Core Java

equals() method

- Non-final method of java.lang.Object class.
  - public boolean equals(Object other);
- Definition of Object.equals():

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- To compare the object contents/state, programmer should override equals() method.
- This equals() must have following properties:
  - Reflexive: for any non-null reference value x, x.equals(x) should return true.
  - Symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
  - Transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
  - Consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
  - For any non-null reference value x, x.equals(null) should return false.
- Example:

```
class Employee {
    // ...
    @Override
    public boolean equals(Object obj) {
        if(obj == null)
            return false;

        if(this == obj)
            return true;

        if(! (obj instanceof Employee))
            return false;

        Employee other = (Employee) obj;
        if(this.id == other.id)
            return true;
        return false;
    }
}
```

Garbage collection

- Garbage collection is automatic memory management by JVM.
- If a Java object is unreachable (i.e. not accessible through any reference), then it is automatically released by the garbage collector.
- An object become eligible for GC in one of the following cases:
    - Nullify the reference.

```
MyClass obj = new MyClass();
obj = null;
```

    - Reassign the reference.

```
MyClass obj = new MyClass();
obj = new MyClass();
```

    - Object created locally in method.

```
void method() {
    MyClass obj = new MyClass();
    // ...
}
```

- GC is a background thread in JVM that runs periodically and reclaim memory of unreferenced objects.
- Before object is destroyed, its finalize() method is invoked (if present).
- One should override this method if object holds any resource to be released explicitly e.g. file close, database connection, etc.

```
class MyClass {
    private Connection con;
    public MyClass() throws Exception {
        con = DriverManager.getConnection("url", "username", "password");
    }
    // ...
    @Override
    public void finalize() {
        try {
            if(con != null)
                con.close();
        }
        catch(Exception e) {
```

```
        }
    }
}
class Main {
    public static void method() throws Exception {
        MyClass my = new MyClass();
        my = null;
        System.gc(); // request GC
    }
    // ...
}
```

- GC can be requested (not forced) by one of the following.
    - System.gc();
    - Runtime.getRuntime().gc();
- JVM GC internally use Mark and Compact algorithm.
- GC Internals: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Marker interfaces

- Interface that doesn't contain any method declaration is called as "Marker interface".
- These interfaces are used to mark or tag certain functionalities/features in implemented class. In other words, they associate some information (metadata) with the class.
- Marker interfaces are used to check if a feature is enabled/allowed for the class.
- Java has a few pre-defined marker interfaces. e.g. Serializable, Cloneable, etc.
    - java.io.Serializable -- Allows JVM to convert object state into sequence of bytes.
    - java.lang.Cloneable -- Allows JVM to create copy of the class object.

Cloneable interface

- Enable creating copy/clone of the object.
- If a class is Cloneable, Object.clone() method creates a shallow copy of the object. If class is not Cloneable, Object.clone() throws CloneNotSupportedException.
- A class should implement Cloneable and override clone() to create a deep/shallow copy of the object.

```
class Date implements Cloneable {
    private int day, month, year;
    // ...
    // shallow copy
    public Object clone() throws CloneNotSupportedException {
        Date temp = (Date)super.clone();
        return temp;
    }
}
```

```
class Person implements Cloneable {
    private String name;
    private int weight;
    private Date birth;
    // ...
    // deep copy
    public Object clone() throws CloneNotSupportedException {
        Person temp = (Person)super.clone(); // shallow copy
        temp.birth = (Date)this.birth.clone(); // + copy reference types
explicitly
        return temp;
    }
}
```

```
class Program {
    public static void main(String[] args) throws CloneNotSupportedException
{
        Date d1 = new Date(28, 9, 1983);
        System.out.println("d1 = " + d1.toString());
        Date d2 = (Date)d1.clone();
        System.out.println("d2 = " + d2.toString());
        Person p1 = new Person("Nilesh", 70, d1);
        System.out.println("p1 = " + p1.toString());
        Person p2 = (Person)p1.clone();
        System.out.println("p2 = " + p2.toString());
    }
}
```

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

- Example 01:

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

- Example 02:

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

- Example 03:

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

- Example 04:

```
String s1 = "Sunbeam";
String s2 = "Sun" + "beam";
```

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

- Example 05:

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

- Example 06:

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

- Example 07:

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

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

- Example 02:

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

- Example 03:

```
String s1 = new String("Sunbeam");
StringBuffer s2 = new StringBuffer("Sunbeam");
System.out.println(s1.equals(s2));  // false -- String compared with
StringBuffer
System.out.println(s1.equals(s2.toString()));  // true -- String compared
with String
```

- Example 04:

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

- Example 05:

```
StringBuilder s1 = new StringBuilder("Sunbeam");
StringBuilder s2 = new StringBuilder("Sunbeam");
System.out.println(s1 == s2);       // false
System.out.println(s1.equals(s2));  // false -- calls Object.equals()
```

- 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
```

Resource Management

- System resources should be released immediately after the use.
- Few system resources are Memory, File, IO Devices, Socket/Connection, etc.
- The Garbage collector automatically releases memory if objects are no more used (unreferenced).
- The GC collector doesn't release memory/resources immediately; rather it is executed only memory is full upto a threshold.
- The standard way to release the resources immediately after their use is java.io.Closeable interface. It has only one method.
    - void close() throws IOException;
- Programmer should call close() explicitly on resource object after its use.

    - e.g. FileInputStream, FileOutputStream, etc.
- Java 7 introduced an interface java.lang.AutoCloseable as super interface of Closeable. It has only one method.
    - void close() throws Exception;
- Since it is super-interface of Closeable, all classes implementing Closeable now also inherit from AutoCloseable.
- If a class is inherited from AutoCloseable, then it can be closed using try-with-resource syntax.

```
class MyResource implements AutoCloseable {
    // ...
    public void close() {
        // cleanup code
    }
}

class Program {
    public static void main(String[] args) {
        try(MyResource res = new MyResource()) {
            // ...
        } // res.close() called automatically
    }
}
```

- The Scanner class is also AutoCloseable.

```
class Program {
    public static void main(String[] args) {
        try(Scanner sc = new Scanner(System.in)) {
            // ...
        } // sc.close() is auto-closed
    }
}
```

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 given values (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.

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.