# Core Java

## Agenda

- Enum
- Exception chaining
- Generics
- Comparable and Comparator

## Q & A

1. Why Strings in Java are immutable?
    - Java has String pool to reduce memory usage. String doesn't store same string multiple times i.e. only one copy of the one string kept. This can be implemented only if strings are immutable.
    - In java type/class names, method name, field names are internally stored as strings (utf-8). Any change done in these strings at runtime will cause unexpected results. This can be a security threat.
    - To avoid these problems, Java strings are immutable.
2. Does StringBuilder use String pool?
    - No. Only string literal will be stored in pool, but StringBuilder object will be created in heap.
    - StringBuilder sb = new StringBuilder("Sunbeam");
3. 
```
StringBuilder sb1 = new StringBuilder("XYZ");
StringBuilder sb2 = sb1.reverse();
System.out.println(sb1 == sb2); // true
```

```
class Employee extends Person {
    // ...
    /*
    public String toString() {
```

```
        StringBuilder sb = new StringBuilder("Employee - ");
        sb.append("Employee - ");
        sb.append(super.toString());
        sb.append("Id = ");
        sb.append(this.id);
        sb.append("Salary = ");
        sb.append(this.salary);
        return sb.toString();
    }
    */
    public String toString() {
        return new StringBuilder("Employee - ")
            .append(super.toString())
            .append("Id = ")
            .append(this.id)
            .append("Salary = ")
            .append(this.salary)
            .toString();
    }
}
```

4. String concat vs StringBuilder.
    - String concatenation internally creates new string objects (for each concat). This is unefficient (time and space).
    - StringBuilder is efficient option for building strings with multiple concat (append).

```
String str1 = "Sun";
String str2 = "Beam"
String str3 = "DAC";
String s1 = str1 + str2 + str3;
    // s1 = str1.concat(str2).concat(str3);
StringBuilder sb = new StringBuilder();
String s2 = sb.append(str1).append(str2).append(str3).toString();
// ...
```

- Why StringBuilder/StringBuffer equals() returns false even if contents are same?
  - equals() method is not overriden in StringBuffer/StringBuilder class.
  - So equals() from Object class is invoked. It compares addresses/references.
  - Now even if two StringBuffer/StringBuilder objects have same contents, their address will differ and equals() will return false.
- substr = str.substring(6, 1);
  - startIndex=6, endIndex=1 --> cause IndexOutOfBoundsException.

## Enum

- Day09 notes

## Exception chaining

- Pre-defined class

```
class HttpServlet ... {
    // ...
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException, ServletException
{
        // ...
    }
}
```

- User-defined class

```
class DatabaseServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException, ServletException
{
        try {
            // JDBC code throwing SQLException
        }
```

```
            catch(SQLException ex) {
                ex.printStackTrace();
                //throw ex; // to the Java web server -- cannot throw SQLException - throws clause
                throw new ServletException(ex); // to the Java web server
            }
        }
    }
```

- Example:

```
public static int divide(int num, int den) throws Exception {
    try {
        return num / den;
    }
    catch(ArithmeticException ex) {
        throw new Exception(ex);
    }
}
public static void main(String[] args) {
    try {
        int result = divide(22, 0);
        System.out.println(result);
    }
    catch(Exception ex) {
        ArithmeticException e = (ArithmeticException) ex.getCause();
        System.out.println(e.getMessage());
    }
}
```

- Sometimes an exception raised is not allowed to throw (due to throws clause). In such cases, raised exception is wrapped in some allowed exception type and thrown. This is exception chaining.

## 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, ...
- Two ways to do Generic Programming in Java
  - using java.lang.Object class -- Non typesafe -- Till Java 1.4
  - using Generics -- Typesafe -- Added in Java 5.0

## Generic Programming 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();  // ??
System.out.println("obj3 : " + obj3);
```

## Generic Programming Using Generics

- Added in Java 5.0.
- Similar to templates in C++.
- We can implement
  - Generic classes
  - Generic methods
  - Generic interfaces

## 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 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();  // ??
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<>(); // compiler error -- type must be given while creating generic class reference
                        // cannot be auto-detected

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

Box b5 = new Box(); // okay -- internally considered "Object" type -- compiler warning "raw types"

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

**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.

```
Box<Number> b1 = new Box<>(); // okay
Box<Boolean> b2 = new Box<>(); // error
Box<Character> b3 = new Box<>(); // error
Box<String> b4 = new Box<>(); // error
Box<Integer> b5 = new Box<>(); // okay
Box<Double> b6 = new Box<>(); // okay
Box<Date> b7 = new Box<>(); // error
Box<Object> b8 = new Box<>(); // erro
```

**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<Number> nb = new Box<Number>(123.45);
printBox(nb); // okay
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Object> ob = new Box<Object>("test");
```

```
printBox(ob); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // okay
```

**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<Number> nb = new Box<Number>(123.45);
printBox(nb); // okay
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Object> ob = new Box<Object>("test");
printBox(ob); // error
Box<Date> db = new Box<Date>(new Date());
printBox(db); // error
```

**Lower bounded generic types**

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

```
public static void printBox(Box<? super Number> b) {
    Object obj = b.get();
```

```
    System.out.println("Box contains: " + obj);
}
```

```
Box<Number> nb = new Box<Number>(123.45);
printBox(nb); // okay
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // error
Box<Object> ob = new Box<Object>("test");
printBox(ob); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // error
```

## 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
```

## Assignments

1. Copy Person class and inherited classes (Employee, Labor, Clerk) from previous assignment/classwork. Implement generic class Box so that it can store any Employee in it. Write a method in Box class that prints total salary of the Employee in it.
2. Write a generic static method to find minimum from an array of Number.
3. Copy Displayable, Person, Book, Car classes from the classwork. Add getter/setters and toString() in these classes. Complete the following non-generic methods with appropriate parameters and call them from main().

```
public static void printDisplayableBox(______ b) {
    b.get().display();
}
```

```
public static void printAnyBox(______ b) {
    System.out.println(b.get().toString());
}
```