Core Java

Day 08 Agenda

- Garbage collection
- Resource management
- Exception handling

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();
    ```

```
    // ...
}
```

- Island of isolation i.e. objects are referencing each other, but not referenced externally.

```
class FirstClass {
    private SecondClass second;
    public void setSecond(SecondClass second) {
        this.second = second;
        second.setFirst(this);
    }
}
class SecondClass {
    private FirstClass first;
    public void setFirst(FirstClass first) {
        this.first = first;
    }
}
class Main {
    public static void method() {
        FirstClass f = new FirstClass();
        f.setSecond(new SecondClass());
        f = null;
    }
    // ...
}
```

* 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 {
            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();
- GC is of two types i.e. Minor and Major.
  - Minor GC: Unreferenced objects from young generation are reclaimed. Objects not reclaimed here are moved to old/permanent generation.
  - Major GC: Unreferenced objects from all generations are reclaimed. This is unefficient (slower process).
- JVM GC internally use Mark and Compact algorithm.
- GC Internals: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

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
    }
}
```

Exception Handling

- Exceptions represents runtime problems.
- If these problems cannot be handled in the current method, then they should 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) {
    // common logic to handle ArithmeticException and InputMismatchException
}
```

- 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 only if JVM/application 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 {

    }
}
```

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

    }
}
```

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.

Assignments

1. A shop sells different types of products like books, music albums, and toys. Book information includes ISBN, title, price, author, and subject. Album information includes title, price, singer, and musician. Toy information includes title, price, age group (string), type. Though books are tax free, music albums have GST tax of 10% and toys have VAT tax of 5%. Sometimes shop keeper announce a sell, where he apply same percentage discount to every purchased products. Assuming that each customer can purchase maximum 5 products at a time, write a menu driven program so that each user can purchase products of his choice. At the end display total bill (including tax) to be paid by customer and total revenue of shop (excluding tax). Design appropriate classes and their relations.
2. Write a Person class with fields (name, age) and appropriate constructors + getter/setters + equals(). Write an Employee class inherited from Person class with additional fields (id and salary). Add abstract method double calcSalary(). Write a Labor class inherited from Employee class with additional fields (rate and hours). Override calcSalary() as hours * rate. Write a Salesman class inherited from Employee class with additional fields (target and

commission). Override calcSalary() as salary + commission. Write a Clerk class with no additional fields and calcSalary() returns basic salary only. The setter methods of all classes should throw a custom exception EmployeeException (with additional field - invalidValue), if invalid values are set. Create Employees helper class helper methods as follows. In main(), create array of Employees and initialize with appropriate objects. Call these Employees helper methods and display results.

- double averageSalManagers(Employee[] arr);
- double averageSalSalesmans(Employee[] arr);
- double averageSalClerks(Employee[] arr);