

|  |  |
| --- | --- |
| **ASSIGNMENT** | |
| **Course Code** | 19CSC213A |
| **Course Name** | Programming Paradigms |
| **Programme** | B. Tech |
| **Department** | Computer Science & Engineering |
| **Faculty** | Faculty of Engineering Technology |

#### 

|  |  |
| --- | --- |
| **Name of the Student** | SUBHENDU MAJI |
| **Reg. No** | 18ETCS002121 |
| **Semester/Year** | 4th / 2020 |
| **Course Leader/s** | Ms. Naveeta |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Declaration Sheet** | | | | | | | | |
| Student Name | SUBHENDU MAJI | | | | | | | |
| Reg. No | 18ETCS002121 | | | | | | | |
| Programme | B. Tech | | | | | Semester/Year | 4th / 2020 | |
| Course Code | 19CSC213A | | | | | | | |
| Course Title | Programming Paradigms | | | | | | | |
| Course Date |  | | To | |  | | | |
| Course Leader | Ms. Naveeta | | | | | | | |
| **Declaration**  The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly. | | | | | | | | |
| Signature of the Student | |  | | | | | Date |  |
| Submission date stamp  (by Examination & Assessment Section) | |  | | | | | | |
| Signature of the Course Leader and date | | | | Signature of the Reviewer and date | | | | |
|  | | | |  | | | | |

# **Contents**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[**Declaration Sheet** ii](#_Toc37456523)

[**Contents** iii](#_Toc37456524)

[**Question No. 1** 5](#_Toc37456525)

[A1.1 Illustration of how C programmers manually and explicitly free memory 5](#_Toc37456526)

[A1.2 Illustration of how Java programmers rely on the Garbage Collector 6](#_Toc37456527)

[A1.3 Comparison of the manual and the automated approaches 10](#_Toc37456528)

[A1.4 The stance taken with justification 11](#_Toc37456529)

[**Question No. 2** 12](#_Toc37456530)

[B1.1 Introduction to an Airline reservation system 12](#_Toc37456531)

[B1.2 Design of the Airline reservation system 12](#_Toc37456532)

[B1.3 Implementation of the Airline reservation system 13](#_Toc37456533)

[Bibliography 21](#_Toc37456534)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Assignment** | | | | |  | | | |
| Register No. | | | 18ETCS002121 | Name of Student |  | SUBHENDU MAJI | | |
|  |  | **Marking Scheme** | | |  | |  |  |
|  | QA.1 | Illustration of how C programmers manually and explicitly free memory | | | 01 | |  |  |
| QA.2 | Illustration of how Java programmers rely on the Garbage Collector | | | 01 | |  |  |
| QA.3 | Comparison of the manual and the automated approaches | | | 02 | |  |  |
| QA.4 | The stance taken with justification. | | | 02 | |  |  |
|  | **Part A Max Marks** | | | **6** | |  |  |
|  | QB.1 | Introduction to an Airline reservation system | | | 01 | |  |  |
| QB.2 to QB.8 | Design of the Airline reservation system | | | 05 | |  |  |
| QB.9 | Implementation of the Airline reservation system | | | 06 | |  |  |
|  | QB.10 | Demonstration of the Airline reservation system | | | 02 | |  |  |
|  | **Part B: Max Marks** | | | **14** | |  |  |
|  | **Total Assignment Marks** | | | | **20** | |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Course Marks Tabulation** | | | | |
| **Component- 1(B) Assignment** | **First**  **Examiner** | **Remarks** | **Moderator** | **Remarks** |
| Q1 |  |  |  |  |
| Q2 |  |  |  |  |
| **Marks (out of 20 )** |  |  |  |  |
| Signature of First Examiner Signature of Second Examiner | | | | |

# **Question No. 1**

**Solution to Question No. 1:**

## A1.1 Illustration of how C programmers manually and explicitly free memory

Dynamic memory allocation in C language programming is the process of memory allocation from heap memory at run time. In other word, when memory is allocated from heap during program execution is called dynamic memory allocation.

Also note that, in dynamic memory allocation, memory get allocated from heap and not from stack.

**How dynamic memory allocation in C is done?**

Dynamic memory allocation in C programming can be done using standard library functions – malloc(), calloc(), realloc() and free(). All these library functions are available in “stdlib.h” header file. Hence, we have to include this header in c program.

malloc (), calloc() and realloc () – perform memory allocation and free() function perform de-allocation. It is important to note that if we allocate memory dynamically then we must de-allocate it manually / explicitly using free () method. Because, dynamically allocated memory can’t be freed automatically like static memory allocation in C.

|  |  |  |
| --- | --- | --- |
| **Function** | **Purpose** | **Syntax** |
| **malloc** | Allocates the memory of requested size and returns the pointer to the first byte of allocated space. | ptr = (cast\_type \*) malloc (byte\_size); |
| **calloc** | Allocates the space for elements of an array. Initializes the elements to zero and returns a pointer to the memory. | ptr=(cast\_type\*)calloc(n, size); |
| **realloc** | It is used to modify the size of previously allocated memory space. | ptr = realloc (ptr,newsize); |
| **Free** | Frees or empties the previously allocated memory space. | free(ptr); |

## A1.2 Illustration of how Java programmers rely on the Garbage Collector

Garbage collection relieves programmers from the burden of freeing allocated memory. Knowing when to explicitly free allocated memory can be very tricky. Giving this job to the JVM has several advantages.

A garbage collector is responsible for

Allocating memory

Ensuring that any referenced objects remain in memory, and

Recovering memory used by objects that are no longer reachable from references in executing code.

The automatic garbage collector figures out which objects are not reachable and, therefore, eligible for garbage collection. It will certainly go to work if there is a danger of running out of memory. Although the automatic garbage collector tries to run unobtrusively, certain programming practices can nevertheless help in minimizing the overhead associated with garbage collection during program execution. Automatic garbage collection should not be perceived as a license for uninhibited creation of objects and forgetting about them.

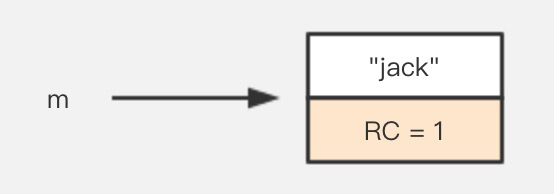
To optimize its memory footprint, a live thread should only retain access to an object as long as the object is needed for its execution. The program can make objects become eligible for garbage collection as early as possible by removing all references to the object when it is no longer needed.

Objects that are created and accessed by local references in a method are eligible for garbage collection when the method terminates, unless reference values to these objects are exported out of the method. This can occur if a reference value is returned from the method, passed as argument to another method that records the reference, or thrown as an exception. However, a method need not always leave objects to be garbage collected after its termination. It can facilitate garbage collection by taking suitable action, for example, by nulling references.

The [Reference Counting Algorithm](https://blogs.msdn.microsoft.com/abhinaba/2009/01/27/back-to-basics-reference-counting-garbage-collection/?spm=a2c41.13644014.0.0) allocates a field in the object header to store the reference count of the object. If this object is referenced by another object, its reference count increments by one. If the reference to this object is deleted, the reference count decrements by one. When the reference count of this object drops to zero, the object will be garbage-collected.

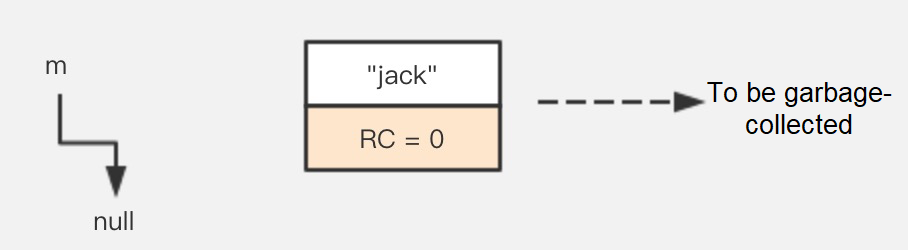
String m = new String("jack");

let’s create a string in which “jack” is referenced by m.



Then, set m to null. The reference count of “jack” is zero. In the Reference Counting algorithm, the memory for “jack” is to be reclaimed.

m = null;



The Reference Counting Algorithm performs GC in the execution of the program. This algorithm does not trigger Stop-The-World events. Stop-The-World means that the execution of the program is suspended for GC till all objects in the heap are processed. Therefore, this algorithm does not strictly follow the Stop-The-World GC mechanism.

It looks pretty applicable to GC. However, we know that GC on the Java virtual machine (JVM) follows the Stop-The-World mechanism. Why did we give up the Reference Counting algorithm? Let’s look at the following example:

public class ReferenceCountingGC { public Object instance;

public ReferenceCountingGC(String name){}  
}

public static void testGC()

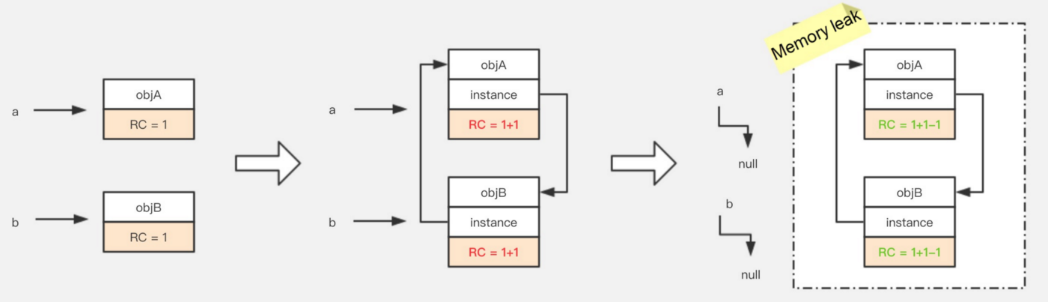
{

ReferenceCountingGC a = new ReferenceCountingGC("objA");  
 ReferenceCountingGC b = new ReferenceCountingGC("objB");

a.instance = b;  
 b.instance = a;

a = null;  
 b = null;  
}

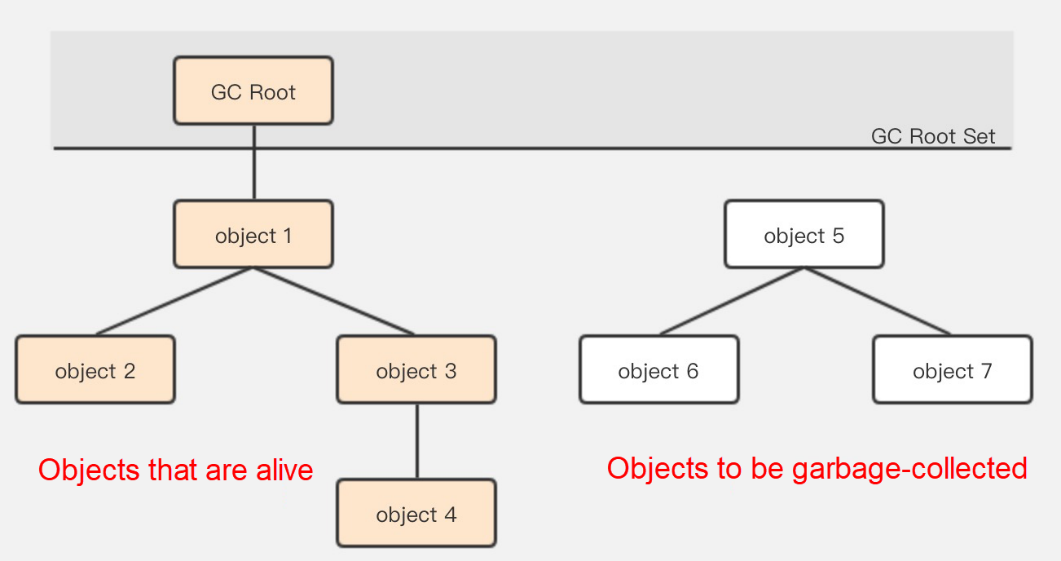
We first define two objects, then make mutual reference to the objects, and lastly set references for each object to null.



We can see that both objects can no longer be accessed. However, they are referenced by each other, and thus their reference count will never be zero. Consequently, the GC collector will never be notified to garbage collect them by using the Reference Counting algorithm.

**Reachability Analysis Algorithm**

The basic idea of the Reachability Analysis Algorithm is to start from GC roots. GC traverses the whole object graph in the memory, starting from these roots and following references from the roots to other objects. The path is called the reference chain. If an object has no reference chain to the GC roots, that is the object cannot be reached from the GC roots, the object is unavailable.

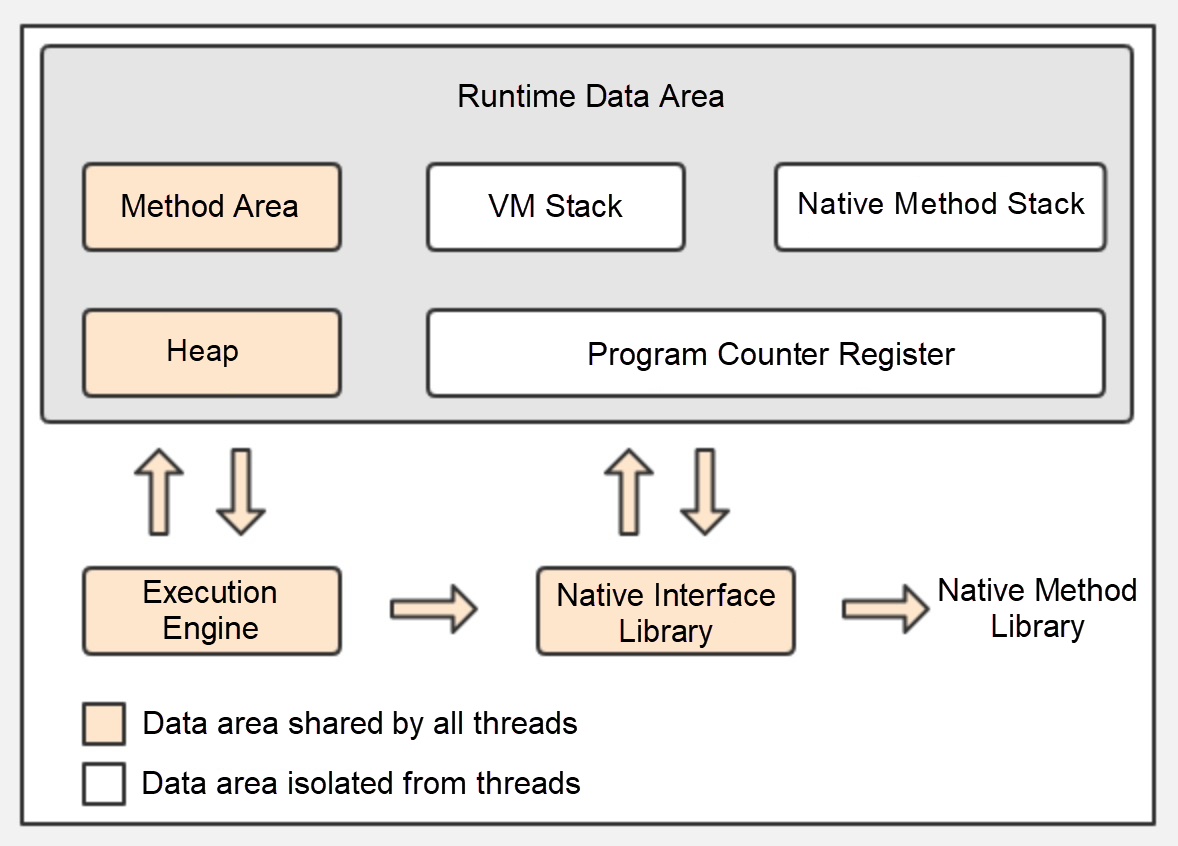


The Reachability Analysis algorithm successfully solves the problem of cyclic references in the Reference Counting algorithm. As long as an object cannot establish a direct or indirect connection with the GC roots, the system determines that the object is to be garbage-collected. Then, another question arises. What are GC roots?

**Java Memory Space**

In Java, GC roots can be four types of objects:

* Objects referenced in the virtual machine (VM) stack, that is the local variable table in the stack frame
* Objects referenced by class static attributes in the method area
* Objects referenced by constants in the method area
* Objects referenced by JNI (the Native method) in the native method stack



Objects referenced in the VM stack, that is the local variable table in the stack frame

In this case, s is the GC root. When s is set to null, the localParameter object has its reference chain with the GC root broken, and the object will be garbage-collected.

public class StackLocalParameter {  
 public StackLocalParameter(String name){}  
}public static void testGC(){  
 StackLocalParameter s = new StackLocalParameter("localParameter");  
 s = null;  
}

Objects referenced by class static attributes in the method area

When s is the GC root and s is set to null, after GC, the properties object to which s points is garbage-collected because it cannot establish a connection with the GC root. As a class static attribute, m is also a GC root. The parameter object is still connected to the GC root, so the parameter object will not be garbage-collected in this case.

public class MethodAreaStaicProperties {  
 public static MethodAreaStaicProperties m;  
 public MethodAreaStaicProperties(String name){}  
}public static void testGC(){  
 MethodAreaStaicProperties s = new MethodAreaStaicProperties("properties");  
 s.m = new MethodAreaStaicProperties("parameter");  
 s = null;  
}

Objects referenced by constants in the method area

As a constant reference in the method area, m is also the GC root. After s is set to null, the final object will not be garbage-collected though it has no reference chain with the GC root.

public class MethodAreaStaicProperties {  
 public static final MethodAreaStaicProperties m = MethodAreaStaicProperties("final");  
 public MethodAreaStaicProperties(String name){}  
}public static void testGC(){  
 MethodAreaStaicProperties s = new MethodAreaStaicProperties("staticProperties");  
 s = null;  
}

## A1.3 Comparison of the manual and the automated approaches

Garbage collection has disadvantages like consuming additional memory (RAM) to run those algorithms and this has a major performance impact. Further, the objects aren’t garbage collected at the very instant.

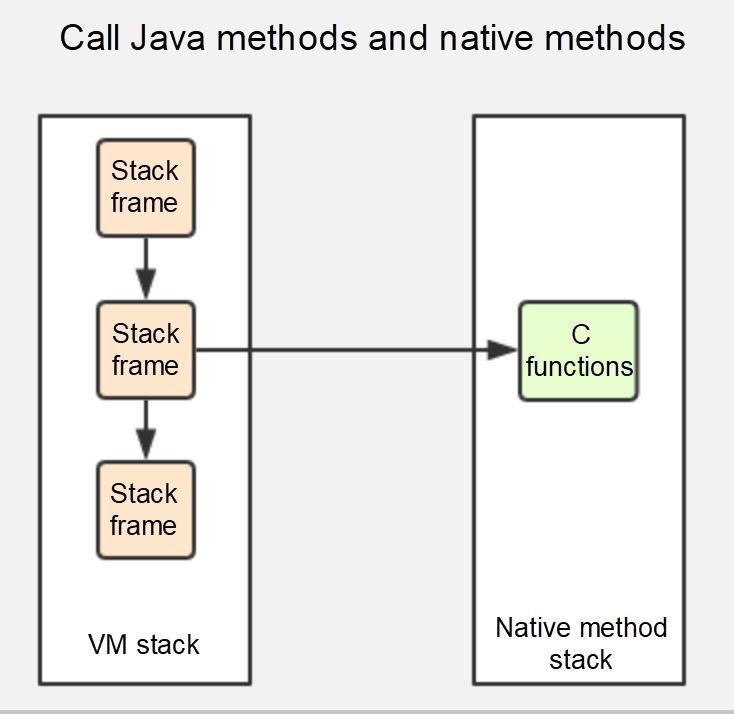
It takes its own time it has a performance impact as well.

A [peer-reviewed paper](http://people.cs.umass.edu/~emery/pubs/gcvsmalloc.pdf) came to the conclusion that GC needs five times the memory to perform as fast as explicit memory management.

 If the memory is compromised, it leads to possible stalls in program execution.

Objects referenced in the native method stack.A native interface always uses a native method stack. If the native method interface is implemented by using the C connection model, its native method stack is the C stack.

When a thread calls the Java method, the VM creates a new stack frame and puts it in the Java stack. However, when it calls the native method, the VM keeps the Java stack unchanged and no longer puts new frames in the thread’s Java stack. Instead, the VM dynamically connects to and directly calls the specified native method.



## A1.4 The stance taken with justification

Garbage collection has disadvantages like consuming additional memory (RAM) to run those algorithms and this has a major performance impact. Further, the objects aren’t garbage collected at the very instant.

It takes its own time it has a performance impact as well.

A [peer-reviewed paper](http://people.cs.umass.edu/~emery/pubs/gcvsmalloc.pdf) came to the conclusion that GC needs five times the memory to perform as fast as explicit memory management.

 If the memory is compromised, it leads to possible stalls in program execution.

# **Question No. 2**

**Solution to Question No. 2:**

## B1.1 Introduction to an Airline reservation system

This is a simple ticket booking system developed in java. This system enables user to book ticket and generate a unique PNR for the ticket while also generating a bill corresponding to the ticket.

In the bill it also includes the excess baggage fee as per given in the question.

This system gives users a choice to book ticket, view PNR status, cancel ticket, and see baggage limit. This program does not use any database; hence the data is stored in a List (PNRbook).

While booking a ticket, system asks user to input class from option like Economy, Business, First Class.

The OOP concepts like multiple inheritance, polymorphism has been implemented.

Note: This program has many redundancies. This program is not implemented in an optimized way; hence the program can be further optimized.

## B1.2 Design of the Airline reservation system

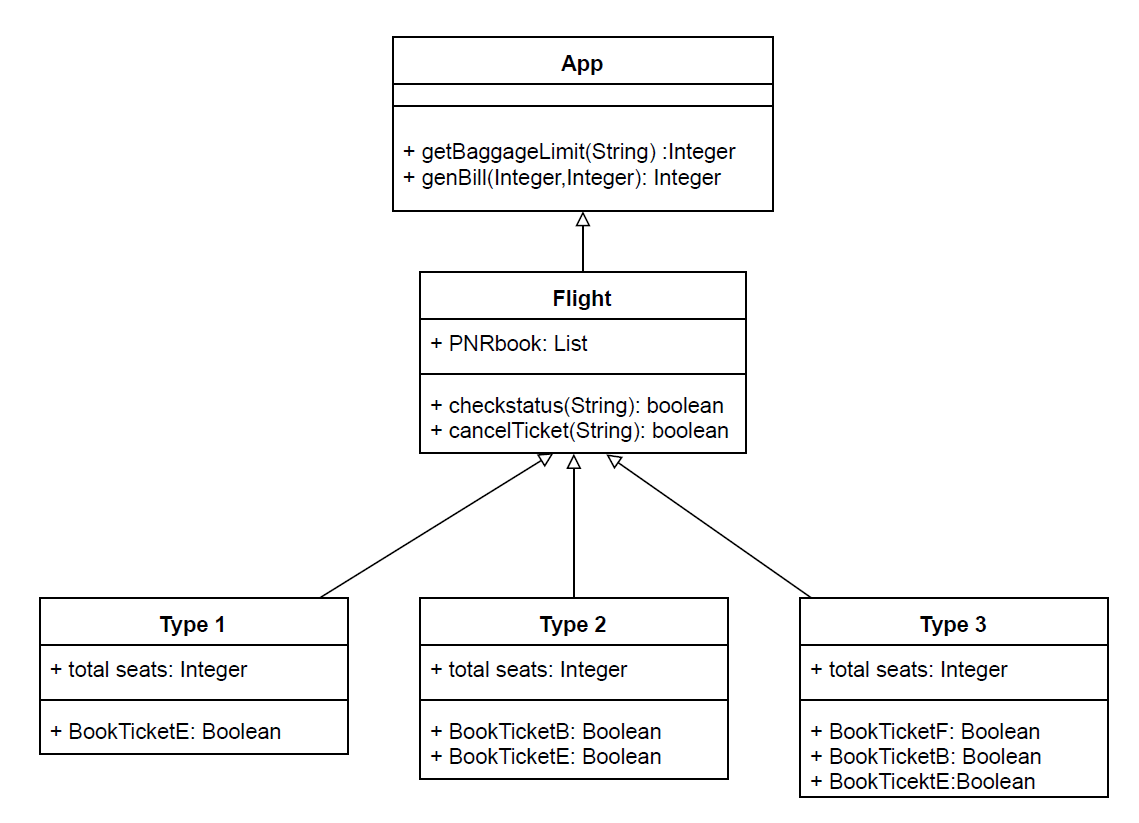


Figure 1 Class of my implementation of Airline Reservation System

## B1.3 Implementation of the Airline reservation system



Figure 2 Flight Class



Figure 3 Type 1 Class which extends Flight Class



Figure 4 Type 2 Class which extends Flight Class



Figure 5 Type 3 Class which extends Flight Class





Figure 6 App Class - Driver Class

Output Screenshot:

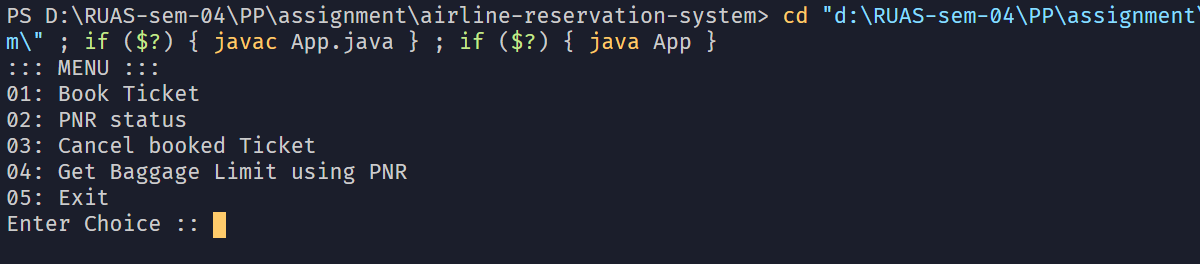


Figure 7 Menu

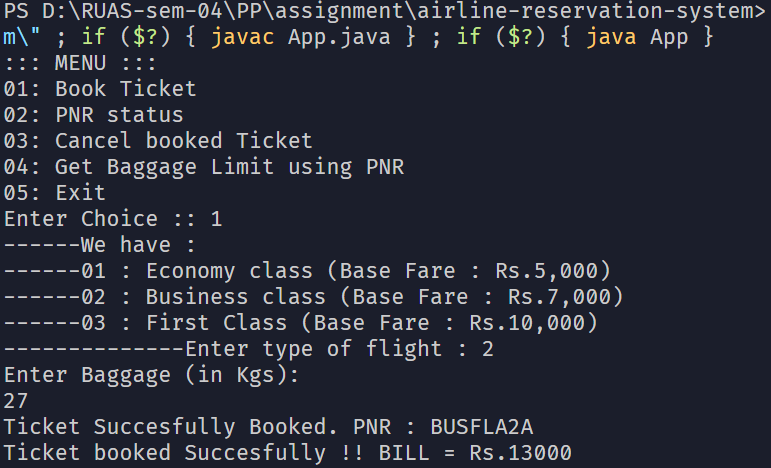


Figure 8 Booking Ticket for Business Class

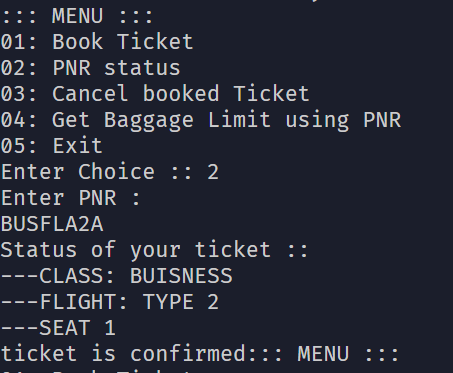


Figure 9 Checking status of PNR

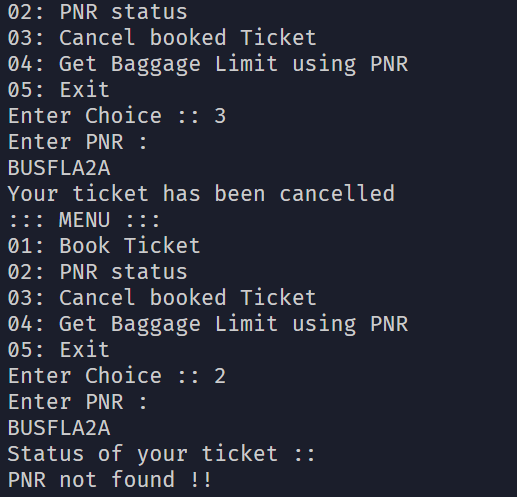


Figure 10 Cancelling ticket – deleting PNR from List

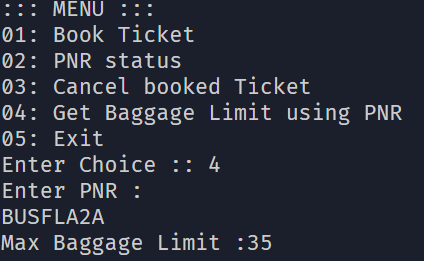


Figure 11 getting Baggage limit

# Bibliography

* <http://etutorials.org/cert/java+certification/Chapter+8.+Object+Lifetime/8.1+Garbage+Collection/>
* <https://en.wikipedia.org/wiki/Manual_memory_management>
* <https://www.programiz.com/c-programming/c-dynamic-memory-allocation>
* <https://www.alibabacloud.com/blog/how-does-garbage-collection-work-in-java_595387>

The source code of the above implemented of Airline Reservation System can be found at :

<https://github.com/subhendu17620/RUAS-sem-04/tree/master/PP/assignment/airline-reservation-system>