Core Java

Agenda

- String
- StringBuffer and StringBuilder
- JVM architecture
- enum

Java Strings

- java.lang.Character is wrapper class that represents char. In Java, each char is 2 bytes because it follows unicode encoding.
- String is sequence of characters.
    1. java.lang.String: "Immutable" character sequence
    2. java.lang.StringBuffer: Mutable character sequence (Thread-safe)
    3. java.lang.StringBuilder: Mutable character sequence (Not Thread-safe)
- String helpers
    1. java.util.StringTokenizer: Helper class to split strings

String objects

- java.lang.String is class and strings in java are objects.
- String constants/literals are stored in string pool.

```
String str1 = "Sunbeam";
```

- String objects created using "new" operator are allocated on heap.

```
String str2 = new String("Nilesh");
```

- In java, String is immutable. If try to modify, it creates a new String object on heap.

String literals

- Since strings are immutable, string constants are not allocated multiple times.
- String constants/literals are stored in string pool. Multiple references may refer the same object in the pool.
- String pool is also called as String literal pool or String constant pool.
- From Java 7, String pool is in the heap space (of JVM).
- The string literal objects are created during class loading.

String objects vs String literals

- Important points
  - Two strings with same contents are never repeated in String pool.
  - == operator compares addresses/references of two objects.
  - equals() compares contents of two String objects -- case sensitive.
  - intern() adds a string in string pool (if not already present) and return its reference.
- Example 01:

```
String s1 = "Sunbeam";
String s2 = "Sunbeam";
System.out.println(s1 == s2);       // true
System.out.println(s1.equals(s2));  // true
```

- Example 02:

```
String s1 = new String("Sunbeam");
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);       // false
System.out.println(s1.equals(s2));  // true
```

- Example 03:

```
String s1 = "Sunbeam";
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);       // false
System.out.println(s1.equals(s2));  // true
```

- Example 04:

```
String s1 = "Sunbeam";
String s2 = "Sun" + "beam";
System.out.println(s1 == s2);       // true
System.out.println(s1.equals(s2));  // true
```

- Example 05:

```
String s1 = "Sunbeam";
String s2 = "Sun";
String s3 = s2 + "beam";
System.out.println(s1 == s3);       // false
System.out.println(s1.equals(s3));  // true
```

- Example 06:

```
String s1 = "Sunbeam";
String s2 = new String("Sunbeam").intern();
```

```
System.out.println(s1 == s2);       // true
System.out.println(s1.equals(s2));  // true
```

- Example 07:

```
String s1 = "Sunbeam";
String s2 = "SunBeam";
System.out.println(s1 == s2);       // false
System.out.println(s1.equals(s2));  // false
System.out.println(s1.equalsIgnoreCase(s2));  // true
System.out.println(s1.compareTo(s2));  // 32
System.out.println(s1.compareToIgnoreCase(s2));  // 0
```

- Lab assignment: Execute all above examples and confirm the output.

String operations

- int length()
- char charAt(int index)
- int compareTo(String anotherString)
- boolean equals(String anotherString)
- boolean equalsIgnoreCase(String anotherString)
- boolean matches(String regex)
- boolean isEmpty()
- boolean startsWith(String prefix)
- boolean endsWith(String suffix)
- int indexOf(int ch)
- int indexOf(String str)
- String concat(String str)
- String substring(int beginIndex)
- String substring(int beginIndex, int endIndex)

- String[] split(String regex)
- String toLowerCase()
- String toUpperCase()
- String trim()
- byte[] getBytes()
- char[] toCharArray()
- String intern()
- static String valueOf(Object obj)
- static String format(String format, Object... args)

StringBuffer vs StringBuilder

- StringBuffer and StringBuilder are final classes declared in java.lang package.
- It is used create to mutable string instance.
- equals() and hashCode() method is not overridden inside it.
- Can create instances of these classes using new operator only. Objects are created on heap.
- StringBuffer capacity grows if size of internal char array is less (than string to be stored).
    - The default capactiy is 16.

```
int max = (minimumCapacity > value.length? value.length * 2 + 2 : value.length);
minimumCapacity = (minimumCapacity < max? max : minimumCapacity);
char[] nb = new char[minimumCapacity];
```

- StringBuffer implementation is thread safe while StringBuilder is not thread-safe.
- StringBuilder is introduced in Java 5.0 for better performance in single threaded applications.

Examples

- Example 01:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
StringBuffer s2 = new StringBuffer("Sunbeam");
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 02:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
String s2 = new String("Sunbeam");
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 03:

```
String s1 = new String("Sunbeam");
StringBuffer s2 = new StringBuffer("Sunbeam");
System.out.println(s1.equals(s2));  // ???
System.out.println(s1.equals(s2.toString()));  // ???
```

- Example 04:

```
StringBuffer s1 = new StringBuffer("Sunbeam");
StringBuffer s2 = s1.reverse();
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 05:

```
StringBuilder s1 = new StringBuilder("Sunbeam");
StringBuilder s2 = new StringBuilder("Sunbeam");
System.out.println(s1 == s2);       // ???
System.out.println(s1.equals(s2));  // ???
```

- Example 06:

```
StringBuffer s = new StringBuffer();
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length()); // 16, 0
s.append("1234567890");
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length()); // 16, 10
s.append("ABCDEFGHIJKLMNOPQRSTUV");
System.out.println("Capacity: " + s.capacity() + ", Length: " + s.length()); // 34, 32
```

StringTokenizer

- Used to break a string into multiple tokens - like split() method.
- Methods of java.util.StringTokenizer
  - int countTokens()
  - boolean hasMoreTokens()
  - String nextToken()
  - String nextToken(String delim)
- Example:

```
String str = "My name is Bond, James Bond.";
String delim = " ,.";
StringTokenizer tokenizer = new StringTokenizer(str, delim);
while(tokenizer.hasMoreTokens()) {
    String token = tokenizer.nextToken();
```

```
        System.out.println(token);
    }
```

JVM Architecture

Java compilation process

- Hello.java --> Java Compiler --> Hello.class
  - javac Hello.java
- Java compiler converts Java code into the Byte code.

Byte code

- Byte code is machine level instructions that can be executed by Java Virtual Machine (JVM).
  - Instruction = Op code + Operands
    - e.g. iadd op1, op2
- Each Instruction in byte-code is of 1 byte.
  - .class --> JVM --> Windows (x86)
  - .class --> JVM --> Linux (ARM)
- JVM converts byte-code into target machine/native code (as per architecture).

.class format

- .class file contains header, byte-code, meta-data, constant-pool, etc.
- .class header contains
  - magic number -- 0xCAFEBABE (first 4 bytes of .class file)
  - information of other sections
- .class file can inspected using "javap" tool.
  - terminal> javap java.lang.Object
    - Shows public and protected members
  - terminal> javap -p java.lang.Object
    - Shows private members as well

    - terminal> javap -c java.lang.Object
        - Shows byte-code of all methods
    - terminal> javap -v java.lang.Object
        - Detailed (verbose) information in .class
            - Constant pool
            - Methods & their byte-code
            - ...
- "javap" tool is part of JDK.

Executing Java program (.class)

- terminal> java Hello
- "java" is a Java Application Launcher.
- java.exe (disk) --> Loader --> (Windows OS) Process
- When "java" process executes, JVM (jvm.dll) gets loaded in the process.
- JVM will now find (in CLASSPATH) and execute the .class.

JVM Architecture (Overview)

- JVM = Classloader + Memory Areas + Execution Engine

Classloader sub-system

- Load and initialize the class

Loading

- Three types of classloaders
    - Bootstrap classloader: Load Java builtin classes from jre/lib jars (e.g. rt.jar).
    - Extended classloader: Load extended classes from jre/lib/ext directory.
    - Application classloader: Load classes from the application classpath.
- Reads the class from the disk and loads into JVM method (memory) area.

Linking

- Three steps: Verification, Preparation, Resolution
- Verification: Byte code verifier does verification process. Ensure that class is compiled by a valid compiler and not tampered.
- Preparation: Memory is allocated for static members and initialized with their default values.
- Resolution: Symbolic references in constant pool are replaced by the direct references.

Initialization

- Static variables of the class are assigned with assigned values (static field initializers).
- Execute static blocks if present.

JVM memory areas

- During execution, memory is required for byte code, objects, variables, etc.
- There are five areas: Method area, Heap area, Stack area, PC Registers, Native Method Stack area.

Method area

- Created during JVM startup.
- Shared by all threads (global).
- Class contents (for all classes) are loaded into Method area.
- Method area also holds constant pool for all loaded classes.

Heap area

- Created during JVM startup.
- Shared by all threads (global).
- All allocated objects (with new keyword) are stored in heap.
- The class Metadata is stored in a java.lang.Class object (in heap) once class is loaded.
- The string pool is part of Heap area.

Stack area

- Separate stack is created for each thread in JVM (when thread is created).
- When a method is called from the stack, a new FAR (stack frame) is created on its stack.
- Each stack frame conatins local variable array, operand stack, and other frame data.
- When method returns, the stack frame is destroyed.

PC Registers

- Separate PC register is created for each thread. It maintains address of the next instruction executed by the thread.
- After an instruction is completed, the address in PC is auto-incremented.

Native method stack area

- Separate native method stack is created for each thread in JVM (when thread is created).
- When a native method is called from the stack, a stack frame is created on its stack.

Monitor memory areas

jconsole

- jconsole (JAVA_HOME/bin) can be used to monitor memory area.

Runtime class

- The Runtime class can be used to monitor JVM memory. The following code prints memory sizes in bytes.

```
class MemoryMontitor {
    public static void main(String[] args) {
        Runtime rt = Runtime.getRuntime();
        System.out.println("Max Memory: " + rt.maxMemory());
        System.out.println("Total Memory: " + rt.totalMemory());
        System.out.println("Free Memory: " + rt.freeMemory());
    }
}
```

Execution engine

- The main comonent of JVM.
- Execution engine executes for executing Java classes.

Interpreter

- Convert byte code into machine code and execute it (instruction by instruction).
- Each method is interpreted by the interpreter at least once.
- If method is called frequently, interpreting it each time slow down the execution of the program.
- This limitation is overcomed by JIT (added in Java 1.1).

JIT compiler

- JIT stands for Just In Time compiler.
- Primary purpose of the JIT compiler to improve the performance.
- If a method is getting invoked multile times, the JIT compiler convert it into native code and cache it.
- If the method is called next time, its cached native code is used to speedup execution process.

Profiler

- Tracks resource (memory, threads, ...) utilization for execution.
- Part of JIT that identifies hotspots. It counts number of times any method is executing. If the number is more than a threshold value, it is considered as hotspot.

Garbage collector

- When any object is unreferenced, the GC release its memory.

JNI

- JNI acts as a bridge between Java method calls and native method implementations.

enum

- "enum" keyword is added in Java 5.0.
- Used to make constants to make code more readable.
- Typical switch case

```
int choice;
// ...
switch(choice) {
    case 1: // addition
        c = a + b;
        break;
    case 2: // subtraction
        c = a - b;
        break;
    // ...
}
```

- The switch constants can be made more readable using Java enums.

```
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULIPLICATION, DIVISION;
}

ArithmeticOperations choice = ArithmeticOperations.ADDITION;
// ...
switch(choice) {
    case ADDITION:
        c = a + b;
        break;
    case SUBTRACTION:
```

```
            c = a - b;
            break;
        // ...
    }
```

- In java, enums cannot be declared locally (within a method).
- The declared enum is converted into enum class.

```
// user-defined enum
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULIPLICATION, DIVISION
}
```

```
// generated enum code
final class ArithmeticOperations extends Enum {
    public static ArithmeticOperations[] values() {
        return (ArithmeticOperations[])$VALUES.clone();
    }
    public static ArithmeticOperations valueOf(String s) {
        return (ArithmeticOperations)Enum.valueOf(ArithmeticOperations, s);
    }
    private ArithmeticOperations(String name, int ordinal) {
        super(name, ordinal); // invoke sole constructor Enum(String,int);
    }
    public static final ArithmeticOperations ADDITION;
    public static final ArithmeticOperations SUBTRACTION;
    public static final ArithmeticOperations MULIPLICATION;
    public static final ArithmeticOperations DIVISION;
    private static final ArithmeticOperations $VALUES[];
    static {
        ADDITION = new ArithmeticOperations("ADDITION", 0);
        SUBTRACTION = new ArithmeticOperations("SUBTRACTION", 1);
```

```
        MULIPLICATION = new ArithmeticOperations("MULIPLICATION", 2);
        DIVISION = new ArithmeticOperations("DIVISION", 3);
        $VALUES = (new ArithmeticOperations[] {
            ADDITION, SUBTRACTION, MULIPLICATION, DIVISION
        });
    }
}
```

- The enum type declared is implicitly inherited from java.lang.Enum class. So it cannot be extended from another class, but enum may implement interfaces.
- The enum constants declared in enum are public static final fields of generated class. Enum objects cannot be created explicitly (as generated constructor is private).
- The generated class will have a values() method that returns array of all constants and valueOf() method to convert String to enum constant.
- The enums constants can be used in switch-case and can also be compared using == operator.
- The enum may have fields and methods.

```
enum Element {
    H(1, "Hydrogen"),
    HE(2, "Helium"),
    LI(3, "Lithium");

    private final int atomNum;
    private final String atomName;

    private Element(int atomNum, String atomName) {
        this.atomNum = atomNum;
        this.atomName = atomName;
    }

    public int getAtomNum() {
        return atomNum;
    }
    public String getAtomName() {
```

```
        return atomName;
    }

    // ...
}
```

- The java.lang.Enum class has following members:

```
public abstract class Enum<E> implements java.lang.Comparable<E>, java.io.Serializable {
    private final String name;
    private final int ordinal;

    protected Enum(String,int); // sole constructor - can be called from user-defined enum class only
    public final String name(); // name of enum const
    public final int ordinal(); // position of enum const (0-based)

    public String toString(); // returns name of const
    public final int compareTo(E);// compares with another enum of same type on basis of ordinal number
    public static <T> T valueOf(Class<T>, String);
    // ...(
}
```

Assignments

1. Type String, StringBuffer and StringBuilder code snippets from the notes above. Execute them and confirm their outputs.
2. Write methods to perform following string conversions.
    - Bank Of Maharashtra --> BOM
    - this is string --> This Is String
    - Example --> eXaMpLe
    - www.sunbeaminfo.com --> www.sunbeaminfo.in
3. From an array of strings print the strings ending with vowels.
4. Write a method to check if string is palindrome or not.

5. Modify assignment 8 question 2 -- Implement toString() method in Salesman, Labor, and Clerk class using String.format() and toString() of Person and Employee using StringBuilder.

```
class Labor {
    // ...
    public String toString() {
        return String.format("Hours: %d, Rate: %f\n", this.hours, this.rate);
    }
}
```

6. Declare an enum for Gender (MALE, FEMALE). Create a class Person with fields name, age and gender. In main(), create a array of Person. Write a menu driven program (using enum), to add new person, display all people, find a person by name.