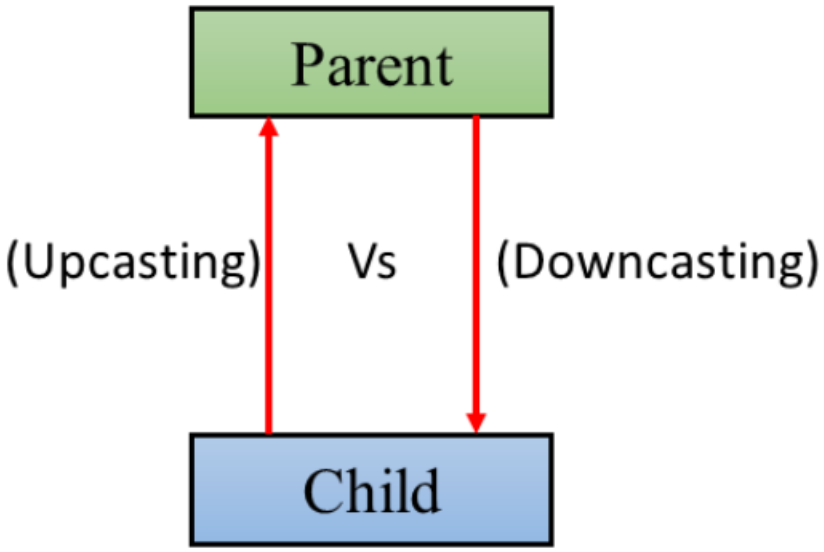
Software Engineering Bootcamp Assignment

Question 03

a) Upcasting and down casting are fundamental concepts in object-oriented programming that allow us to work with objects of different types. In simple words, upcasting refers to casting an object to its parent type, while down casting refers to casting an object to its child type. Let's explore these concepts further with some code examples in Java.



Up Casting

Upcasting (Generalization or Widening) allows us to treat an object of a subclass as if it were an object of its superclass. This is also known as generalization or widening. Let's consider a simple example:

Example:

```
class Animal {
   public void eat() {
      System.out.println("Eating...");
   }
}

class Dog extends Animal {
```

```
    public void bark() {
        System.out.println("Barking...");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal animal = dog; // Upcasting
        animal.eat(); // Calling superclass method
    }
}
```

In the above code, we have two classes, Animal and Dog. Dog is a subclass of Animal. We create an object of the Dog class and then upcast it to its superclass Animal. After that, we call the eat() method using the upcasted object. As expected, the output of the program will be Eating....

Down Casting

Down casting is the opposite of upcasting. It allows us to cast an object of a superclass to its subclass. This is also known as specialization or narrowing. Let's consider another example:

Example:

```
class Animal {
    public void eat() {
        System.out.println("Eating...");
    }
}

class Dog extends Animal {
    public void bark() {
```

```
        System.out.println("Barking...");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        Dog dog = (Dog) animal; // Downcasting
        dog.bark(); // Calling subclass method
    }
}
```

In the above code, we first create an object of the Dog class and upcast it to its superclass Animal. Then, we downcast the Animal object back to the Dog class using explicit casting. Finally, we call the bark() method using the down casted object. As expected, the output of the program will be Barking....

To sum up, upcasting and down casting are essential concepts in Java that allow us to work with objects of different types. Upcasting is casting an object to its superclass, while down casting is casting an object to its subclass. In general, upcasting is always allowed, but down casting involves a type of check and can throw a **ClassCastException**.

b) Both HashSet and Hashtable are Java collection classes that can be used to store and manage groups of related objects. However, they have different features and applications.

HashSet

HashSet is a collection class that implements the Set interface. It stores a collection of unique objects, meaning that it cannot contain duplicate elements. The objects in a HashSet are not ordered, and they can be inserted, deleted, and searched in constant time, making it suitable for applications that require efficient lookup operations.

Here are some suitable examples where HashSet can be used:

Removing duplicate elements: Suppose you have a list of names, and you want to remove duplicate names. You can use a HashSet to store the unique names and then iterate over the HashSet to get the unique names.

Example:

```
List<String> names = Arrays.asList("John", "Mary", "David", "Mary", "John");
Set<String> uniqueNames = new HashSet<>(names); System.out.println(uniqueNames);
// Output: [John, Mary, David]
```

Storing a collection of items with fast lookup: Suppose you have a large collection of items, and you need to quickly check if an item exists in the collection. You can use a HashSet to store the items and then use the contains() method to check if an item exists.

```
Set<String> items = new HashSet<>();
items.add("item1"); items.add("item2");
 items.add("item3");
if (items.contains("item2")){
System.out.println("Item2 exists"); }
```

Hashtable

Hashtable is a collection class that implements the Map interface. It stores key-value pairs, where each key is associated with a value. The keys in a Hashtable are unique, and they are used to access the corresponding values. The objects in a Hashtable are not ordered, and they can be inserted, deleted, and searched in constant time.

Here are some suitable examples where Hashtable can be used:

Caching data: Suppose you have a time-consuming operation that generates data, and you want to cache the data for faster access. You can use a Hashtable to store the data, where the input to the operation is the key, and the output is the value.

Example:

```
Hashtable<Integer, String> dataCache = new Hashtable<>();
public String getData(int input) {
if (dataCache.containsKey(input)) {
 return dataCache.get(input); }
```

```
else {
String result = // Time-consuming operation to generate data dataCache.put(input, result); return result; }
}
```

Storing configuration settings: Suppose you have a Java application that reads configuration settings from a file, and you want to store the settings in memory for faster access. You can use a Hashtable to store the settings, where the setting name is the key, and the setting value is the value.

Example:

```
Hashtable<String, String> configSettings = new Hashtable<>();
// Read configuration settings from file
configSettings.put("username", "john");
configSettings.put("password", "secret");
configSettings.put("host", "localhost");
```

In summary, HashSet and Hashtable are both useful Java collection classes with different features and applications. HashSet is suitable for storing a collection of unique objects with fast lookup, while Hashtable is suitable for storing key-value pairs with fast access to the values.

c) In Java, memory is divided into two parts: the Stack and the Heap. The Stack is used for storing local variables and method calls, while the Heap is used for storing objects and arrays.

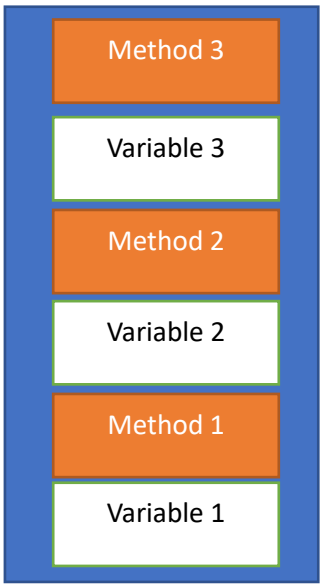
Stack

The Stack is a region of memory that is used for storing local variables and method calls. Each thread in a Java program has its own Stack, and the Stack is divided into frames, where each frame corresponds to a method call. When a method is called, a new frame is pushed onto the Stack, and when the method returns, the frame is popped off the Stack.

Local variables are stored on the Stack, and they are automatically freed when the method returns. The Stack is a LIFO (Last-In-First-Out) data structure, meaning that the last item pushed onto the Stack is the first item to be popped off.

Here is a diagram that illustrates how the Stack works:

In this example, there are three methods, and each method has its own set of local variables. When Method 1 is called, a new frame is pushed onto the Stack, and the local variables for Method 1 are stored in the frame. When Method 1 calls Method 2, a new frame is pushed onto the Stack, and the local variables for Method 2 are stored in the new frame. When Method 2 returns, its frame is popped off the Stack, and control returns to Method 1. When Method 1 returns, its frame is popped off the Stack, and control returns to the caller.



Heap

The Heap is a region of memory that is used for storing objects and arrays. Objects are allocated on the Heap using the new keyword, and they are not automatically freed when they are no longer needed. Instead, they are garbage collected when they are no longer reachable by any running code.

Here is a diagram that illustrates how the Heap works:

In this example, there are three objects stored on the Heap. Each object has its own set of instance variables, which are also stored on the Heap. When an object is allocated, it is given a reference, which is a pointer to the object's memory location on the Heap. The reference is stored on the Stack or in another object's instance variable.

In summary, the Stack and Heap are both regions of memory in Java that serve different purposes. The Stack is used for storing local variables and method calls, and it is automatically managed by the Java Virtual Machine. The Heap is used for storing objects and arrays, and it is managed by the garbage collector.