**SLASSCOM Bootcamp**

**Question 2**

**a.**

* Casting an item from a subclass to its superclass is known as upcasting. In other words, it entails transforming an object of a derived class to an object of a base class. Because the derived class object can be handled as if it were a base class object, this is done implicitly and is safe.

On the other hand, downcasting describes the action of casting an object from a superclass to its subclass. In other words, it entails changing an item from a base class to a derived class. Because the base class object might not truly be an instance of the derived class, doing this directly is not always safe.

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 the process of assigning an object of a subclass type to a variable of a superclass type. In the illustration, a Deer object is created and assigned to an upcast Animal variable. Although it can still only access the Animal class's methods and properties, the object is still a Deer object.

When the eat() method is applied on the animal variable, dynamic method dispatch is employed. The method to be executed is determined at runtime based on the actual object assigned to the animal variable. Despite the fact that the reference is of type Animal, the eat() function of the Deer class is called because animal points to a Deer object.

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.

Upcasting, in which an object of a superclass type is assigned to a variable of a subclass type, is the opposite of downcasting. With the (Deer) cast operator, we downcast the Animal object in the example to a Deer object. This enables us to access methods unique to the Deer class, such as the run() method, which is absent from the Animal class.

**b.**

**HashSet** 🡪 A hash table is used by HashSet, an implementation of the Set interface, to store and retrieve objects. It is helpful when we need to store distinct elements without having to keep the pieces in a particular sequence. HashSet has a variety of uses, such as:

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.

Quick lookup: Using the hash codes of the items, HashSet offers quick lookup. This makes it useful when we need to quickly determine whether an element is present in a collection. A HashSet can be used, for instance, to keep a list of usernames and rapidly determine whether a particular username is already in use.

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.

The key differences between the stack and the heap can be summarized as follows:

1. Usage: The stack is used for temporary data storage, while the heap is used for dynamic memory allocation.
2. Organization: The stack is organized as a LIFO data structure, while the heap is organized as a free list of memory blocks.
3. Allocation: Memory is allocated automatically on the stack, while memory is allocated manually on the heap using functions such as malloc() and new.
4. Deallocation: Memory is deallocated automatically on the stack, while memory must be manually deallocated on the heap using functions such as free() and delete.

In summary, the stack and the heap are both important areas of memory used in computer systems, but they are used for different purposes and have different characteristics. The stack is used for temporary data storage, while the heap is used for dynamic memory allocation. The stack is organized as a LIFO data structure, while the heap is organized as a free list of memory blocks.