**SLASSCOM Software Engineering Bootcamp Final Assignment**

**Question 2**

**a.**

* Upcasting refers to the process of casting an object of a subclass to its superclass. In other words, it involves converting a derived class object to a base class object. This is done implicitly, and it is safe because the derived class object can be treated as if it were a base class object.
* Downcasting, on the other hand, refers to the process of casting an object of a superclass to its subclass. In other words, it involves converting a base class object to a derived class object. This is done explicitly, and it is not always safe because the base class object may not actually be an instance of the derived class.

Diagram

Description automatically generated

public class Animal {  
 public void eat() {  
 System.*out*.println("Animal is eating.");  
 }  
}

public class Deer extends Animal {  
 @Override  
 public void eat() {  
 System.*out*.println("Deer is eating.");  
 }  
  
 public void run() {  
 System.*out*.println("Deer is running.");  
 }  
}

Example 1:

// Upcasting  
Animal animal = new Deer(); // Deer object is upcasted to Animal type  
animal.eat(); // Output: Deer is eating. (Dynamic method dispatch)

Upcasting is when an object of a subclass type is assigned to a variable of the superclass type. In the example, we create a Deer object and assign it to an Animal variable, which is an upcast. The object is still a Deer object, but now it can only access the Animal class's methods and properties.

Dynamic method dispatch is used when the eat() method is called on the animal variable. The method to be executed is determined at runtime based on the actual object assigned to the animal variable. Since animal points to a Deer object, the Deer class's eat() method is called, even though the reference is of type Animal.

Example 2:

// Downcasting  
Deer deer = (Deer) animal; // Animal object is downcasted to Deer type  
deer.eat(); // Output: Deer is eating.  
deer.run(); // Output: Deer is running.

Downcasting is the opposite of upcasting, which is when an object of a superclass type is assigned to a variable of the subclass type. In the example, we downcast the Animal object to a Deer object using the (Deer) cast operator. This allows us to access the Deer class's specific methods, such as the run() method, which is not available in the Animal class.

**b.**

**HashSet** 🡪 HashSet is an implementation of the Set interface that uses a hash table for storing and retrieving objects. It is useful when we need to store unique elements and do not need to maintain any specific order of the elements. Some applications of HashSet include:

1. Removing duplicates: Since HashSet does not allow duplicate elements, we can use it to remove duplicates from a collection of objects. For example, we can use a HashSet to remove duplicate words from a list of words.
2. Fast lookup: HashSet provides fast lookup of elements using their hash codes. This makes it useful when we need to quickly determine whether an element is present in a collection. For example, we can use a HashSet to store a list of usernames and quickly check if a given username is already taken.

Here's an example of using HashSet to remove duplicates from an array of integers:

import java.util.HashSet;  
  
public class Main {  
 public static void main(String[] args) {  
 int[] arr = {1, 2, 3, 2, 1, 4};  
  
 HashSet<Integer> set = new HashSet<>();  
 for (int i = 0; i < arr.length; i++) {  
 set.add(arr[i]);  
 }  
  
 System.*out*.println(set); // Output: [1, 2, 3, 4]  
 }  
}

In the preceding code, duplicate integers are created in an array, which is then iterated through to add each element to the hash set. The set only contains unique elements because HashSet disallow duplication.

**HashTable** 🡪 A HashTable is an implementation of the Map interface that uses a hash table to store key-value pairs. It does not allow duplicate keys and provides constant time performance for basic operations such as adding, removing, and checking for the presence of keys. Some applications of HashTable include:

1. Caching: HashTable can be used for caching frequently accessed data. We can store the data in a HashTable with the keys being the input arguments to a function and the values being the corresponding output of the function.
2. Indexing: HashTable can be used for indexing large datasets. We can create a HashTable with keys being unique identifiers and values being the corresponding data objects.

Here's an example of using HashTable to store and retrieve key-value pairs:

import java.util.Hashtable;  
  
public class Main {  
 public static void main(String[] args) {  
 Hashtable<String, Integer> ht = new Hashtable<>();  
  
 ht.put("Alice", 25);  
 ht.put("Bob", 30);  
 ht.put("Charlie", 35);  
  
 System.*out*.println(ht.get("Bob")); // Output: 30  
  
 ht.remove("Charlie");  
 System.*out*.println(ht); // Output: {Bob=30, Alice=25}  
 }  
}

**c.**

**Stack** 🡪 The stack is a LIFO (Last In First Out) data structure that is used to manage the execution of a program. It stores variables and method calls during the execution of a method, and each method call creates a new stack frame (also known as an activation record) that stores the method's local variables, parameters, and return address. When a method completes, its stack frame is removed from the stack.

The stack is a LIFO (Last In First Out) data structure that is used to manage the execution of a program. It stores variables and method calls during the execution of a method, and each method call creates a new stack frame (also known as an activation record) that stores the method's local variables, parameters, and return address. When a method completes, its stack frame is removed from the stack.

Here's a diagram that shows the basic structure of the stack:

A picture containing text

Description automatically generated

In the diagram above, we have two methods (Method 1 and Method 2) that are being executed. Each method has its own stack frame, which contains the method's local variables, parameters, and return address. The stack grows downward from high memory addresses to low memory addresses, and each new stack frame is pushed onto the top of the stack.

Heap 🡪 The heap is a dynamic memory area that is used to store objects and data that are created at runtime. It is a large, flexible data structure that can grow or shrink as needed during the execution of a program. Objects in the heap can be accessed by multiple threads, and they are allocated and deallocated by the Java Virtual Machine (JVM) using a process known as garbage collection.

Here's a diagram that shows the basic structure of the heap:

Table

Description automatically generated

In the diagram above, we have three objects (Object 1, Object 2, and Object 3) that are stored in the heap. Each object occupies a certain amount of memory, which is allocated and deallocated by the JVM as needed. The heap grows upward from low memory addresses to high memory addresses, and objects are stored in a fragmented fashion as they are allocated and deallocated.

In summary, the stack and heap are two distinct memory areas used by Java to manage the runtime behavior of a program. The stack is used to manage the execution of a program and stores method calls and local variables, while the heap is used to store objects and data that are created at runtime.