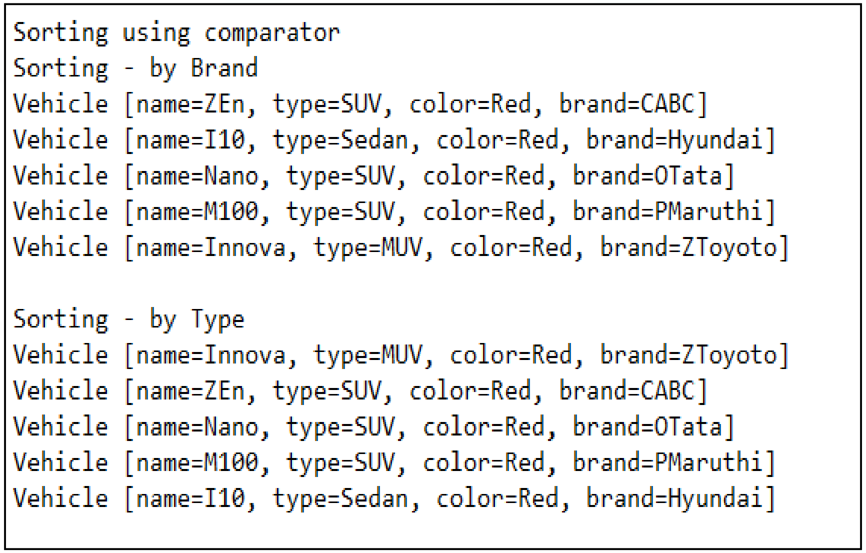


```
class TypeSort implements Comparator<Vehicle>{

    @Override
    public int compare(Vehicle o1, Vehicle o2) {

        return o1.getType().compareTo(o2.getType());
    }

}

class BrandSort implements Comparator<Vehicle>{

    @Override
    public int compare(Vehicle o1, Vehicle o2) {

        return o1.getBrand().compareTo(o2.getBrand());
    }
```

Example - using sort() method

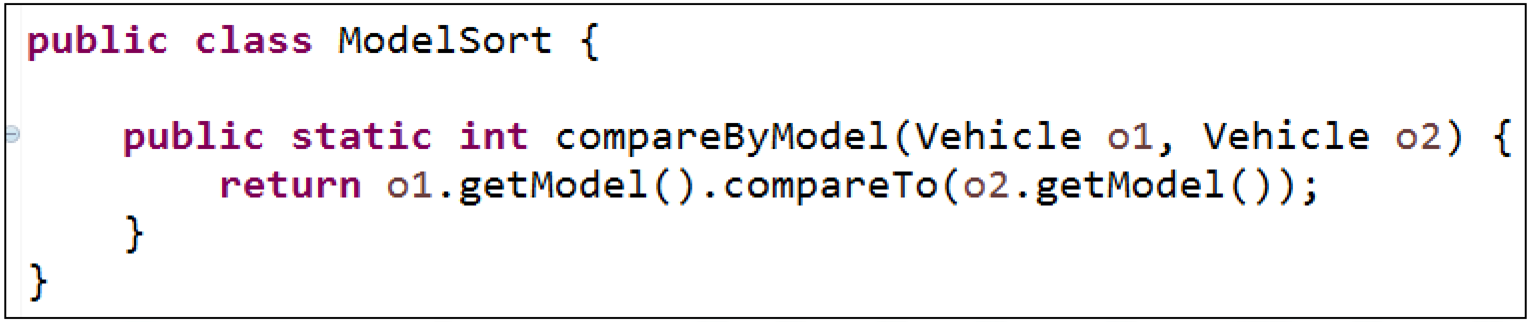


```
System.out.println("Sorting using comparator");
System.out.println("Sorting - by Brand");

BrandSort bs = new BrandSort();
Collections.sort(list,bs);

 it = list.iterator();
while (it.hasNext()) {
    Vehicle v = it.next();
    System.out.println(v);
}
System.out.println();

System.out.println("Sorting - by Type");
TypeSort ts = new TypeSort();
Collections.sort(list,ts);

 it = list.iterator();
while (it.hasNext()) {
    Vehicle v = it.next();
    System.out.println(v);
}
```

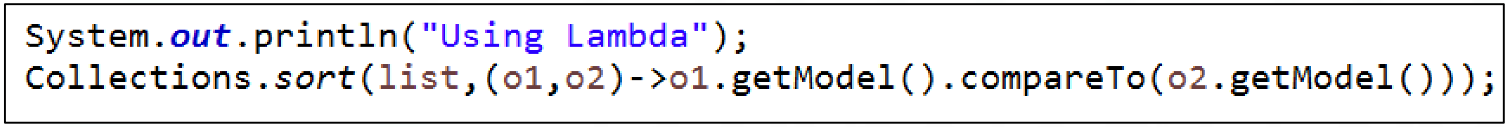


```
Sorting using comparator
Sorting - by Brand
Vehicle [name=ZEn, type=SUV, color=Red, brand=CABC]
Vehicle [name=I10, type=Sedan, color=Red, brand=Hyundai]
Vehicle [name=Nano, type=SUV, color=Red, brand=OTata]
Vehicle [name=M100, type=SUV, color=Red, brand=PMaruthi]
Vehicle [name=Innova, type=MUV, color=Red, brand=ZToyoto]

Sorting - by Type
Vehicle [name=Innova, type=MUV, color=Red, brand=ZToyoto]
Vehicle [name=ZEn, type=SUV, color=Red, brand=CABC]
Vehicle [name=Nano, type=SUV, color=Red, brand=OTata]
Vehicle [name=M100, type=SUV, color=Red, brand=PMaruthi]
Vehicle [name=I10, type=Sedan, color=Red, brand=Hyundai]
```

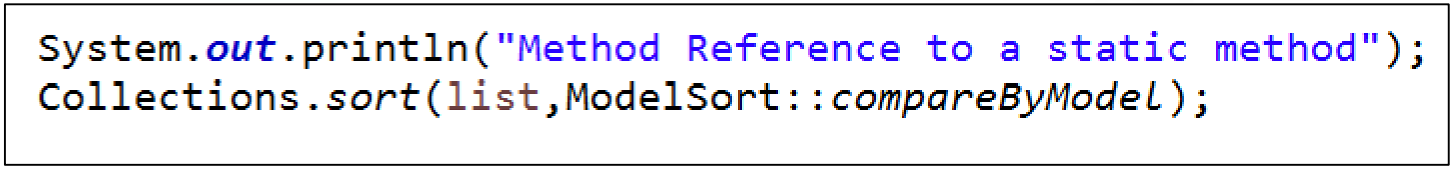
Using Method references - static



```
public class ModelSort {

    public static int compareByModel(Vehicle o1, Vehicle o2) {
        return o1.getModel().compareTo(o2.getModel());
    }
}
```

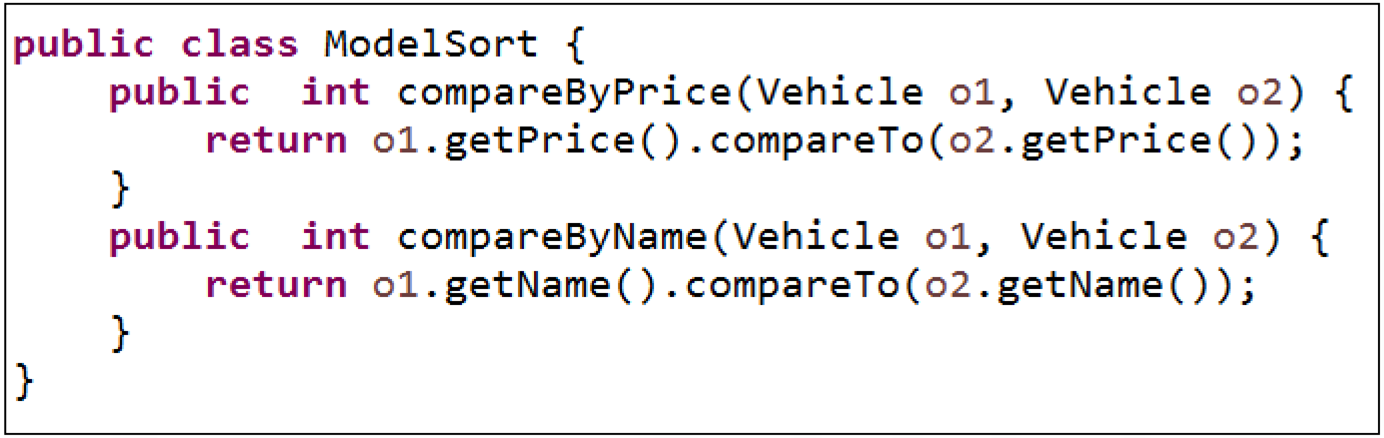


```
System.out.println("Using Lambda");
Collections.sort(list,(o1,o2)->o1.getModel().compareTo(o2.getModel()));
```

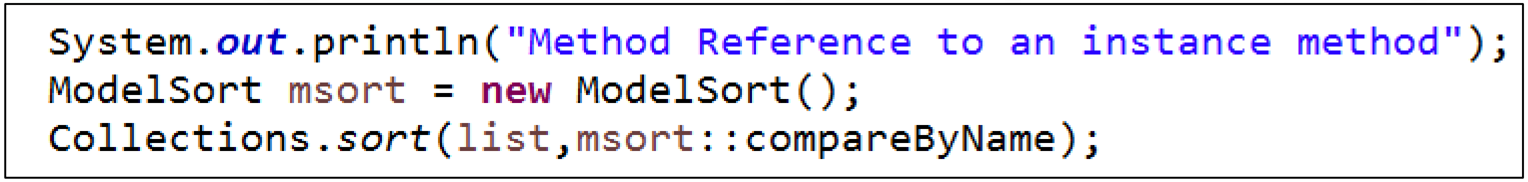


```
System.out.println("Method Reference to a static method");
Collections.sort(list,ModelSort::compareByModel);
```

Using Method references - instance



```
public class ModelSort {
    public  int compareByPrice(Vehicle o1, Vehicle o2) {
        return o1.getPrice().compareTo(o2.getPrice());
    }
    public  int compareByName(Vehicle o1, Vehicle o2) {
        return o1.getName().compareTo(o2.getName());
    }
}
```

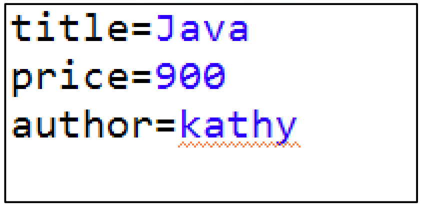


```
System.out.println("Method Reference to an instance method");
ModelSort msort = new ModelSort();
Collections.sort(list,msort::compareByName);
```

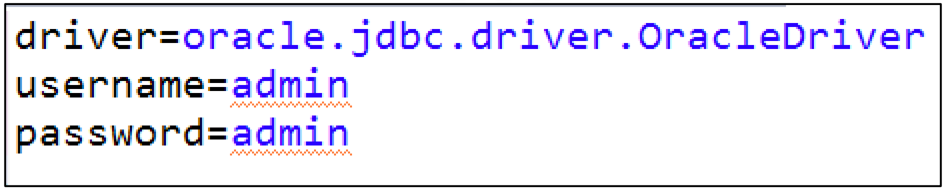
Properties, Random, UUID

Properties

- Is a class which represents a persistent set of properties.
- Is a sub class of Hashtable
- Has methods to get data from properties file and store data into properties file.
- It can be also used to get properties of system.
- The object contains both key and value as string.
- The Properties can be saved to a stream or loaded from a stream



```
title=Java
price=900
author=kathy
```

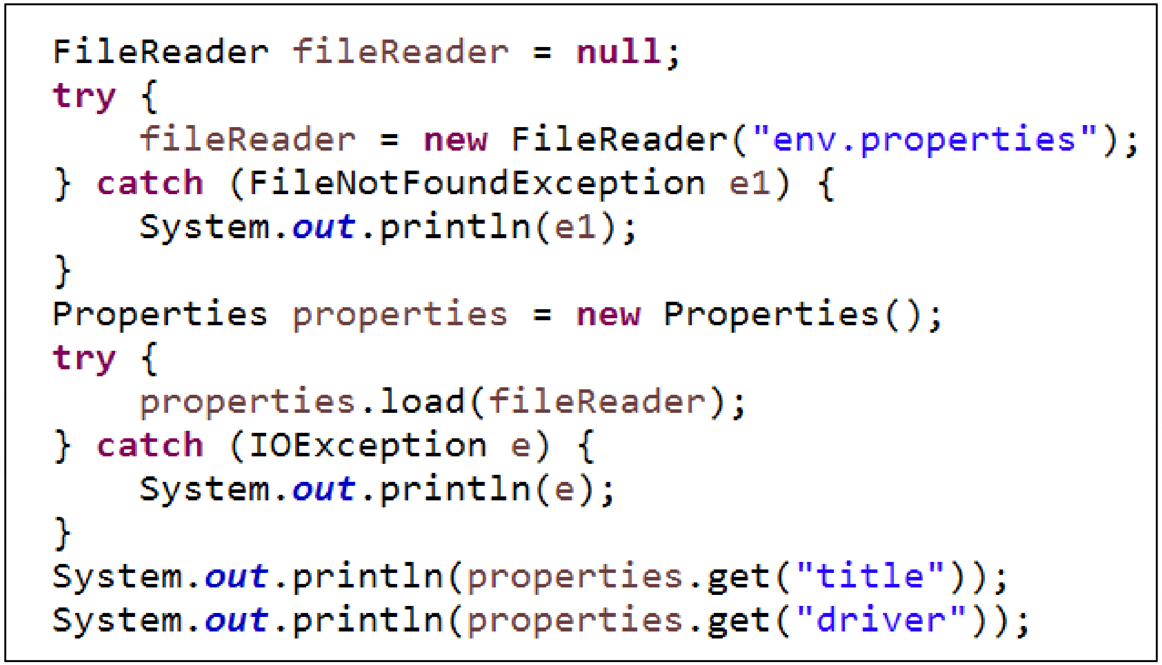


```
driver=oracle.jdbc.driver.OracleDriver
username=admin
password=admin
```

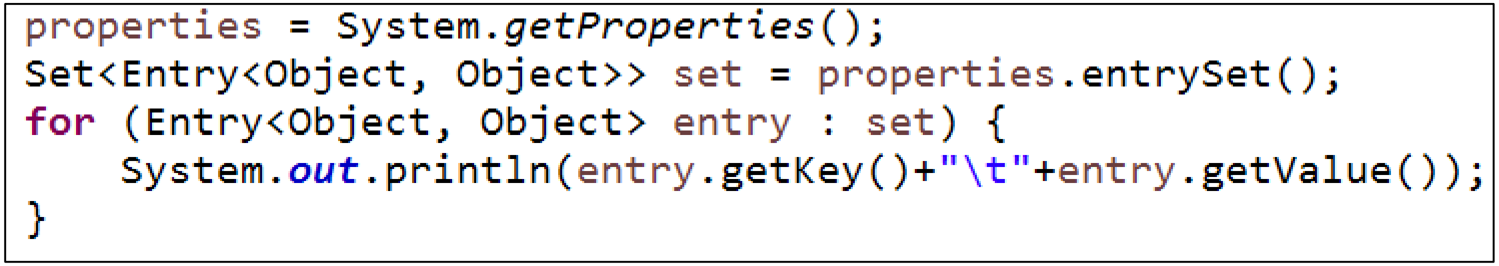
Methods

Method Name	Description
public void load(Reader r)	loads data from the Reader object.
public void load(InputStream is)	loads data from the InputStream object
public String getProperty(String key)	returns value based on the key.
public void setProperty(String key, String value)	sets the property in the properties object.
public void store(Writer w, String comment)	stores the properties in the writer object.
public void store(OutputStream os, String comment)	stores the properties in the OutputStream object.

Example



```
FileReader fileReader = null;
try {
    fileReader = new FileReader("env.properties");
} catch (FileNotFoundException e1) {
    System.out.println(e1);
}
Properties properties = new Properties();
try {
    properties.load(fileReader);
} catch (IOException e) {
    System.out.println(e);
}
System.out.println(properties.get("title"));
System.out.println(properties.get("driver"));
```



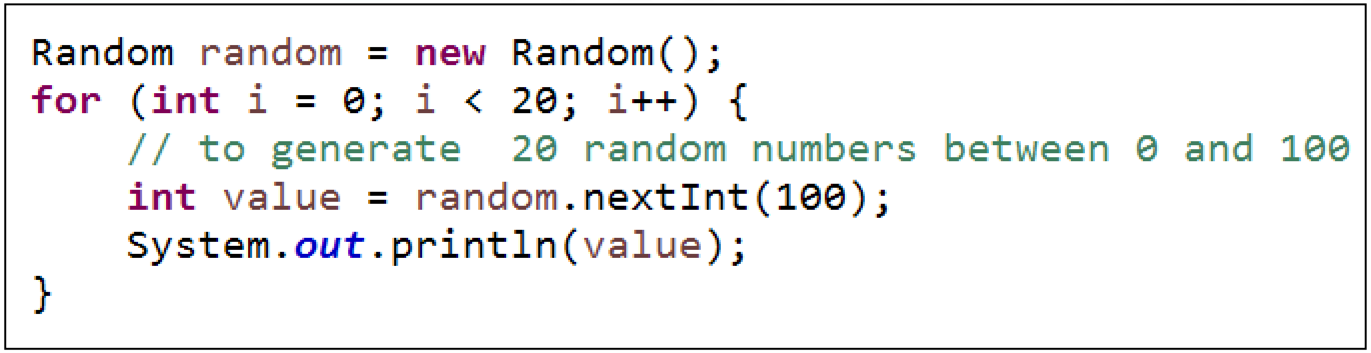
```
properties = System.getProperties();
Set<Entry<Object, Object>> set = properties.entrySet();
for (Entry<Object, Object> entry : set) {
    System.out.println(entry.getKey()+"\t"+entry.getValue());
}
```

Random

- A class that is used to generate a stream of pseudorandom numbers.
- Can also be used to generate random numbers for boolean, float, long.

Random random = new Random(long seed);

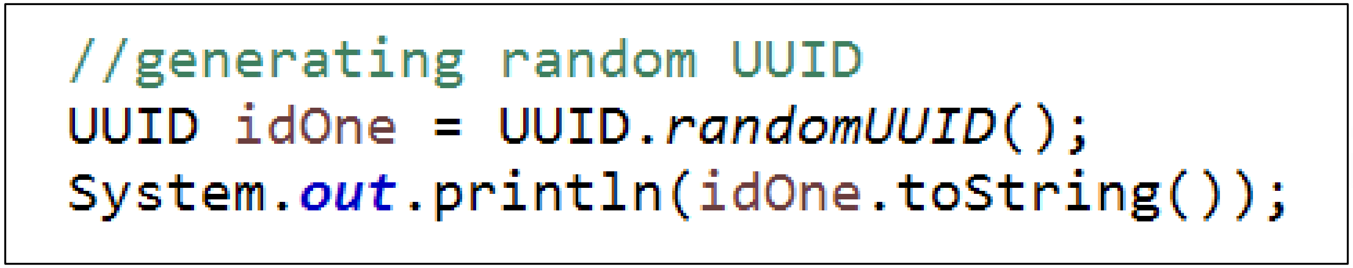
- ***Seed is used get the same numbers on every execution***
- ***Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.***



```
Random random = new Random();
for (int i = 0; i < 20; i++) {
    // to generate  20 random numbers between 0 and 100
    int value = random.nextInt(100);
    System.out.println(value);
}
```

UUID

- Is a class for generating unique Ids
- Is a class that represents an immutable universally unique identifier (UUID).
- Is used for creating sessionId in web application, random file names, transactionId etc.

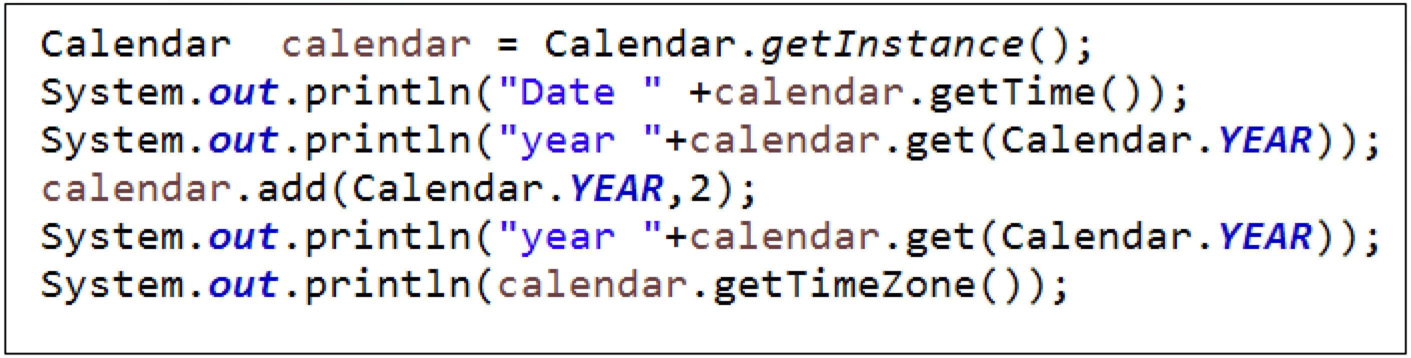


```
//generating random UUID
UUID idOne = UUID.randomUUID();
System.out.println(idOne.toString());
```

Calendar, Locale

Calendar

- Is an abstract class with methods for converting date between a specific instant in time
- Has a set of calendar fields such as MONTH, YEAR, HOUR, etc.
- Is used for manipulating the calendar fields, such as getting the date of the next week.



```
Calendar  calendar = Calendar.getInstance();
System.out.println("Date " +calendar.getTime());
System.out.println("year "+calendar.get(Calendar.YEAR));
calendar.add(Calendar.YEAR,2);
System.out.println("year "+calendar.get(Calendar.YEAR));
System.out.println(calendar.getTimeZone());
```

Locale

- A Locale object represents a specific locale – like geographical, political, or cultural region.
- Locale object is a just an identifier for region

Example:

- To display the number specific to the locale where the number should be formatted according to the conventions of the user's native country or region.

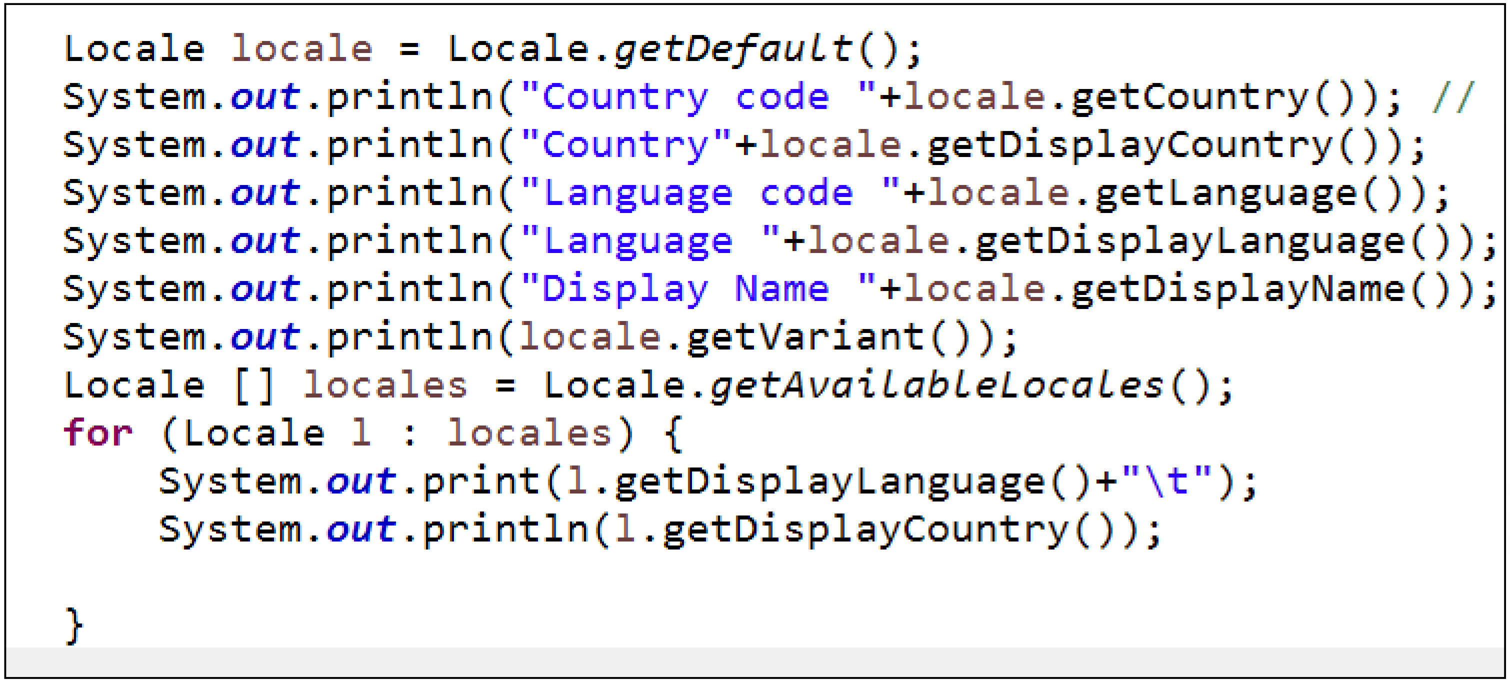
Create Locale Object

Locale(String language(): creates Locale from the specified language.

Locale(String language, String country): creates a Locale object with language and country code.

Locale(String language, String country, String variant): creates a Locale object with language, country code and variant.

Example



```java
Locale locale = Locale.getDefault();
System.out.println("Country code "+locale.getCountry()); //
System.out.println("Country"+locale.getDisplayCountry());
System.out.println("Language code "+locale.getLanguage());
System.out.println("Language "+locale.getDisplayLanguage());
System.out.println("Display Name "+locale.getDisplayName());
System.out.println(locale.getVariant());
Locale [] locales = Locale.getAvailableLocales();
for (Locale l : locales) {
    System.out.print(l.getDisplayLanguage()+"\t");
    System.out.println(l.getDisplayCountry());

}
```

Regular Expressions

Regular Expressions(regex)

- is an API to define pattern for searching or manipulating strings.
- The search pattern can be
  - a simple character,
  - a fixed string
  - a complex expression containing special characters describing the pattern.
- can be used to search, edit and manipulate text/string

Example

- Password, email, phone number validation

Classes in java.util.regex package

Pattern

- is a compiled representation of a regular expression.

Matcher

- the regex engine that interprets the pattern and performs match operations against an input string.

Character classes

Predefined Character classes

- They have a special meaning,
- Provide convenient shorthands for commonly used regular expressions **e.g** Use **.** for any one character search, **\d** for digits [0-9] search

Character Classes

- A set of characters enclosed within square brackets.
- It specifies the characters that will successfully match a single character from a given input string**.**

e.g **[abc]** means a, b, or c (simple class)

Quantifiers

- allows to specify the number of occurrences to match against

e.g **?** for 0 or one, ***** for 0 or more , **+** for 1 or more

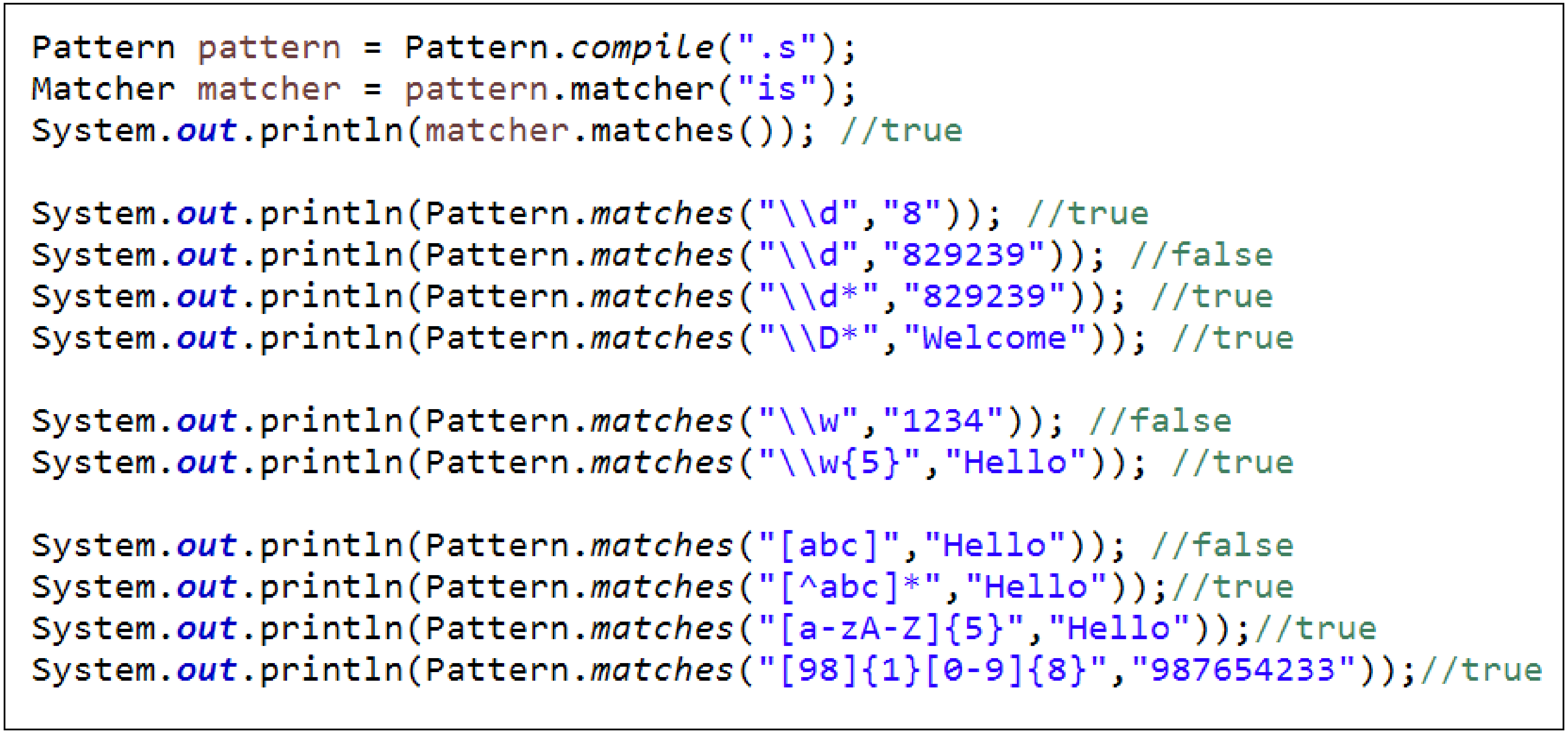
Predefined character classes – meta characters

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	Matches a word boundary where a word character is [a-zA-Z0-9_]
\B	A non word boundary

Character classes

Construct	Description
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z, or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction

Example



```
Pattern pattern = Pattern.compile(".s");
Matcher matcher = pattern.matcher("is");
System.out.println(matcher.matches()); //true

System.out.println(Pattern.matches("\\d","8")); //true
System.out.println(Pattern.matches("\\d","829239")); //false
System.out.println(Pattern.matches("\\d*","829239")); //true
System.out.println(Pattern.matches("\\D*","Welcome")); //true

System.out.println(Pattern.matches("\\w","1234")); //false
System.out.println(Pattern.matches("\\w{5}","Hello")); //true

System.out.println(Pattern.matches("[abc]","Hello")); //false
System.out.println(Pattern.matches("[^abc]*","Hello"));//true
System.out.println(Pattern.matches("[a-zA-Z]{5}","Hello"));//true
System.out.println(Pattern.matches("[98]{1}[0-9]{8}","987654233"));//true
```

Summary

- Collection Framework
- Properties, Random, UUID
- Calendar
- Locale
- Regular Expressions
- Example for functional Interfaces

Thank you