Core Java

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.

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

    - Reassign the reference.

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

    - Object created locally in method.

      ```java
      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.

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

```java
                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

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.

Exception Handling

- Exceptions represents runtime problems.
- If these problems may not be handled in the current method, so they can be sent back to the calling method.
- Java keywords for exception handling
  - throw
  - try
  - catch
  - finally
  - throws
- Example 1:

```
static double divide(int numerator, int denominator) {
    if(denominator == 0)
        throw new RuntimeException("Cannot divide by zero");
    return (double)numerator / denominator;
}
public static void main(String[] args) {
    // ...
    try {
        int num1 = sc.nextInt();
        int num2 = sc.nextInt();
        double result = divide(num1, num2);
        System.out.println("Result: " + result);
    }
    catch(RuntimeException e) {
        e.printStackTrace();
    }
}
```

- Java operators/APIs throw pre-defined exception if runtime problem occurs.
- For example, ArithmeticException is thrown when divide by zero is tried.
- Example 2:

```
static int divide(int numerator, int denominator) {
    return numerator / denominator;
}
public static void main(String[] args) {
    // ...
    try {
        int num1 = sc.nextInt();
        int num2 = sc.nextInt();
        int result = divide(num1, num2);
        System.out.println("Result: " + result);
    }
    catch(ArithmeticException e) {
        e.printStackTrace();
    }
}
```

- Java exception class hierarchy

```
Object
    |- Throwable
        |- Error
        |   |- AssertionError
        |   |- VirtualMachineError
        |           |- StackOverflowError
        |           |- OutOfMemoryError
        |- Exception
            |- CloneNotSupportedException
            |- IOException
            |       |- EOFException
            |       |- FileNotFoundException
            |- SQLException
            |- InterruptedException
            |- RuntimeException
                |- NullPointerException
                |- ArithmeticException
                |- NoSuchElementException
                |       |- InputMismatchException
```

```
                    |- IndexOutOfBoundsException
                            |- ArrayIndexOutOfBoundsException
                            |- StringIndexOutOfBoundsException
```

- One catch block cannot handle problems from multiple try blocks.
- One try block may have multiple catch blocks. Specialized catch block must be written before generic catch block.
- If certain code to be executed irrespective of exception occur or not, write it in finally block.
- Example:

```
try {
    // file read code -- possible problems
        // 1. file not found
        // 2. end of file is reached
        // 3. error while reading from file
        // 4. null reference (programmer's mistake)
}
catch(NullPointerException ex) {
    // ...
}
catch(FileNotFoundException ex) {
    // ...
}
catch(EOFException ex) {
    // ...
}
catch(IOException ex) {
    // ...
}
finally {
    // close the file
}
```

- When exception is raised, it will be caught by nearest matching catch block. If no matching catch block is found, the exception will be caught by JVM and it will abort the program.

java.lang.Throwable class

- Throwable is root class for all errors and exceptions in Java.
- Only objects of
- Methods
  - Throwable()
  - Throwable(String message)
  - Throwable(Throwable cause)
  - Throwable(String message, Throwable cause)
  - String getMessage()
  - void printStackTrace()
  - void printStackTrace(PrintStream s)

    - void printStackTrace(PrintWriter s)
    - String toString()
    - Throwable getCause()

java.lang.Error class

- Usually represents the runtime problems that are not recoverable.
- Generated due to environmental condition/Runtime environment (e.g. OS error, Memory error, etc.)
- Examples:
    - AssertionError
    - VirtualMachineError
    - StackOverflowError
    - OutOfMemoryError

java.lang.Exception class

- Represents the runtime problems that can be handled.
- The exception is handled using try-catch block.
- Examples:
    - CloneNotSupportedException
    - IOException
    - SQLException
    - NullPointerException
    - ArrayIndexOutOfBoundsException
    - ClassCastException

Exception types

- There are two types of exceptions
    - Checked exception -- Checked by compiler and forced to handle
    - Unchecked exception -- Not checked by compiler

Unchecked exception

- RuntimeException and all its sub classes are unchecked exceptions.
- Typically represents programmer's or user's mistake.
    - NullPointerException, NumberFormatException, ClassCastException, etc.
- Compiler doesn't provide any checks -- if exception is handled or not.
- Programmer may or may not handle (catch block) the exception. If exception is not handled, it will be caught by JVM and abort the application.

Checked exception

- Exception and all its sub classes (except RuntimeException) are checked exceptions.
- Typically represents problems arised out of JVM/Java i.e. at OS/System level.
    - IOException, SQLException, InterruptedException, etc.
- Compiler checks if the exception is handled in one of following ways.
    - Matching catch block to handle the exception.

```
void someMethod() {
    try {
        // file io code
    }
    catch(IOException ex) {
        // ...
    }
}
```

- throws clause indicating exception to be handled by calling method.

```
void someMethod() throws IOException {
    // file io code
}
void callingMethod() {
    try {
        someMethod();
    }
    catch(IOException ex) {
        // ...
    }
}
```

Exception handling keywords

- "try" block
  - Code where runtime problems may arise should be written in try block.
  - try block must have one of the following
    - catch block
    - finally block
    - try-with-resource
  - Can be nested in try, catch, or finally block.
- "throw" statement
  - Throws an exception/error i.e. any object that is inherited from Throwable class.
  - Can throw only one exception at time.
  - All next statements are skipped and control jumps to matching catch block.
- "catch" block
  - Code to handle error/exception should be written in catch block.
  - Argument of catch block must be Throwable or its sub-class.
  - Generic catch block -- Performs upcasting -- Should be last (if multiple catch blocks)

```
try {
    // ...
}
catch(Throwable e) {
    // can handle exception of any type
}
```

- Multi-catch block -- Same logic to execute different exceptions

    ```
    try {
        // ...
    }
    catch(ArithmeticException|InputMismatchException e) {
        // can handle exception of any type
    }
    ```

- Exception sub-class should be caught before super-class.

    ```
    try {
        // ...
    }
    catch(EOFException ex) {
        // ...
    }
    catch(IOException ex) {
        // ...
    }
    ```

- "finally" block
    - Resources are closed in finally block.
    - Executed irrespective of exception occurred or not.
    - finally block not executed if JVM exits (System.exit()).

    ```
    Scanner sc = new Scanner(System.in);
    try {
        // ...
    }
    catch(Exception ex) {
        // ...
    }
    finally {
        sc.close();
    }
    ```

- "throws" clause
    - Written after method declaration to specify list of exception not handled by called method and to be handled by calling method.
    - Writing unhandled checked exceptions in throws clause is compulsory. The unchecked exceptions written in throws clause are ignored by the compiler.

```
void someMethod() throws IOException, SQLException {
    // ...
}
```

  - Sub-class overridden method can throw same or subset of exception from super-class method.

```
class SuperClass {
    // ...
    void method() throws IOException, SQLException,
InterruptedException {

    }
}
class SubClass extends SuperClass {
    // ...
    void method() throws IOException, SQLException {

    }
}
```

```
class SuperClass {
    // ...
    void method() throws IOException {

    }
}
class SubClass extends SuperClass {
    // ...
    void method() throws FileNotFoundException, EOFException {

    }
}
```

Exception Handling

Exception chaining

- Sometimes an exception is generated due to another exception.
- For example, database SQLException may be caused due to network problem SocketException.
- To represent this an exception can be chained/nested into another exception.
- If method's throws clause doesn't allow throwing exception of certain type, it can be nested into another (allowed) type and thrown.

User defined exception class

- If pre-defined exception class are not suitable to represent application specific problem, then user-defined exception class should be created.
- User defined exception class may contain fields to store additional information about problem and methods to operate on them.
- Typically exception class's constructor call super class constructor to set fields like message and cause.
- If class is inherited from RuntimeException, it is used as unchecked exception. If it is inherited from Exception, it is used as checked exception.

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
  - using Generics -- Typesafe

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();  // ClassCastException
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

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

```java
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 -- type of object is inferred/guessed looking at reference declaration

Box<> b3 = new Box<>(); // compiler error -- type must be given while declaring reference

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

Box b5 = new Box(); // okay -- compiler warning "raw types" -- internally considered as Object type (for T).
    // not recommended -- doesn't do compiler time type-checking

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.

```
// T can be any type so that T is Number or its sub-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<>(); // 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); // okay
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // okay
Box<Float> fb = new Box<Float>(200.5f);
printBox(fb); // 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<String> sb = new Box<String>("DAC");
printBox(sb); // error
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // error
Box<Object> ob = new Box<Object>(new Object());
printBox(ob); // 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<String> sb = new Box<String>("DAC");
printBox(sb); // error
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // error
Box<Object> fb = new Box<Object>(new Object());
printBox(fb); // okay
Box<Number> nb = new Box<Number>(null);
printBox(nb); // okay
```