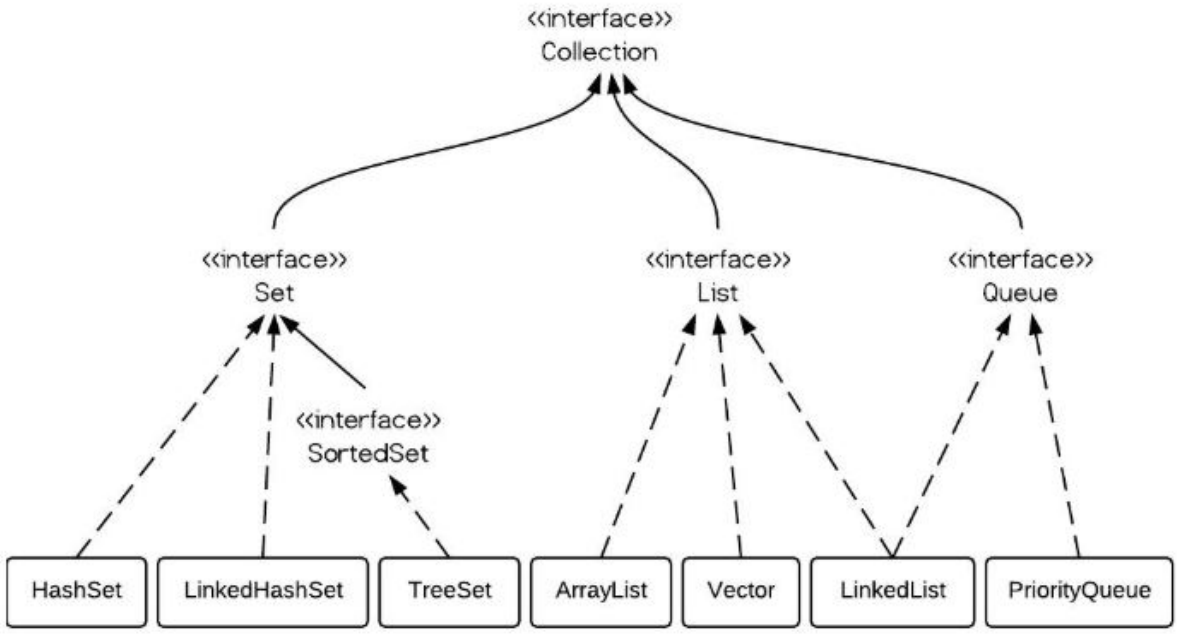
Personally, I use **-cp** or **-classpath** option with **java** command to run my program. This way you always know which JARs are included in your classpath.

@ difference between PATH and CLASSPATH

PATH is used locate system executable, commands or `.exe`, `.dll` files (in Windows) and `.so` files (in Linux). It is also used to locate native libraries used by your Java program.
While, CLASSPATH is used to locate the class file or JAR files. It's Java class loader who looked into CLASSPATH for loading classes.

@ List Overview

List, as its name indicates, is an ordered sequence of elements. When we talk about List, it is a good idea to compare it with Set which is a set of unique and unordered elements. The following is the class hierarchy diagram of Collection. From the hierarchy diagram you can get a general idea of Java Collections.



@ ArrayList vs. LinkedList vs. Vector

From the hierarchy diagram, they all implement List interface. They are very similar to use.
Their main difference is their implementation which causes **different performance** for different operations.

ArrayList is implemented as a **resizable** array. As more elements are added to ArrayList, its size is increased **dynamically.**
Its elements can be accessed directly by using the **get** and **set** methods, since ArrayList is essentially an array.

LinkedList is implemented as a double linked list. Its **performance** on add and remove is better than Arraylist, but worse on get and set methods.

Vector is similar with ArrayList, but it is **synchronized**.
ArrayList is a better choice if your program is **thread-safe**. Vector and ArrayList require more space as more elements are added. **Vector** each time **doubles** its array size, while ArrayList grow 50% of its size each time.
LinkedList, however, also implements Queue interface which adds more methods than ArrayList and Vector, such as offer(), peek(), poll(), etc.

Note: The default initial capacity of an ArrayList is pretty small. It is a good habit to construct the ArrayList with a higher initial capacity. This can avoid the resizing cost.

@What does T means in java?

<> (Diamond brackets) means Generic class. A Generic Class is a class which can work on any type of data type or in other words we can say it is data type independent, in simple terms....

Now if we look at the syntax of a generic class in java:

```
1. public class Box<T> {
2.     // T stands for "Type"
3.     private T t;
4.
5.     public void set(T t) { this.t = t; }
6.     public T get() { return t; }
7. }
```

Where 'T' means type. Now when you create instance of this Box class you will need to tell the compiler for what data type this will be working on.

ex),
Box<Integer> b1 = new Box();
Box<String> b2 = new Box();

Integer is a type and String is also a type.
<**T**> specifically stands for **generic** type.

@ What is Abstract Window Toolkit (AWT)?

Abstract Window Toolkit (**AWT**) is a set of (API s) used by **Java** programmers to create graphical user interface (GUI) objects, such as buttons, scroll bars, and windows.

@ What is **Static Variable** in Java?

- It is a variable which **belongs to the class** and **not to object(instance)**
- Static variables are **initialized only once**, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables
- A single copy to be shared by all instances of the class
- A static variable can be accessed directly by the class name and doesn't need any object

@ What is **Static Method** in Java?

- It is a method which belongs to the class and not to the object(instance)
- A static method can access only static data. It can not access non-static data (instance variables)
- A static method can call only other static methods and can not call a non-static method from it.
- A static method can be accessed directly by the class name and doesn't need any object
- A static method cannot refer to "this" or "super" keywords in anyway

@What is an Interface?

An interface is just like Java Class, but it only has static constants and abstract method. Java uses Interface to implement multiple inheritance. A Java class can implement multiple Java Interfaces. All methods in an interface are implicitly public and abstract.

Why do we use interface ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

  The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

```
// An example to show that interfaces can
// have default methods from JDK 1.8 onwards

interface in1
{
   final int a = 10;
   default void display()
   {
      System.out.println("hello");
   }
}

// A class that implements interface.
class testClass implements in1
{
   // Driver Code
   public static void main (String[] args)
   {
      testClass t = new testClass();
      t.display();
   }
}
```

Output :

Hello

@JAVA INNER CLASS

Java inner class or nested class is a class which is declared inside the class or interface. We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
1.  class Java_Outer_class{
2.    //code
3.    class Java_Inner_class{
4.      //code
5.    }
6.  }
```

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code**because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Difference between **nested class** and **inner class** in Java

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)

a. Member inner class
b. Anonymous inner class
c. Local inner class

- Static nested clas

Member Inner class : A class created within class and outside method.

Anonymous Inner Class : A class created for implementing interface or extending class. Its name is decided by the java compiler.

Local Inner Class : A class created within method.
Static Nested Class : A static class created within class.
Nested Interface : An interface created within class or interface

Example of java inner class :

```
1. class example{
2.    private int data=30;
3.    class Inner{
4.       void msg(){System.out.println("data is "+data);}
5.    }
6.    public static void main(String args[]){
7.       example obj=new example();
8.       example.Inner in=obj.new Inner();
9.       in.msg();
10.        }
   }
```

Difference between JPanel, JFrame, JComponent, and JApplet

Those classes are common extension points for Java UI designs. First off, realize that they don't necessarily have much to do with each other directly, so trying to find a relationship between them might be counterproductive.

JFrame and **JApplet** are top level **containers**. If you whish to create desktop application, you will use JFrame and if you plan to host your application in browser you will use JApplet.

JComponent is abstract class for all Swing compoents and you can use it as the base class for your new component. **JPanel** is simple usable compoent you can use for almost anything.

Since this is for fun project, the simplest way for you is to work with JPanel and then host it inside JFrame or JApplet. Netbeans has visual designer for Swing with simple examples.

JApplet - A base class that let's you write code that will run within the context of a browser, like for an interactive web page. This is cool and all but it brings limitations which is the price for it playing nice in the real world. Normally JApplet is used when you want to have your own UI in a web page. I've always wondered why people don't take advantage of applets to store state for a session so no database or cookies are needed.

JComponent - A base class for objects which intend to interact with Swing.

JFrame - Used to represent the stuff a window should have. This includes borders (resizeable y/n?), titlebar (App name or other message), controls (minimize/maximize allowed?), and event handlers for various system events like 'window close' (permit app to exit yet?).

JPanel - Generic class used to gather other elements together. This is more important with working with the visual layout or one of the provided layout managers e.g. gridbaglayout, etc. For example, you have a textbox that is bigger then the area you have reserved. Put the textbox in a scrolling pane and put that pane into a JPanel. Then when you place the JPanel, it will be more manageable in terms of layout.

What is Synchronization in Java
by using Java keywords ***"synchronized"*** and ***"volatile"***.

If your code is executing in a multi-threaded environment, you need **sync**hronization for objects,which are shared among multiple threads, to avoid any corruption of state or any kind of unexpected behavior. Synchronization in Java will only be needed if shared object is mutable. if your shared object is either read-only or immutable object, then you don't need synchronization, despite running multiple threads. Same is true with what threads are doing with an object if all the threads are only reading value then you don't require synchronization in Java. JVM guarantees that ***Java synchronized code will only be executed by one thread at a time***.

1) The synchronized keyword in Java provides locking, which ensures mutually exclusive access to the shared resource and prevents data race.

2) synchronized keyword also prevent reordering of code statement by the compiler which can cause a subtle concurrent issue if we don't use synchronized or volatile keyword.

3) synchronized keyword involve locking and unlocking. before entering into **synchronized method or block** thread needs to acquire the lock, at this point it reads data from main memory than cache and when it release the lock, it flushes write operation into main memory which eliminates memory inconsistency errors.

Example of **Synchronized Block** in Java
Using **synchronized block in java** is also similar to using **synchronized keyword in methods**. Only important thing to note here is that if object used to lock synchronized block of code, Singleton.class in below example is null then Java synchronized block will throw a NullPointerException.

```
public class Singleton{

   private static volatile Singleton _instance;

   public static Singleton getInstance(){
      if(_instance == null){
         synchronized(Singleton.class){
            if(_instance == null)
            _instance = new Singleton();
         }
      }
      return _instance;
   }
}
```

This is a classic example of double checked locking in Singleton. In this **example of Java synchronized code,** we have made the only critical section (part of the code which is creating an instance of singleton) synchronized and saved some performance.

If you make the whole method synchronized than every call of this method will be blocked, while you only need blocking to create singleton instance on the first call. By the way, this is not the only way to write threadsafe singleton in Java. You can use Enum, or lazy loading to avoid thread-safety issue during instantiation.

Java Interface Static Method

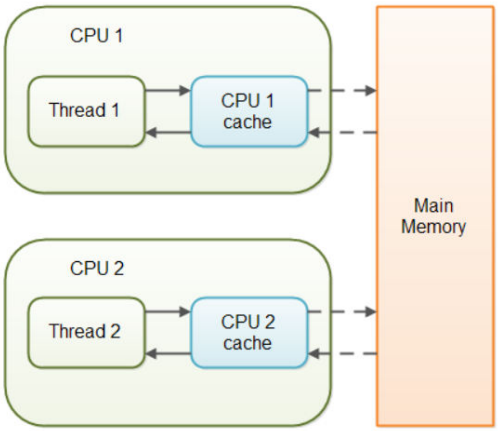
Is similar to default method except that we can't override them in the implementation classes. This feature helps us in avoiding undesired results incase of poor implementation in implementation classes.

Volatile Keyword

Variable Visibility Problems

The Java `volatile` keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.

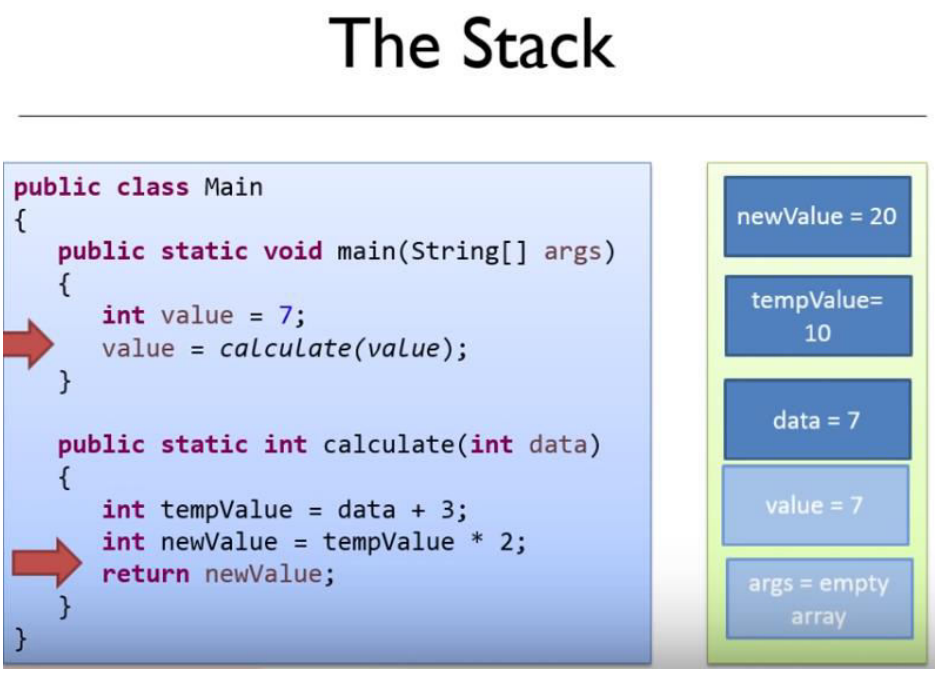
In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs.



Difference between **synchronized** and **volatile** keyword

In short, volatile keyword in Java is not a replacement of synchronized block or method but in some situation is very handy and can save performance overhead which comes with use of synchronization in Java.

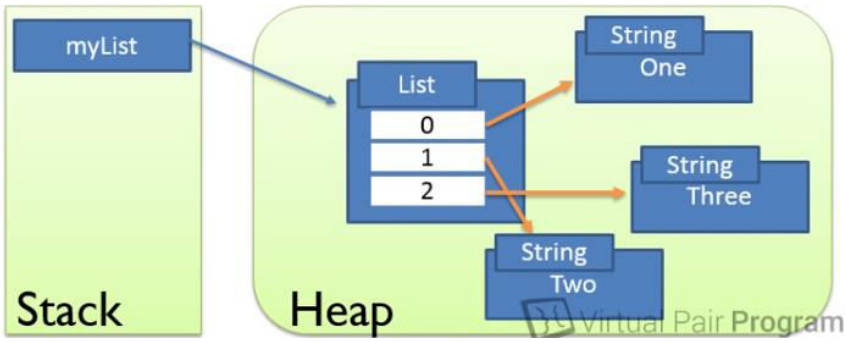
Since **volatile** keyword only synchronizes the value of **one variable** between Thread memory and "main" memory while **synchronized** synchronizes the value of **all variable** between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.



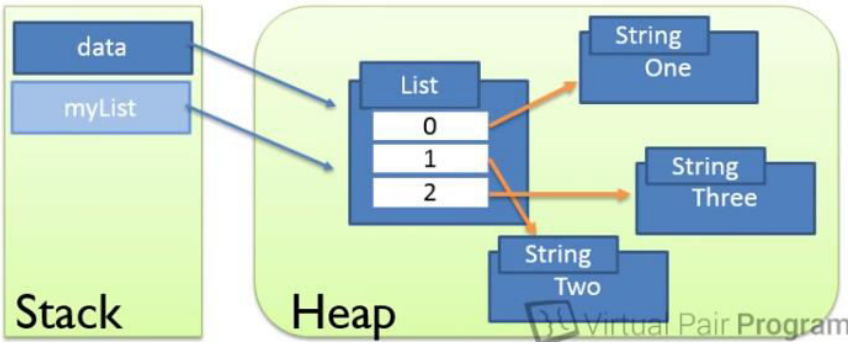
FILO (first in last out) apply for all the local variables in the stack.

Sharing Data

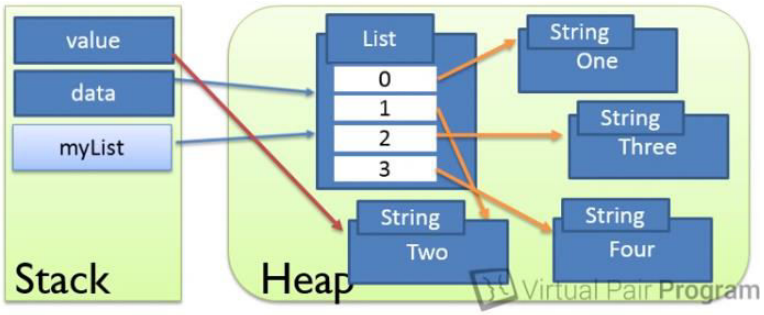
```
public static void main(String[] args) {
      List<String> myList = new ArrayList<String>();
      myList.add("One");
      myList.add("Two");
      myList.add("Three");
      printList(myList);
   }

   public static void printList(List<String> data) {
      System.out.println(data);
   }
```




```
public static void main(String[] args) {
      List<String> myList = new ArrayList<String>();
      public void printList(List<String> data)
      printList(myList);
   }

   public static void printList(List<String> data) {
      System.out.println(data);
   }
```

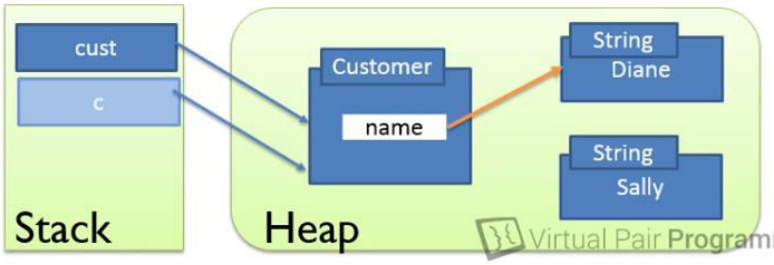


```
public static void printList(List<String> data) {
    String value = data.get(1);
    data.add("Four");
    System.out.println(value);
}
```



Passing Values

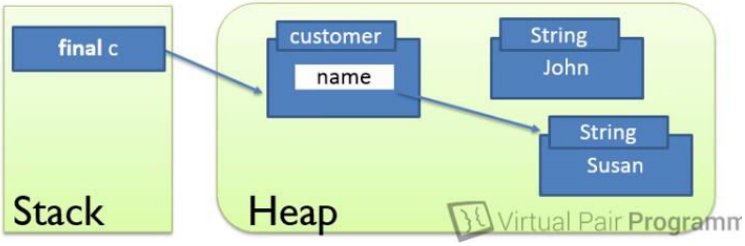
```
public static void main(String[] args) {
        Customer c = new Customer("Sally");
        renameCustomer(c);
        System.out.println(c.getName());
    }

public static void renameCustomer(Customer cust) {
        cust.setName("Diane");
    }
```



The final Keyword

```
final Customer c = new Customer("John");
```

```
final Customer c;
c = new Customer("John");
c.setName("Susan");
```



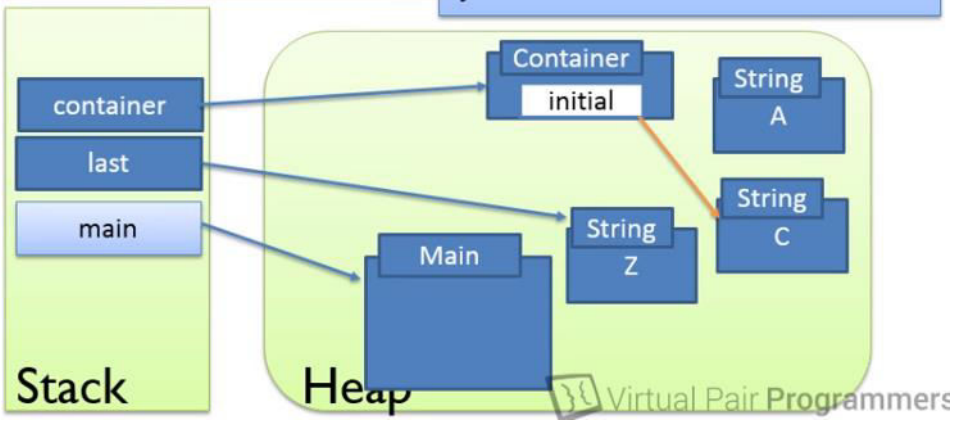
MemoryTest

```
public class Main {

public static void main(String args[]) {
Main main = new Main();
main.start();
}
```

Memo

```
public void start() {
  String last = "Z";
  Container container = new Container(
  container.setInitial("C");
  another(container,last);
  System.out.print(container.getIniti
}
```

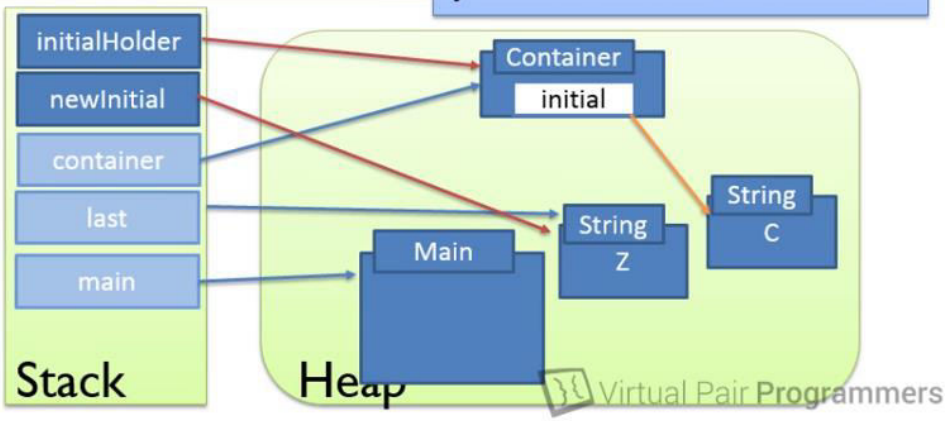
```
public class Container {
  private String initial = "A";

  public String getInitial() {
  return initial;
  }

  public void setInitial
    (String initial) {
  this.initial = initial;
  }
}
```

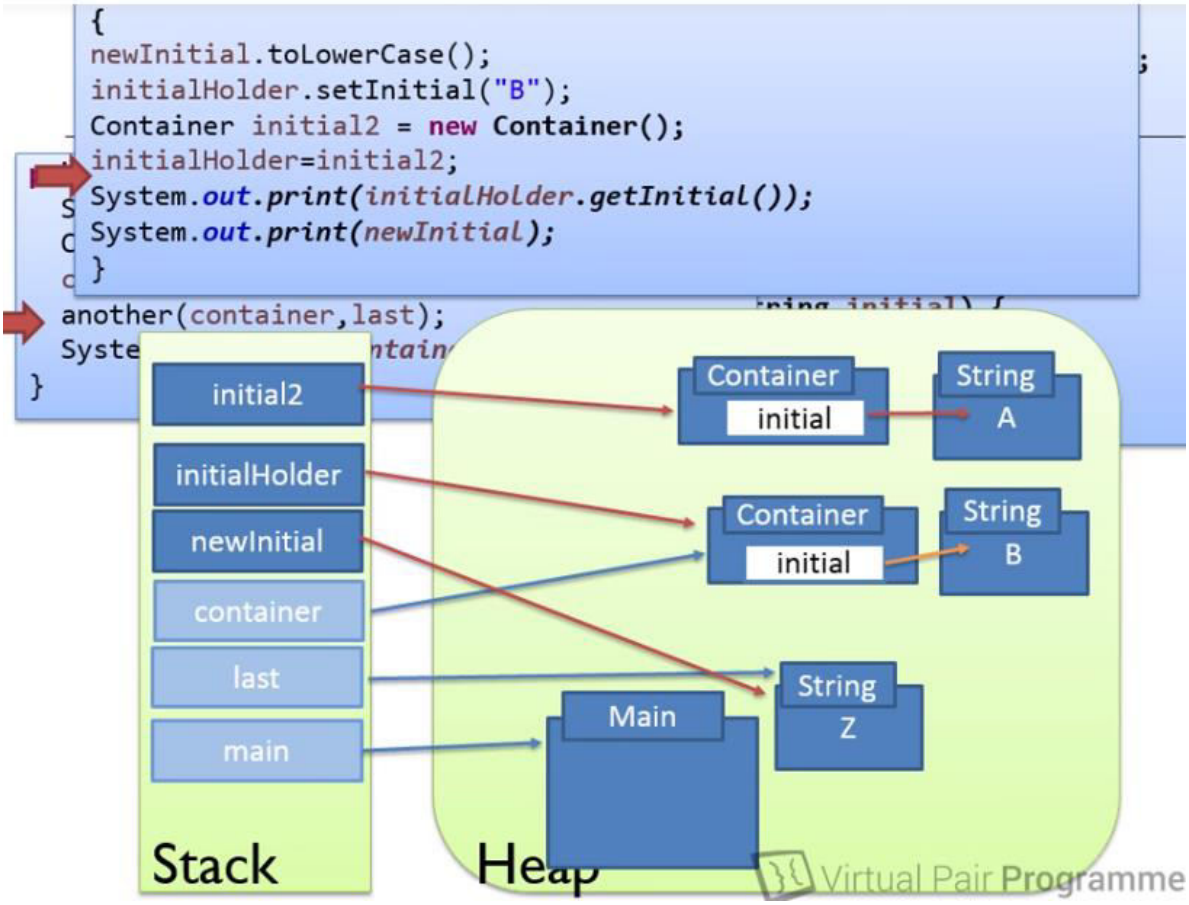



```
{
newInitial.toLowerCase();
initialHolder.setInitial("B");
Container initial2 = new Container();
initialHolder=initial2;
System.out.print(initialHolder.getInitial());
System.out.print(newInitial);
}
```

```
another(container,last);
System.out.print(container.getIniti
}
```

```
  (String initial) {
this.initial = initial;
}
}
```




{
newInitial.toLowerCase();
initialHolder.setInitial("B");
Container initial2 = new Container();
initialHolder=initial2;
System.out.print(initialHolder.getInitial());
System.out.print(newInitial);
}
another(container,last);
}
initial2
initialHolder
newInitial
container
last
main
Stack
Container
initial
String
A
Container
initial
String
B
String
Z
Main
Heap
Virtual Pair Programme