Core Java

Day 11 Agenda

- Q & A
- Generic Programming
  - Generic Methods
  - Generic Limitations
  - Generics Limitations
  - Generics Interfaces
- Comparable vs Comparator interfaces
- Java collection framework
  - Concepts
  - Hierarchy
  - Collection iterface
  - Iterator

Generic Programming

Generic Methods

- Generic methods are used to implement generic algorithms.
- Example:

```
// non type-safe
static void printArray(Object[] arr) {
    for(Object ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
// type-safe
static <T> void printArray(T[] arr) {
    for(T ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
String[] arr1 = { "John", "Dagny", "Alex" };
printArray(arr1); // printArray<String> -- String type is inferred

Integer[] arr2 = { 10, 20, 30 };
printArray(arr2); // printArray<Integer> -- Integer type is inferred
```

Generics Limitations

1. Cannot instantiate generic types with primitive Types. Only reference types are allowed.

```
ArrayList<Integer> list = new ArrayList<Integer>(); // okay
ArrayList<int> list = new ArrayList<int>(); // compiler error
```

2. Cannot create instances of Type parameters.

```
Integer i = new Integer(11); // okay
T obj = new T(); // error
```

3. Cannot declare static fields with generic type parameters.

```
class Box<T> {
    private T obj; // okay
    private static T object; // compiler error
    // ...
}
```

4. Cannot Use casts or instanceof with generic Type params.

```
if(obj instanceof T) {  // compiler error
    newobj = (T)obj;    // compiler error
}
```

5. Cannot Create arrays of generic parameterized Types

```
T[] arr = new T[5]; // compiler error
```

6. Cannot create, catch, or throw Objects of Parameterized Types

```
throw new T(); // compiler error

try {
    // ...
} catch(T ex) { // compiler error
    // ...
}
```

7. Cannot overload a method just by changing generic type. Because after erasing/removing the type param, if params of two methods are same, then it is not allowed.

```
public void printBox(Box<Integer> b) {
    // ...
}
public void printBox(Box<String> b) { // compiler error
    // ...
}
```

Type erasure

- The generic type information is erased (not maintained) at runtime (in JVM). `Box<Integer>` and `Box<Double>` both are internally (JVM level) treated as Box objects. The field "T obj" in Box class, is treated as "Object obj".
- Because of this method overloading with genric type difference is not allowed.

```
void printBox(Box<Integer> b) { ... }
    // void printBox(Box b) { ... } <-- In JVM
void printBox(Box<Double> b) { ... } //compiler error
    // void printBox(Box b) { ... } <-- In JVM
```

Generic Interfaces

- Interface is standard/specification.

```
// Comparable is pre-defined interface -- non-generic till Java 1.4
interface Comparable {
    int compareTo(Object obj);
}
class Person implements Comparable {
    // ...
    public int compareTo(Object obj) {
        Person other = (Person)obj; // down-casting
```

```
        // compare "this" with "other" and return difference
            // if both are same, then return 0.
            // if this is greater than other, then return +ve.
            // if this is smaller than other, then return -ve.
    }
}
class Program {
    public static void main(String[] args) {
        Person p1 = new Person("James Bond", 50);
        Person p2 = new Person("Ironman", 45);
        int diff = p1.compareTo(p2);
        if(diff == 0)
            System.out.println("Both are same");
        else if(diff > 0)
            System.out.println("p1 is greater than p2");
        else //if(diff < 0)
            System.out.println("p1 is less than p2");

        diff = p2.compareTo("Superman"); // will fail at runtime with ClassCastException (in down-casting)
    }
}
```

- Generic interface has type-safe methods (arguments and/or return-type).

```
// Comparable is pre-defined interface -- generic since Java 5.0
interface Comparable<T> {
    int compareTo(T obj);
}
class Person implements Comparable<Person> {
    // ...
    public int compareTo(Person other) {
        // compare "this" with "other" and return difference
    }
}
```

```
class Program {
    public static void main(String[] args) {
        Person p1 = new Person("James Bond", 50);
        Person p2 = new Person("Ironman", 45);
        int diff = p1.compareTo(p2);
        if(diff == 0)
            System.out.println("Both are same");
        else if(diff > 0)
            System.out.println("p1 is greater than p2");
        else //if(diff < 0)
            System.out.println("p1 is less than p2");

        diff = p2.compareTo("Superman"); // compiler error
    }
}
```

Comparable<>

- Standard for comparing the current object to the other object.
- Also referred as "Natural Ordering" for the class.
- Has single abstract method `int compareTo(T other);`
- In java.lang package.
- Used by various methods like Arrays.sort(Object[]), ...

```
// pre-defined interface
interface Comparable<T> {
    int compareTo(T other);
}
```

```
class Employee implements Comparable<Employee> {
    private int empno;
```

```
    private String name;
    private int salary;
    // ...
    public int compareTo(Employee other) {
        int diff = this.empno - other.empno;
        return diff;
    }
}
```

```
Employee e1 = new Employee(1, "Sarang", 50000);
Employee e2 = new Employee(2, "Nitin", 40000);
int diff = e1.compareTo(e2);
```

```
Employee[] arr = { ... };
Arrays.sort(arr);
for(Employee e:arr)
    System.out.println(e);
```

- Conventionally, Comparable.compareTo() implementation should be consistent with equals() method i.e. comparison should be done on the same fields on which equality is tested.

Comparator<>

- Standard for comparing two (other) objects.
- Has single abstract method `int compare(T obj1, T obj2);`
- In java.util package.
- Used by various methods like Arrays.sort(T[], comparator), ...

```
// pre-defined interface
interface Comparator<T> {
    int compare(T obj1, T obj2);
}
```

```
class EmployeeSalaryComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        if(e1.getSalary() == e2.getSalary())
            return 0;
        if(e1.getSalary() > e2.getSalary())
            return +1;
        return -1;
    }
}
```

Multi-level sorting

```
class Employee implements Comparable<Employee> {
    private int empno;
    private String name;
    private String designation;
    private int department;
    private int salary;
    // ...
}
```

```

```

```
// Multi-level sorting -- 1st level: department, 2nd level: designation, 3rd level: salary(int)
class CustomComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        int diff = e1.getDepartment().compareTo(e2.getDepartment());
        if(diff == 0)
            diff = e1.getDesignation().compareTo(e2.getDesignation());
        if(diff == 0)
            diff = e1.getSalary() - e2.getSalary();
        return diff;
    }
}
```

```
Employee[] arr = { ... };
Arrays.sort(arr, new CustomComparator());
// ...
```

Java Collection Framework

- Collection framework is Library of reusable data structure classes that is used to develop application.
- Main purpose of collection framework is to manage data/objects in RAM efficiently.
- Collection framework was introduced in Java 1.2 and type-safe implementation is provided in 5.0 (using generics).
- java.util package.
- Java collection framework provides
    - Interfaces -- defines standard methods for the collections.
    - Implementations -- classes that implements various data stuctures.
    - Algorithms -- helper methods like searching, sorting, ...

Collection Hierarchy

- Interfaces: Iterable, Collection, List, Queue, Set, Map, Deque, SortedSet, SortedMap, ...

- Implementations: ArrayList, LinkedList, HashSet, HashMap, ...
- Algorithms: sort(), reverse(), max(), min(), ... -> in Collections class static methods

Iterable interface

- To traverse any collection it provides an Iterator.
- Enable use of for-each loop.
- In java.lang package
- Methods
    - Iterator iterator() // SAM
    - default Spliterator spliterator()
    - default void forEach(Consumer<? super T> action)

Collection interface

- Root interface in collection framework interface hierarchy.
- Most of collection classes are inherited from this interface (indirectly).
- Provides most basic/general functionality for any collection
- Abstract methods
    - boolean add(E e)
    - int size()
    - boolean isEmpty()
    - void clear()
    - boolean contains(Object o)
    - boolean remove(Object o)
    - boolean addAll(Collection<? extends E> c)
    - boolean containsAll(Collection<?> c)
    - boolean removeAll(Collection<?> c)
    - boolean retainAll(Collection<?> c)
    - Object[] toArray()
    - Iterator iterator() -- inherited from Iterable
- Default methods

    - default Stream stream()
    - default Stream parallelStream()
    - default boolean removeIf(Predicate<? super E> filter)

Traversal

- Using Iterator

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

- Using for-each loop

```
for(Integer i:list)
    System.out.println(i);
```

    - Gets converted into Iterator traversal

```
for(Iterator<Integer> itr = list.iterator(); itr.hasNext();)  {
    Integer i = itr.next();
    System.out.println(i);
}
```

- Enumeration -- Traversing Vector (Java 1.0)

```
// v is Vector<Integer>
Enumeration<Integer> e = v.elements();
while(e.hasMoreElements()) {
    Integer i = e.nextElement();
    System.out.println(i);
}
```

Iterator

- Part of collection framework (1.2)
- Methods
  - boolean hasNext()
  - E next()
  - void remove()
- Example

```
Iterator<E> e = v.iterator();
while(e.hasNext()) {
    E ele = e.next();
    System.out.println(ele);
}
```

Assignment

1. Write a generic static method to find minimum from an array of "Number".

```
static <T extends Number> T findMin(T[] arr) {
    // your logic
}
static void main(String[] args) {
```

```
    Integer[] arr1 = { 22, 55, 77, 11, 33 };
    Integer min1 = findMin(arr1);
    Double[] arr2 = { 2.2, 5.5, 7.7, 1.1, 3.3 };
    Double min2 = findMin(arr2);
}
```

2. A generic sort method for implementing selection sort algorithm is given below. In main(), create array of Double and sort it by calling selectionSort().

```
static <T> void selectionSort(T[] arr, Comparator<T> c) {
    for(int i=0; i<arr.length-1; i++) {
        for(int j=i+1; j<arr.length; j++) {
            if(c.compare(arr[i], arr[j]) > 0) {
               T temp = arr[i];
               arr[i] = arr[j];
               arr[j] = temp;
            }
        }
    }
}
```

3. Use Arrays.sort() to sort array of Students using Comparator. The 1st level sorting should be on city (desc), 2nd level sorting should be on marks (desc), 3rd level sorting should be on name (asc).

```
class Student {
    private int roll;
    private String name;
    private String city;
    private double marks;
    // ...
}
```

4. Store book details in a library in a list -- ArrayList.
    - Book details: isbn(string), price(double), authorName(string), quantity(int)
    - Write a menu driven (do-while + switch-case) program to
        1. Add new book in list.
        2. Display all books in forward order.
        3. Delete at book given index -- list.remove(index);
        4. Check if book with given isbn is in list or not
        5. Delete all books in list
        6. Display number of books in list
        7. Sort all books by price in desc order -- list.sort();