Core Java

Agenda

- Q & A
- Generic Programming
    - Introduction
    - Generic Classes
    - Advantages of Generics
    - Bounded & Unbounded generic types
    - Upper & Lower bounded generic types
    - Generic Methods
    - Generics Limitations
- Comparable vs Comparator interfaces

Singleton class

- A single object of the class is accessible throughout the application.
- This is similar to global variables/objects in C.
- Example:
    - Error/Exception collection that needs to be available throughout the application.
    - Scanner object can be accessed throughout of the application.

```
// Input is a Singleton class.
public class Input {
    // fields
    private Scanner scanner;
    // private constructor
    private Input() {
```

```
            // initialize the fields
            scanner = new Scanner(System.in);
        }
        // methods
        public Scanner getScanner() {
            return scanner;
        }
        // static reference to hold the object address
        private static Input in;
        static {
            // create object only once
            in = new Input();
        }
        // return address of the same object
        public static Input getInstance() {
            return in;
        }
}
```

```
Input input = Input.getInstance();
Scanner sc = input.getScanner();
// OR
Scanner sc = Input.getInstance().getScanner();
```

Generic Programming

- Code is said to be generic if same code can be used for various (practically all) types.
- Best example:
  - Data structure e.g. Stack, Queue, Linked List, ...
  - Algorithms e.g. Sorting, Searching, ...

Introduction

- Two ways to do Generic Programming in Java
  - using java.lang.Object class
  - using Generics

Using java.lang.Object

```Java
class Box {
    private Object obj;
    public void set(Object obj) {
        this.obj = obj;
    }
    public Object get() {
        return this.obj;
    }
}
```

```Java
Box b1 = new Box();
b1.set("Nilesh");
String obj1 = (String)b1.get();
System.out.println("obj1 : " + obj1);

Box b2 = new Box();
b2.set(new Date());
Date obj2 = (Date)b2.get();
System.out.println("obj2 : " + obj2);

Box b3 = new Box();
b3.set(new Integer(11));
String obj3 = (String)b3.get();  // ClassCastException
System.out.println("obj3 : " + obj3);
```

Using Generics

- Similar to templates in C++.
- We can implement generic classes, interfaces, and methods (as per requirement).
- Added in Java 5.0.

Generic Classes

- Implementing a generic class

```
class Box<TYPE> {
    private TYPE obj;
    public void set(TYPE obj) {
        this.obj = obj;
    }
    public TYPE get() {
        return this.obj;
    }
}
```

```
Box<String> b1 = new Box<String>();
b1.set("Nilesh");
String obj1 = b1.get();
System.out.println("obj1 : " + obj1);

Box<Date> b2 = new Box<Date>();
b2.set(new Date());
Date obj2 = b2.get();
System.out.println("obj2 : " + obj2);

Box<Integer> b3 = new Box<Integer>();
b3.set(new Integer(11));
```

```
String obj3 = b3.get();  // Compiler Error
System.out.println("obj3 : " + obj3);
```

- Instantiating generic class

```
Box<String> b1 = new Box<String>(); // Okay

Box<String> b2 = new Box<>(); // Okay -- type inference

Box<> b3 = new Box<>(); // Error -- type must be given while creating reference

Box<Object> b4 = new Box<String>(); // Error

Box b5 = new Box(); // Acceptable (not standard practice) -- internally considered Object type -- compiler
warning "raw types"

Box<Object> b6 = new Box<Object>(); // Okay -- Not usually required
```

Advantages of Generics

- Stronger type checking at compile time i.e. type-safe coding.
- Explicit type casting is not required.
- Generic data structure and algorithm implementation.

Generic types naming convention

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value

6. S,U,R : Additional type param

Bounded generic types

- Bounded generic param restricts data type that can be used as type argument.
- Decided by the developer of the generic class.

```
class Box<T extends Number> {
    private T obj;
    public T get() {
        return this.obj;
    }
    public void set(T obj) {
        this.obj = obj;
    }
}
```

- The Box<> can now be used only for the classes inherited from the Number class.

```
1. Box<Number> b1 = new Box<>(); // okay
2. Box<Boolean> b2 = new Box<>(); // error
3. Box<Character> b3 = new Box<>(); // error
4. Box<String> b4 = new Box<>(); // error
5. Box<Integer> b5 = new Box<>(); // okay
6. Box<Double> b6 = new Box<>(); // okay
7. Box<Date> b7 = new Box<>(); // error
8. Box<Object> b8 = new Box<>(); // error
```

Unbounded generic types

- Unbounded generic type is indicated with wild-card "?".

- Can be given while declaring generic class reference.

```
class Box<T> {
    private T obj;
    public Box(T obj) {
        this.obj = obj;
    }
    public T get() {
        return this.obj;
    }
    public void set(T obj) {
        this.obj = obj;
    }
}
```

```
public static void printBox(Box<?> b) {
    Object obj = b.get();
    System.out.println("Box contains: " + obj);
}
```

```
Box<String> sb = new Box<String>("DAC");
printBox(sb); // ?
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // ?
Box<Date> db = new Box<Date>(new Date());
printBox(db); // ?
Box<Float> fb = new Box<Float>(200.5);
printBox(fb); // ?
```

Upper bounded generic types

- Generic param type can be the given class or its sub-class.

```
public static void printBox(Box<? extends Number> b) {
    Object obj = b.get();
    System.out.println("Box contains: " + obj);
}
```

```
Box<String> sb = new Box<String>("DAC");
printBox(sb); // ?
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // ?
Box<Date> db = new Box<Date>(new Date());
printBox(db); // ?
Box<Float> fb = new Box<Float>(200.5);
printBox(fb); // ?
```

Lower bounded generic types

- Generic param type can be the given class or its super-class.

```
public static void printBox(Box<? super Integer> b) {
    Object obj = b.get();
    System.out.println("Box contains: " + obj);
}
```

```

```

```
Box<String> sb = new Box<String>("DAC");
printBox(sb); // ?
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // ?
Box<Date> db = new Box<Date>(new Date());
printBox(db); // ?
Box<Float> fb = new Box<Float>(200.5f);
printBox(fb); // ?
```

Generic Methods

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

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

```
// type-safe
<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) 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: Comparable<> vs Comparator<>

Comparable<>

- Standard for comparing the current object to the other object.
- 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);
```

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: name
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());
// ...
```

Assignment

1. Copy Shape interface and inherited classes (Circle, Rectangle, and Square) from previous assignment/classwork. Implement generic class Box so that it can store any Shape in it.
2. Write a generic static method to find minimum from an array of Number.
3. Write a generic sort method for implementing selection sort algorithm with given comparator. Refer code below.

```
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;
            }
        }
    }
}
```

4. 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;
    // ...
}
```