Core Java

Day 12 - Agenda

- Generic Programming
    - Limitations
    - Generic Methods
    - Generic Interfaces
- Comparable
- Comparator
- Collection Framework
    - Collection

Generic Programming

Generic Methods

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

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

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

```java
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.

```java
    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.

```java
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.

```java
// 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
    }
}
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).

```java
// 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[]), ...

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

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

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

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

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

```java
// 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;
    }
}
```

```java
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)