Agenda

- Final
- toString()
- equals()
- Abstract class/method
- Interfaces Till Java 7
- Marker interfaces
- JVM Architecture
- Java Buzzwords
- ~~Exception Handling~~

toString() method

- it is a non final method of object class
- To return state of Java instance in String form, programmer should override toString() method.
- The result in toString() method should be a concise, informative, and human-readable.
- It is recommended that all subclasses override this method.

equals() method

- It is non final method of object class
- 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.
- It is recommended to override hashcode method along when equals method is overriden.

final

- In Java, const is reserved word, but not used.
- Java has final keyword instead.
- It can be used for
    - variables
    - fields
    - methods
    - class

- if variables and fields are made final, they cannot be modified after initialization.
- final fields of the class must be initialized using any of the following below
    - field initializer
    - object initializer
    - constructor
- final methods cannot be overriden, final class cannot be extended(we will see at the time of inheritance)

final Method

- If implementation of a super-class method is logically complete, then the method should be declared as final.
- Such final methods cannot be overridden in sub-class. Compiler raise error, if overridden.
- But final methods are inherited into sub-class i.e. The super-class final methods can be invoked in sub-class object (if accessible).

final Class

- If implementation of a super-class is logically complete, then the class should be declared as final.
- The final class cannot be extended into a sub-class. Compiler raise error, if inherited.
- Effectively all methods in final class are final methods.
- Examples of final classes
    - java.lang.Integer (and all wrapper classes)
    - java.lang.String
    - java.lang.System

Abstract Methods

- If implementation of a method in super-class is not possible/incomplete, then method is declared as abstract.
- Abstract method does not have definition/implementation.
- If class contains one or more abstract methods, then class must be declared as abstract. Otherwise compiler raise an error.
- The super-class abstract methods must be overridden in sub-class; otherwise sub-class should also be marked abstract.
- The abstract methods are forced to be implemented in sub-class. It ensures that sub-class will have corresponding functionality.
- The abstract method cannot be private, final, or static.
- Example: abstract methods declared in Number class are:
    - abstract int intValue();
    - abstract float floatValue();

Abstract class

- If implementation of a class is logically incomplete, then the class should be declared abstract.
- If class contains one or more abstract methods, then class must be declared as abstract.
- An abstract class can have zero or more abstract methods.

- Abstract class object cannot be created; however its reference can be created.
- Abstract class can have fields, methods, and constructor.
- Its constructor is called when sub-class object is created and initializes its (abstract class) fields.
- Example:
- java.lang.Number
- java.lang.Enum

Fragile base class problem

- If changes are done in super-class methods (signatures), then it is necessary to modify and recompile all its sub-classes. This is called as "Fragile base class problem".
- This can be overcomed by using interfaces.

Interface (Java 7 or Earlier)

- Interfaces are used to define standards/specifications.
- A standard/specification is set of rules.
- Interfaces are immutable i.e. once published interface should not be modified.
- Interfaces contains only method declarations. All methods in an interface are by default abstract and public.
- They define a "contract" that is must be followed/implemented by each sub-class.
- Interfaces enables loose coupling between the classes i.e. a class need not to be tied up with another class implementation.
- Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces.
- Java 7 interface can only contain public abstract methods and static final fields (constants).
- They cannot have non-static fields, static methods, and constructors.
- Examples:
  - java.io.Closeable / java.io.AutoCloseable
  - java.lang.Runnable
- Multiple interface inheritance is allowed in Java

```
interface Displayable {
void display();
}
interface Acceptable {
void accept();
}
class Employee implements Displayable,Acceptable{}
```

- If two interfaces have same method, then it is implemented only once in sub-class.

class vs abstract class vs interface

- class
  - Has fields, constructors, and methods

    - Can be used standalone -- create objects and invoke methods
    - Reused in sub-classes -- inheritance
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism

- abstract class
    - Has fields, constructors, and methods
    - Cannot be used independently -- can't create object
    - Reused in sub-classes -- inheritance -- Inherited into sub-class and must override abstract methods
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism

- interface
    - Has only method declarations
    - Cannot be used independently -- can't create object
    - Doesn't contain anything for reusing (except static final fields)
    - Used as contract/specification -- Inherited into sub-class and must override all methods
    - Can invoke overridden methods in sub-class using super-class reference -- runtime polymorphism

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.

JVM Archicecture

- 1. Compilation
    - .class file is cretaed which consists of byte code
- 2. Byte Code
    - It is a machine level instructions that gets executed by the JVM
    - JVM converts byte code into target machine/native code
- 3. Execution
    - java is a tool used to execute the .class file.
    - It loads the .class file and invokes jvm for executing the file from the classpath
- JVM Archiceture Overview
    - ClassLoader + Memory Area + Execution Engine

ClassLoader SubSystem

- It loads and initialize the class

1. Loading

- Three types of classLoaders
    - 1. BootStrap classloader that loads built in java classes from jre/lib jars (rt.jar)
    - 2. Extension classloader that loads the extended classes from jre/lib/ext directory
    - 3. Application classloader that loads the classes from the application classpath
- It reads the classes from the disk and loads into JVM method(memory) area

2. Linking

- Three steps
    - 1. Verifiaction : Bytecode verifier ensures that class is compiled by valid compiler and not tampered
    - 2. Preparation : Memory is allocated for static members and initialized with default values
    - 3. Resolution : Symbolic references in constant pool are replaced by the direct references

3. Initialization

- All static variables of class are assigned with their assigned values(field initializers)
- all static blocks are executed if present

Memory Areas

- Their are 5 memory areas
    - 1. Method Area
    - 2. heap Area
    - 3. Stack Area
    - 4. PC Registers
    - 5. Native Method Stack Area

1. Method Area

- Create during JVM startup
- shared by all the threads
- class contents (for all classes) loaded into this area
- Method area also holds constant pool for all loaded classes.

2. Heap Area

- Create during JVM startup
- shared by all the threads
- All allocated objects (with new) are stored in heap
- The string pool is part of heap Area.
- The class Metadata is stored in a java.lang.Class object (in heap) once class is loaded.

3. Stack Area

- Separate stack is created for each thread in JVM (when thread is created).
- When a method is called 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.

4. 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.

5. Native Method Stack

- 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 component of JVM
- Convert byte code into machine code and execute it (instruction by instruction).
- It consists of
    - 1. Interpreter
    - 2. JIT Compiler
    - 3. Garbage Collector

1. Interpreter

- 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).

2. 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.

3. 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.

4. 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.

Java BuzzWords

- 1. Simple
  - Simple for Professional Programmers if aware about OOP.
  - It removed the complicated fetaures like pointers and rarely used features like operator overloading from c++
  - It was simple till java 1.4
  - the new features added made it powerful (but also complex)
- 2. Object Oriented
  - Java is a object-oriented programming language.
  - It supports all the pillars of OOP
- 3. Distributed
  - Java is designed to create distributed applications on networks.
  - Java applications can access remote objects on the Internet as easily as they can do in the local system.
  - Java enables multiple programmers at multiple remote locations to collaborate and work together on a single project.
- 4. Compiled and Interpreted
  - Usually, a computer language is either compiled or Interpreted.
  - Java combines both this approach and makes it a two-stage system.
  - Compiled: Java enables the creation of cross-platform programs by compiling them into an intermediate representation called Java Bytecode.
  - Interpreted: Bytecode is then interpreted, which generates machine code that can be directly executed by the machine/CPU.
- 5. Robust
  - It provides many features that make the program execute reliably in a variety of environments.
  - Java is a strictly typed language. It checks code both at compile time and runtime.
  - Java takes care of all memory management problems with garbage collection.
- 6. Secure
  - Java achieves this protection by confining a Java program to the Java execution environment and not allowing it to access other parts of the computer
- 7. Architecture Neutral
  - Java language and Java Virtual Machine helped in achieving the goal of WORA - Write Once Run Anywhere.
  - Java byte code is interpreted by JIT and convert into CPU machine code/native code.
  - So Java byte code can execute on any CPU architecture (on which JVM is available)
- 8. Portable
  - As java is Architecture Neutral it is portable.
  - Java is portable because of the Java Virtual Machine (JVM).
- 9. High Performance
  - Java performance is high because of the use of bytecode.
  - The bytecode was used so that it can be efficiently translated into native machine code by JIT compiler (in JVM).
- 10. Multithreaded
  - Multithreaded Programs handled multiple tasks simultaneously (within a process)
  - Java supports multi-process/thread communication and synchronization.
  - When Java application executes 2 threads are started
    - 1. main thread

        - 2. garbage collector thread.
- 11. Dynamic
    - Java is capable of linking in new class libraries, methods, and objects.
    - Java classes has run-time type information that is used to verify and resolve accesses to objects/members at runtime.