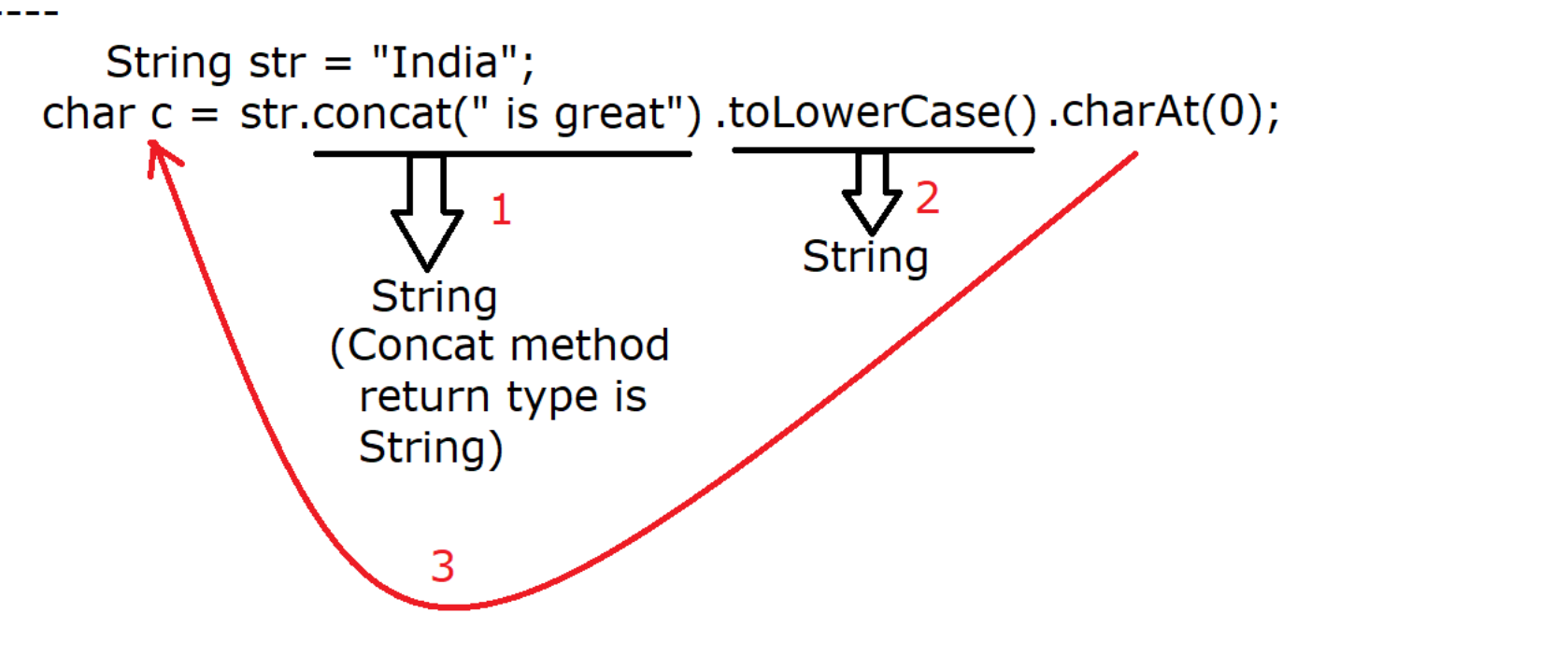
What is Method chaining in java ?
----
* It is a technique through which we can call multiple methods in a single statement.

Example :

I want to read a character from the Scanner class.

Scanner sc = new Scanner(System.in);
System.out.print("Enter a Character :");
char ch = sc.**next().chatAt(0); //Method chaining**

Example :



* In method chaining, When we call another methdo then we need to depend upon the last method return type.

* We are calling so many method so the return type of the method will depend upon the last method call.

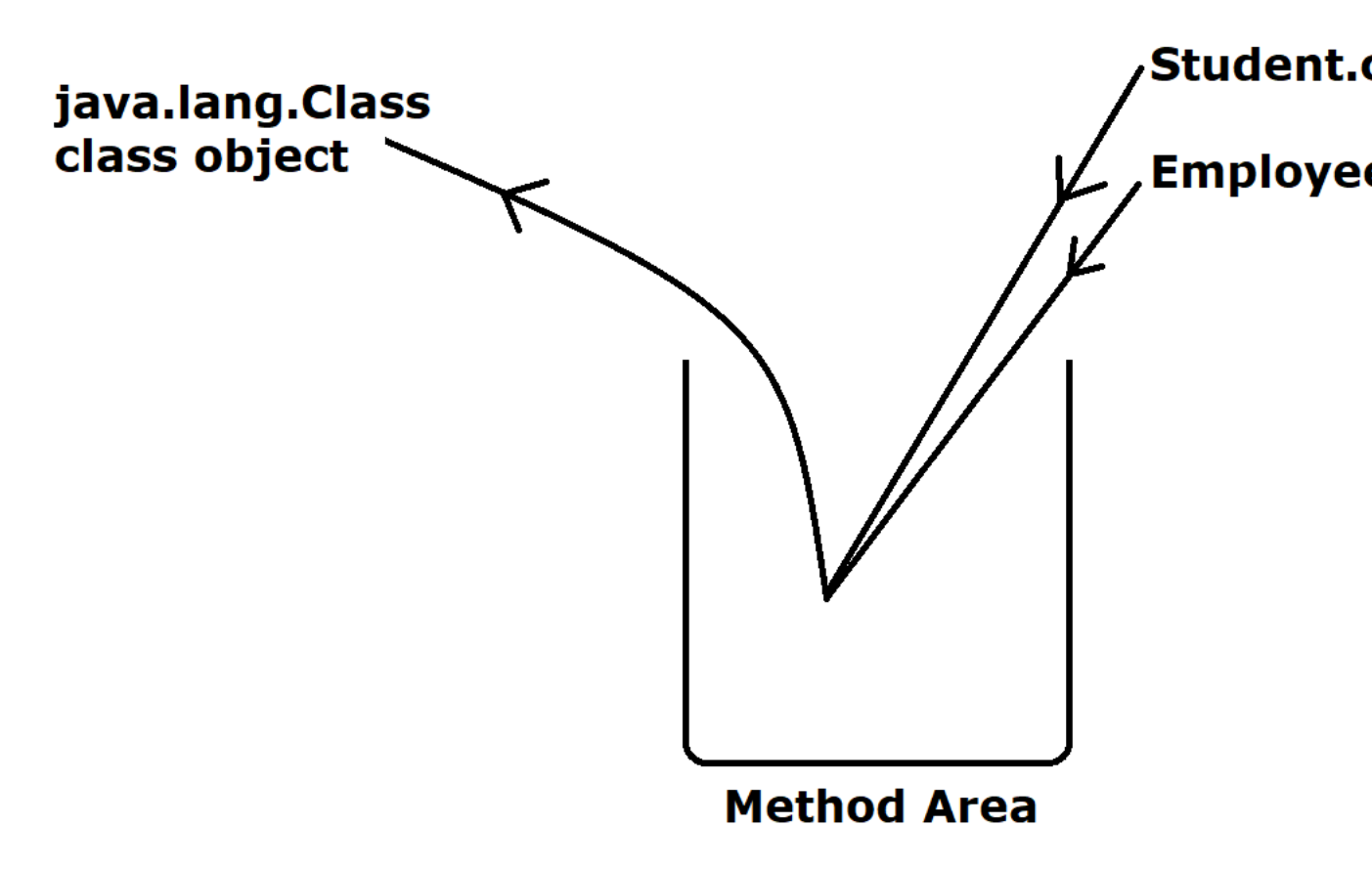
//Program

```java
package com.ravi.method_chaining;

public class MethodChainngDemo1
{
    public static void main(String[] args)
    {
        String str = "India";
        char ch = str.concat(" is great").toLowerCase().charAt(1);
         System.out.println(ch);
    }

}
```

```java
package com.ravi.method_chaining;

public class MethodChainingDemo2
{
    public static void main(String[] args)
    {
        String str = "Hyderabad";
        int length = str.concat(" is an IT city").toUpperCase().length();
        System.out.println(length);
    }
}
```

Role of **java.lang.Class** class in class loading :
----
* In Java, Whenever we load .class file into JVM memory then actually the .class file will be loaded in a very special area called **"Method Area"**

* The .class file is loaded in the Method Area and returns **java.lang.Class class object.**



Class cls = AnyClass.class

* java.lang.Class class has provided a predefined non static method called **getName(),** It will provide the Fully Qualified Name (**FQN**) [FQN = Package Name + class Name] of the specified class.

public String getName();

WAP to show that java.lang.Class class is the return type of any loaded class in the JVM memory :

```java
package com.ravi.jvm_arch;

class Customer{}

class Student {}

class Employee{}


public class RetunTypeOfLoadedClass
{
    public static void main(String[] args)
    {
        Class cls = Customer.class;
        System.out.println("FQN is :"+cls.getName());

        cls = Student.class;
        System.out.println("FQN is :"+cls.getName());

        cls = Employee.class;
        System.out.println("FQN is :"+cls.getName());
    }
}
```

WAP to show that Application class loader is responsible to load user-defined classes :
----
java.lang.Class class has provided a predefined non static factory method called **getClassLoader()** which will provide the name of Class loader which is responsible for loading the specified .class file. The return type of this method is **ClassLoader which is a predefined abstract class.**

public ClassLoader getClassLoader();

```java
package com.ravi.jvm_arch;

public class Test
{
   public static void main(String[] args)
   {
      System.out.println("Test.class file is loaded by
:"+Test.class.getClassLoader());
   }
}
```

WAP to show that Platfrom class loader is the super class of Application class loader :
----
* ClassLoader class has provided a predefined non static method called getParent() which will provide the parent class (super class) name. The return type of this method is again ClassLoader class.

ClassLoader c1 = Test.class.getClassLoader();
ClassLoader c2 = c1.getParent();

```java
package com.ravi.jvm_arch;

public class Demo {

    public static void main(String[] args)
    {
        System.out.println("Super class of Application class loader is :");
        System.out.println(Demo.class.getClassLoader().getParent());
    }
}
```

```java
package com.ravi.jvm_arch;

public class Sample {

    public static void main(String[] args)
    {
        System.out.println(String.class.getClassLoader());

        System.out.println("......................");
        System.out.println("Super class of Platform class loader is :");
        System.out.println(Sample.class.getClassLoader().getParent().getParent());
    }

}
```

Note :- Here we will get the output as null because it is built in class loader for JVM which is used for internal purpose (loading only predefined .class file) so implementation is not provided hence we are getting null.

Linking :
----
verify :-

It ensures the correctness of the .class files, If any suspicious activity is there in the .class file then It will stop the execution immediately by throwing a runtime error i.e java.lang.VerifyError.

There is something called ByteCodeVerifier(Component of JVM), responsible to verify the loaded .class file i.e byte code. Due to this verify module JAVA is highly secure language.

java.lang.VerifyError is the sub class of java.lang.linkageError.

prepare : [Static field memory allocation + Initialization with default value]

[Static variable memory allocation + static variable initialization with default value even the variable is final]

It will allocate the memory for all the static data members, here all the static data member will get the default values so if we have static int x = 100; then for variable x memory will be allocated (4 bytes) and now it will initialize with default value i.e 0, even the variable is final.

static Test t = new Test();

Here, t is a static reference variable so for t variable (reference variable) memory will be allocated as per JVM implementation i.e for 32 bit JVM (4 bytes of Memory) and for 64 bit (8 bytes of memory) and initialized with null.