Assignment 2

1.

(a) What is the output of Line 1?
Answer: A.p
A.q

(b) What is the output of Line 2?
Answer: B.p
A.q

(c) What is the output of Line 3?
Answer: B.p
C.q

(d) What is the output of Line 4?
Answer: B.p
A.q

(e) What is the output of Line 5?
Answer: B.p
C.q

(f) What is the output of Line 6?
Answer: B.p
C.q

Imagine we design a new language called JavaStatic. It has the identical syntax as Java. Its semantics is also identical to Java, except that all message dispatch is static, NOT dynamic.

(g) If the code above were JavaStatic, what would be the output of Line 1?
Answer: A.p
A.q

(h) If the code above were JavaStatic, what would be the output of Line 2?
Answer: A.p
A.q

(i) If the code above were JavaStatic, what would be the output of Line 3?
Answer: B.p
C.q

(j) If the code above were JavaStatic, what would be the output of Line 4?
Answer: A.p
A.q

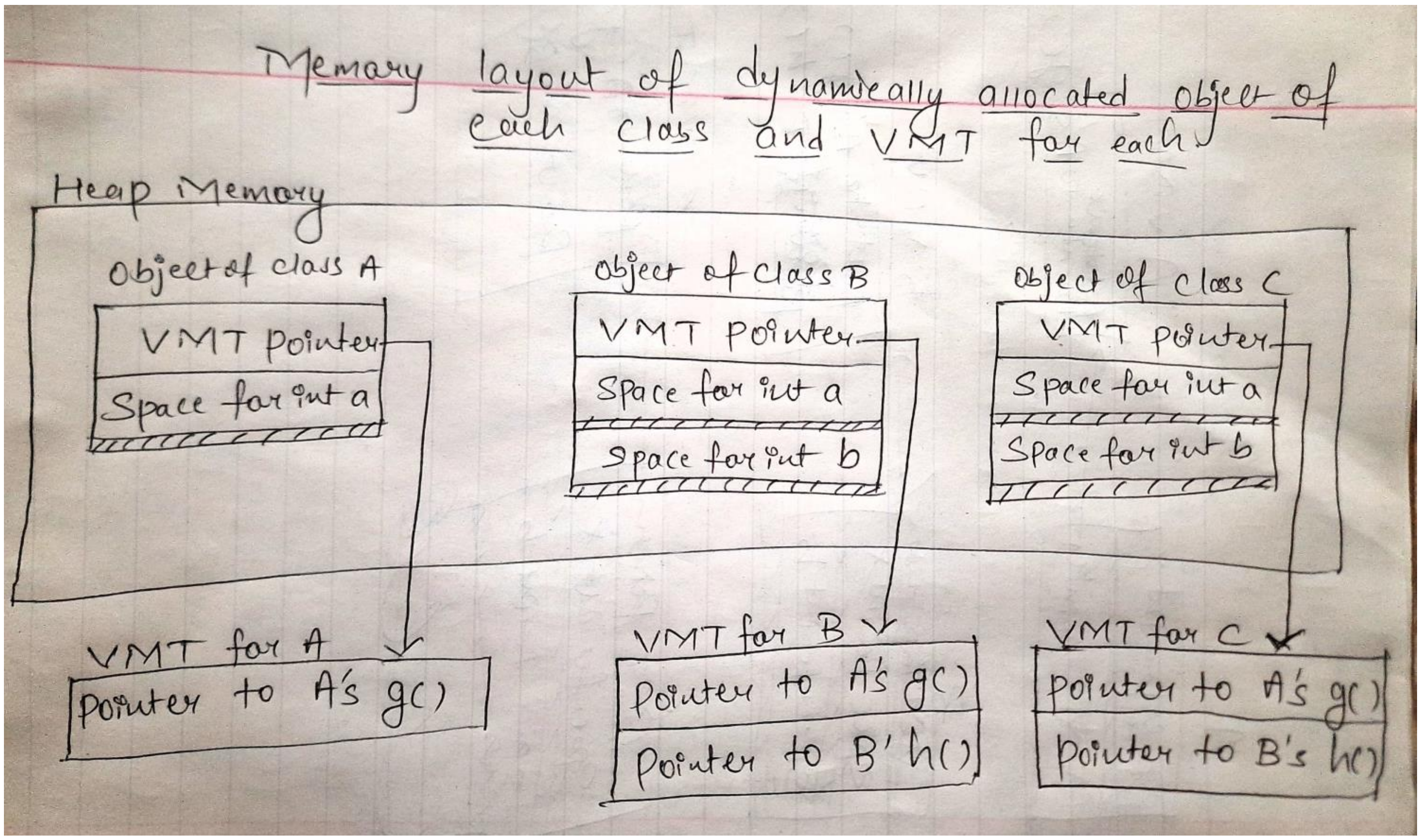
(k) If the code above were JavaStatic, what would be the output of Line 5?
Answer: A.p
A.q

(l) If the code above were JavaStatic, what would be the output of Line 6?
Answer: A.p
A.q

2. [24 points] Given the following C++ declarations, draw the VMT of each class and the layout of memory for a dynamically allocated object of **each class**.

Answer: Below is the the VMT of each class and the layout of memory for a dynamically allocated object of each class.

Note: The slashed area in the memory represents padding required if any.



3. [10 points] Many people think the Java "interface" mechanism is an encapsulation mechanism. Their arguments are as follows. Suppose we have an interface A, being implemented by class B. If programmers consistently use type A every time an instance of B is referred to, then any method, say m, defined in B but not specified in A is hidden and cannot be accessed by any code holding a reference of type A. There however is a fatal flaw in this argument. Explain how a piece of code referring to an instance of B (with type A) can in fact access m.

Answer: A piece of code referring to an instance of B with type A can in fact access m using the type casting functionality. We can type cast the instance to B and call the method m. In this way we can access m. Below is an example:

```
interface A
{
   final int var1 = 10;
   void a_method();
}

public class B implements A
{
public void a_method(){
   System.out.print("a_method");
}

public void m(){
   System.out.print("m");
}

 public static void main(String[] args)
 {
 A a = new B();
 ((B)a).m();
}

}
```

4. **[30 points]** Read paper "**Streamflex: high-throughput stream programming in Java**" and answer the following questions. The paper can be downloaded at: http://janvitek.org/pubs/oopsla07.pdf.

(a) [20 points] In Fig 3, several kinds of object references are explicitly disallowed. Now answer the following questions:

(1) Why do we have two separate memory areas, one for Java objects (the “HeapMemory” in the figure), and the other for stream objects (the “Stable” and “Transient” and “Capsule” objects)?

Answer: we have two separate memory areas, one for Java objects and other for stream objects because java object memory has garbage collection functionality which can cause unexpected pause and interruptions. STREAMFLEX supports the principal of ‘keep the data moving’ and high throughput which will then be compromised. Running a stream processing application on a standard JVM would uncover a number of drawbacks of Java for applications with any quality of service requirements. An example given in the paper, allocates a StringBuffer and a String at each release. Eventually filling up the heap, triggering garbage collection and blocking the filter for hundreds of milliseconds.

Another issue is the default Java scheduler may decide to preempt a filter at any point of time in favor of a plain Java thread. STREAMFLEX ensures low-latency by executing filters in a partition of the JVM’s memory which is outside of the control of the garbage collector. This allows the STREAMFLEX scheduler to safely preempt any Java thread, including the garbage collector. Activities can thus run without fear of being blocked by the garbage collector.

Another danger is potential for priority inversion. To prevent synchronization hazards, such as an activity blocking on a lock held by a Java thread which in turn can block for the GC, filters are isolated from the Java heap. Non-blocking synchronization in the form of software transactional memory is used when Java threads need to communicate with filters.

A Java virtual machine implementation is the source of many potential interferences due to global data structures, just-in-time compilation and, of course, garbage collection. According to paper they have performed empirical measurements of the performance of standard and real-time garbage collectors. Our stop-the-world collector introduced pauses of 114 milliseconds. Using a real-time collector pause time went down to around 1 milliseconds at the expense of application throughput. In both cases, the pauses and performance overheads were too severe for some of our target applications.

Due to the above reasons we have separate memory areas for Java objects and Stream objects.

(2) Why would it be a bad idea if a capsule object holds a reference to a transient object?

Answer: Transient objects live as long as the activity. Capsules are data objects used in messages and are managed by the STREAMFLEX runtime engine. Capsule objects are used by filters to communicate with other filters. It would be a bad idea if capsule object holds a reference to a transient object because transient object memory is deallocated after the particular activity but capsule reference could still hold the memory location after the activity and this could lead to dangling pointers which is a costly disadvantage.

Capsules cannot be allocated in the transient memory of a filter as they would be deallocated as soon as the filter's **work**() method returns.

(3) Why would it be a bad idea if an object in the Java HeapMemory holds a reference to a Capsule object?

Answer: It would be a bad idea if an object in the java heap memory holds reference to a capsule object because heap memory is managed by garbage collector and capsule objects are managed by STREAMFLEX runtime manager. If the STREAMFLEX runtime manager dellocates the memory of capsule object after its use or if channel overflows, then object in java heap memory can still hold reference that. This could lead to errors and unexpected program behavior(can lead to dangling pointers).

Capsules are designed with one key requirement: allow for **zero-copy** communication in a linear filter pipeline. If object in heap memory holds reference to capsule objects, this requirement could be compromised.

Also, capsule objects used in Stream threads can interfere with java threads and java heap memory objects could also use non-primitive datatypes which violates STREAMFLEX capsule object restrictions. They are restricted to having fields of primitive types or of primitive arrays types.

(4) Why would it be a bad idea if a stable object holds a reference to a capsule object?

Answer: Capsule is an object that is manipulated in a linear fashion. At any given time the type system enforce that both of the following holds: (1) there is at most a single reference to the capsule from data channels, and (2) there are no references to a capsule from stable memory. These invariants permit zero-copy common uses of capsules.

It would be a bad idea if stable object holds reference to capsule object because it will eliminate the Zero copy communication requirement.

Also, both capsule and stable objects have different lifetimes. Capsule objects are sued for message passing between filters and filters are stable objects itself. Could lead to invalid references and cause error.

(b) [5 points] Please give a counterexample to show why rule R1 in Section 4.2 is necessary. (In other words, this example would show that if R1 is compromised, something property sought after by Streamflex cannot be guaranteed.)

Answer: R1:
Line1:The type of arguments to public and protected methods of any subclass of Filter can be primitive types as well as arrays of primitive types.
Line2:Returns types of these methods are limited to primitive types.
Line3:The type of public and protected fields of any subclass of Filter are limited to primitive types.

The advantage of using primitives is that one does not need to worry about memory management or aliasing for data transferred between filters. If we allow non-primitive types then we might face problem from aliasing and memory management could be impractical . Below is an example:

```
public class Sedan_filter extends Filter {
private Channel<example_capsule> in, out;
public Vehicle v; //assuming stable object
public Design vd = Vehicle.getSedanDesign(); //Example violating line3 of rule

public Vehicle select_sedans(Vehicle v) {  // Example violating line1 and 2 of rule

        if(v.getDesign().equals(vd)) {
                return v;
        }
        return null;
}

public void work() {
v = Vehicle.getVehicle(in)
if(select_sedan(v)!=null)
        out.put(1);
else
        out.put(0);}

}
```

Explaination: Above is a simple filter example that puts 1 on the channel in case of sedan vehicles and 0 in other cases. The filter violates rule R1 by using public field vd of a non-primitive type Design. Also, public method select_sedans returns and takes argument of non-primitive type Vehicle. Using this filter could pose problems as another filter could access v and maintain reference to it which will result in aliasing. Similarly public method select_sedan will also be exposed outside of clients of the filter.

(c) **[5 points]** Explain why the “@atomic” declaration must be present for the “write” method in Fig 15.

Answer: We implement channels using the @atomic annotation for methods that would otherwise be synchronized. According to paper, Figure 15 shows PacketReader. This filter creates capsules representing network packets from a raw stream of bytes. For our experiments we simulate the network with the Synthesizer class. The synthesizer runs as a **plain Java thread**, and feeds the reader with a raw stream of bytes to be analyzed. Communication between the synthesizer and the PacketReader is done by calling **PacketReader.write().** This method takes a reference to a buffer of data allocated in plain Java and parses it to create packets. **write() is annotated @atomic to ensure that a filter can safely preempt the synthesizer at any time.**

If write() was not annotated with @atomic, then preempting the synthesizer by the filter could lead to race conditions causing possible data corruption because synthesizer runs a plain java thread instead of STREAMFLEX thread.